IBM DB2 10.5
for Linux, UNIX, and Windows

SQL Reference Volume 1

Updated October, 2014

IBM

IBM DB2 10.5
for Linux, UNIX, and Windows

SQL Reference Volume 1
Updated October, 2014

IBM

Edition Notice

Contents

Chapter 2. Language elements 57

Contents **xi**

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.
* "Catalog views" describes the 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.

How this book is structured

- "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.
- "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

This topic describes the structure of SQL syntax diagrams.

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

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

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

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

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

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

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

►►—*required_item*——————————————————————————————►◄

Optional items appear below the main path.

►►—*required_item*——————————————————————————————►◄
       └─*optional_item*─┘

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

       ┌─*optional_item*─┐
►►—*required_item*─┴─────────────┴──————————————————►◄
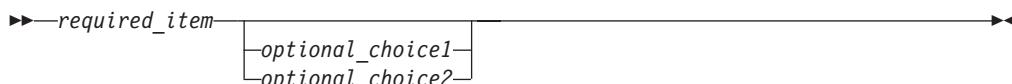
If you can choose from two or more items, they appear in a stack.

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

►►—*required_item*─┬─*required_choice1*─┬───————————►◄
          └─*required_choice2*─┘
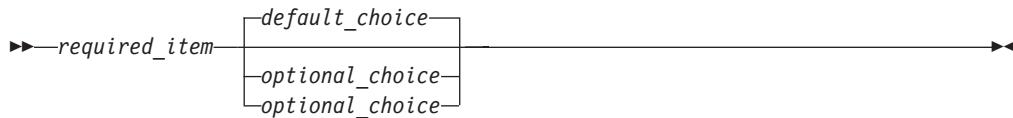
If choosing one of the items is optional, the entire stack appears below the main path.

►►—*required_item*─┬─────────────────┬──————————————►◄
          ├─*optional_choice1*─┤
          └─*optional_choice2*─┘

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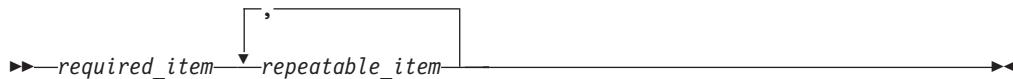
If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.

```
►►──required_item──┬──default_choice──┬──────────────────────►◄
                   ├──optional_choice──┤
                   └──optional_choice──┘
```

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

```
        ┌─────────────┐
►►──required_item──▼──repeatable_item──┴───────────────────────►◄
```

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

```
        ┌──────,──────┐
►►──required_item──▼──repeatable_item──┴───────────────────────►◄
```

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

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

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

Sometimes a single variable represents a 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**:

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

parameter-block:

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

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

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

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

This topic covers the conventions used in the SQL Reference.

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

Conventions describing Unicode data

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

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

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.

Related documentation

- *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.

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 (built-in 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 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.

Introduction to DB2 Call Level Interface and ODBC

DB2 Call Level Interface (CLI) is IBM's callable SQL interface to the DB2 family of database servers. It is a 'C' and 'C++' application programming interface for relational database access that uses function calls to pass dynamic SQL statements as function arguments.

CLI is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require host variables or a precompiler. 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 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.

CLI is based on the Microsoft Open Database Connectivity (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2 specific extensions have been added to help the application programmer specifically exploit DB2 features.

The CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. It conforms to ODBC 3.51.

CLI Background information

To understand CLI or any callable SQL interface, it is helpful to understand what it is based on, and to compare it with existing interfaces.

The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the *X/Open Call Level Interface*. The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database vendor's programming interface. Most of the X/Open Call Level Interface specification has been accepted as part of the ISO Call Level Interface International Standard (ISO/IEC 9075-3:1995 SQL/CLI).

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for Microsoft operating systems based on a preliminary draft of X/Open CLI.

The ODBC specification also includes an operating environment where database-specific ODBC drivers are dynamically loaded at run time by a driver manager based on the data source (database name) provided on the connect request. The application is linked directly to a single driver manager library rather than to each DBMS's library. The driver manager mediates the application's function calls at run time and ensures they are directed to the appropriate DBMS-specific ODBC driver. Because the ODBC driver manager only knows about the ODBC-specific functions, DBMS-specific functions cannot be accessed in an ODBC environment. DBMS-specific dynamic SQL statements are supported through a mechanism called an escape clause.

ODBC is not limited to Microsoft operating systems; other implementations are available on various platforms.

The CLI load library can be loaded as an ODBC driver by an ODBC driver manager. For ODBC application development, you must obtain an ODBC Software Development Kit. For the Windows platform, the ODBC SDK is available as part of the Microsoft Data Access Components (MDAC) SDK, available for download from http://www.microsoft.com/downloads. For non-Windows platforms, the ODBC SDK is provided by other vendors. When developing ODBC applications that may connect to DB2 servers, use the Call Level Interface Guide and Reference Volume 1 and the Call Level Interface Guide and Reference Volume 2 (for information about DB2 specific extensions and diagnostic information), in conjunction with the ODBC Programmer's Reference and SDK Guide available from Microsoft.

Applications written using CLI APIs link directly to the CLI library. CLI includes support for many ODBC and ISO SQL/CLI functions, as well as DB2 specific functions.

The following DB2 features are available to both ODBC and CLI applications:
- double byte (graphic) data types
- stored procedures
- Distributed Unit of Work (DUOW), two phase commit
- compound SQL
- user defined types (UDT)

- user defined functions (UDF)

Java application development for IBM data servers

The DB2 and IBM Informix® database systems provide driver support for client applications and applets that are written in Java™.

You can access data in DB2 and IBM Informix database systems using JDBC, SQL, or pureQuery.

JDBC

JDBC is an application programming interface (API) that Java applications use to access relational databases. IBM data server support for JDBC lets you write Java applications that access local DB2 or IBM Informix data or remote relational data on a server that supports DRDA®.

SQLJ

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM, Oracle, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.

For connections to DB2, in general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL.

For connections to IBM Informix, SQL statements in JDBC or SQLJ applications run dynamically.

Because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

pureQuery

pureQuery is a high-performance data access platform that makes it easier to develop, optimize, secure, and manage data access. It consists of:
- Application programming interfaces that are built for ease of use and for simplifying the use of best practices
- Development tools, which are delivered in IBM Data Studio, for Java and SQL development
- A runtime, which is delivered in IBM InfoSphere® Optim™ pureQuery® Runtime, for optimizing and securing database access and simplifying management tasks

With pureQuery, you can write Java applications that treat relational data as objects, whether that data is in databases or JDBC DataSource objects. Your applications can also treat objects that are stored in in-memory Java collections as though those objects are relational data. To query or update your relational data or Java objects, you use SQL.

For more information on pureQuery, see the Integrated Data Management Information Center.

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

The owner of an implicitly created schema is SYSIBM. When the database is restrictive, PUBLIC does not have the CREATEIN privilege on the schema. The user who implicitly creates the schema has CREATEIN privilege on the schema. When the database is not restrictive, PUBLIC has the CREATEIN privilege on the schema.

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, and drop objects in the schema. This ability 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, except in a restrictive database environment. A user with ACCESSCTRL or SECADM authority can change the privileges that are held by users on any schema. Therefore, access to create, alter, 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.

Types of tables

DB2 databases store data in tables. In addition to tables used to store persistent data, there are also tables that are used for presenting results, summary tables and temporary tables; multidimensional clustering tables offer specific advantages in a warehouse environment.

Base tables
These types of tables hold persistent data. There are different kinds of base tables, including

Regular tables
Regular tables with indexes are the "general purpose" table choice.

Multidimensional clustering (MDC) tables
These types of tables are implemented as tables that are physically clustered on more than one key, or dimension, at the same time. MDC tables are used in data warehousing and large database environments. Clustering indexes on regular tables support single-dimensional clustering of data. MDC tables provide the benefits of data clustering across more than one dimension. MDC tables provide *guaranteed clustering* within the composite dimensions. By contrast, although you can have a clustered index with regular tables, clustering in this case is attempted by the

database manager, but not guaranteed and it typically degrades over time. MDC tables can coexist with partitioned tables and can themselves be partitioned tables.

Multidimensional clustering tables are not supported in a DB2 pureScale® environment.

Insert time clustering (ITC) tables

These types of tables are conceptually, and physically similar to MDC tables, but rather than being clustered by one or more user specified dimensions, rows are clustered by the time they are inserted into the table. ITC tables can be partitioned tables.

ITC tables are not supported in a DB2 pureScale environment.

Range-clustered tables (RCT)

These types of tables are implemented as sequential clusters of data that provide fast, direct access. Each record in the table has a predetermined record ID (RID) which is an internal identifier used to locate a record in a table. RCT tables are used where the data is tightly clustered across one or more columns in the table. The largest and smallest values in the columns define the range of possible values. You use these columns to access records in the table; this is the most optimal method of using the predetermined record identifier (RID) aspect of RCT tables.

Range-clustered tables are not supported in a DB2 pureScale environment.

Partitioned tables

These types of tables use 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 partitions can be added to, attached to, and detached from a partitioned table, and you can store multiple data partition ranges from a table in one table space. Partitioned tables can contain large amounts of data and simplify the rolling in and rolling out of table data.

Temporal tables

These types of tables are used to associate time-based state information to your data. Data in tables that do not use temporal support represents the present, while data in temporal tables is valid for a period defined by the database system, customer applications, or both. For example, a database can store the history of a table (deleted rows or the original values of rows that have been updated) so you can query the past state of your data. You can also assign a date range to a row of data to indicate when it is deemed to be valid by your application or business rules.

Temporary tables

These types of tables are used as temporary work tables for various database operations. *Declared temporary tables* (DGTTs) do not appear in the system catalog, which makes them not persistent for use by, and not able to be shared with other applications. When the application using this table terminates or disconnects from the database, any data in the table is deleted and the table is dropped. By contrast, *created temporary tables* (CGTTs) do appear in the system catalog and are not required to be

defined in every session where they are used. As a result, they are persistent and able to be shared with other applications across different connections.

Neither type of temporary table supports
- User-defined reference or user-defined structured type columns
- LONG VARCHAR columns

In addition XML columns cannot be used in created temporary tables.

Materialized query tables
These types of tables are defined by a query that is also used to determine the data in the table. Materialized query tables can be used to improve the performance of queries. If the database manager determines that a portion of a query can be resolved using a summary table, the database manager can rewrite the query to use the summary table. This decision is based on database configuration settings, such as the CURRENT REFRESH AGE and the CURRENT QUERY OPTIMIZATION special registers. A summary table is a specialized type of materialized query table.

You can create all of the preceding types of tables using the CREATE TABLE statement.

Depending on what your data is going to look like, you might find one table type offers specific capabilities that can optimize storage and query performance. For example, if you have data records that are loosely clustered (not monotonically increasing), consider using a regular table and indexes. If you have data records that have duplicate (but not unique) values in the key, do not use a range-clustered table. Also, if you cannot afford to preallocate a fixed amount of storage on disk for the range-clustered tables you might want, do not use this type of table. If you have data that has the potential for being clustered along multiple dimensions, such as a table tracking retail sales by geographic region, division and supplier, a multidimensional clustering table might suit your purposes.

In addition to the various table types described previously, you also have options for such characteristics as *partitioning*, which can improve performance for tasks such as rolling in table data. Partitioned tables can also hold much more information than a regular, nonpartitioned table. You can also use capabilities such as *compression*, which can help you significantly reduce your data storage costs.

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 pointers that are logically ordered by the values of one or more keys. The pointers can refer to rows in a table, blocks in an MDC or ITC table, XML data in an XML storage object, and so on.

Indexes are used 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, it is appended to the bottom (unless other actions have been carried out on the table or the data being added). There is no inherent 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.

Typically, when you search for data in a table, you are looking for rows with columns that have specific values. A column value in a row of data can be used to identify the entire row. For example, an employee number would probably uniquely define a specific individual employee. Or, more than one column might be needed to identify the row. For example, a combination of customer name and telephone number. Columns in an index used to identify data rows are known as *keys*. 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.

Not all indexes point to rows in a table. MDC and ITC 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 and ITC block indexes and XML indexes are system generated indexes.

Example

Table A in Figure 1 on page 10 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.

Indexes

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.

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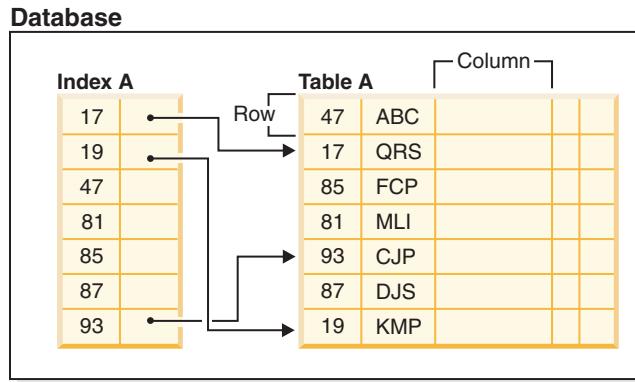
Figure 1. Relationship between an index and a table

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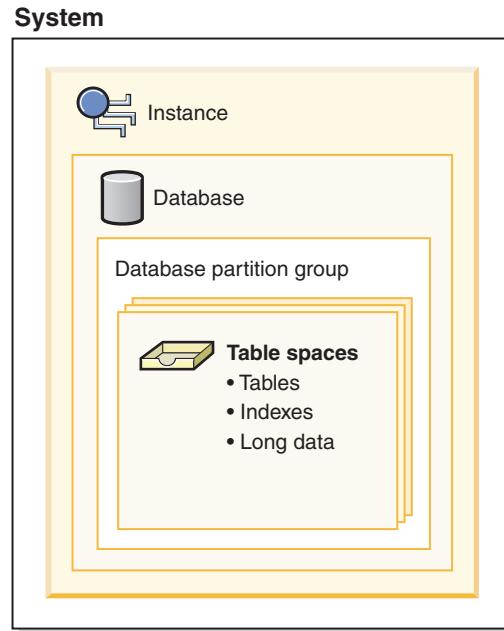
Figure 2. Relationships among selected database objects

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), or activation time (BEFORE, AFTER, INSTEAD OF). 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.

Triggers

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 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 on page 13 shows the relationship between tables and views.

Database
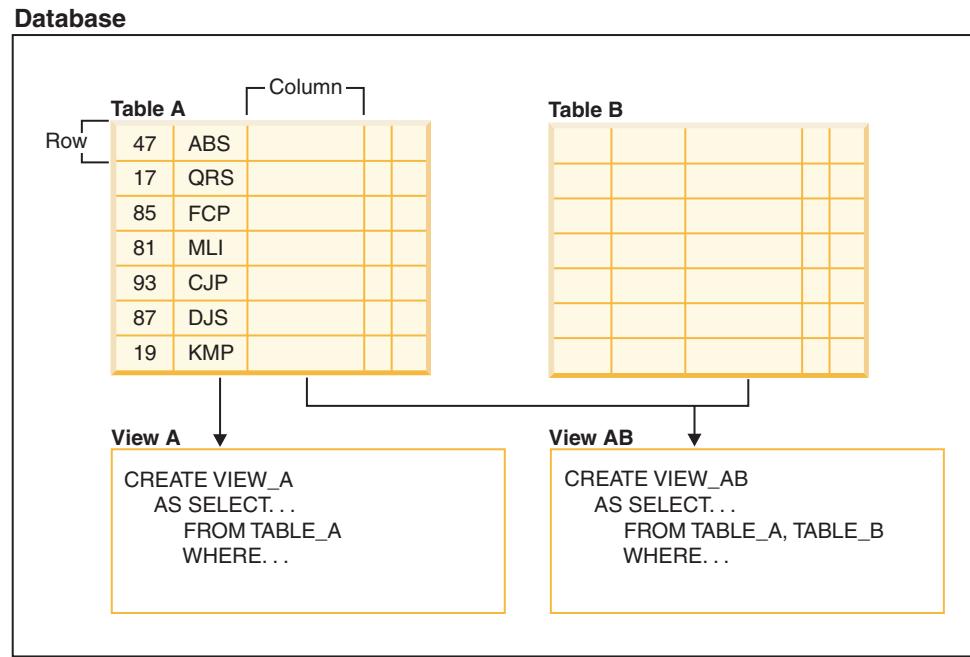
Figure 3. Relationship between tables and views

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.

Aliases

An *alias* is an alternative name for an object such as a module, table or another alias. It can be used to reference an object wherever that object can be referenced directly.

Aliases

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 but it can reference a created temporary table.

Like other objects, an alias can be created, dropped, and have comments associated with it. Aliases can refer to other aliases in a process called *chaining* as long as there are no circular references. Aliases do not require any special authority or privilege to use them. Access to the object referred to by an alias, however, does require the authorization associated with that object.

If an alias is defined as a *public* alias, it can be referenced by its unqualified name without any impact from the current default schema name. It can also be referenced using the qualifier SYSPUBLIC.

Synonym is an alternative name for alias.

For more information, refer to "Aliases in identifiers" in the *SQL Reference Volume 1*.

Authorization, privileges, and object ownership

Users (identified by an authorization ID) can successfully execute operations 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.

The database manager requires that each user be specifically authorized to use each database function needed to perform a specific task. A user can acquire the necessary authorization through a grant of that authorization to their user ID or through membership in a role or a group that holds that authorization.

There are three forms of authorization, *administrative authority*, *privileges*, and *LBAC credentials*. In addition, ownership of objects brings with it a degree of authorization on the objects created. These forms of authorization are discussed in the following section.

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.

System-level authorization

The system-level authorities provide varying degrees of control over instance-level functions:

- SYSADM (system administrator) authority

  The SYSADM (system administrator) authority provides control over all the resources created and maintained by the database manager. The system administrator possesses all the authorities of SYSCTRL, SYSMAINT, and SYSMON 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.

- SYSCTRL authority

  The SYSCTRL authority 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.

- SYSMAINT authority

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

- SYSMON (system monitor) authority

  The SYSMON (system monitor) authority provides the authority required to use the database system monitor.

Database-level authorization

The database level authorities provide control within the database:

- DBADM (database administrator)

  The DBADM authority level provides administrative authority over a single database. This database administrator possesses the privileges required to create objects and issue database commands.

  The DBADM authority can be granted only by a user with SECADM authority. The DBADM authority cannot be granted to PUBLIC.

- SECADM (security administrator)

  The SECADM authority level provides administrative authority for security over a single database. The security administrator authority possesses the ability to manage database security objects (database roles, audit policies, trusted contexts, security label components, and security labels) and grant and revoke all database privileges and authorities. A user with 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. It can only be granted by a user with SECADM authority. The SECADM authority cannot be granted to PUBLIC.

- SQLADM (SQL administrator)

  The SQLADM authority level provides administrative authority to monitor and tune SQL statements within a single database. It can be granted by a user with ACCESSCTRL or SECADM authority.

- WLMADM (workload management administrator)

  The WLMADM authority provides administrative authority to manage workload management objects, such as service classes, work action sets, work class sets, and workloads. It can be granted by a user with ACCESSCTRL or SECADM authority.

- EXPLAIN (explain authority)

  The EXPLAIN authority level provides administrative authority to explain query plans without gaining access to data. It can only be granted by a user with ACCESSCTRL or SECADM authority.

- ACCESSCTRL (access control authority)

  The ACCESSCTRL authority level provides administrative authority to issue the following GRANT (and REVOKE) statements.

  – GRANT (Database Authorities)

ACCESSCTRL authority does not give the holder the ability to grant ACCESSCTRL, DATAACCESS, DBADM, or SECADM authority. Only a user who has SECADM authority can grant these authorities.

– GRANT (Global Variable Privileges)
– GRANT (Index Privileges)
– GRANT (Module Privileges)
– GRANT (Package Privileges)
– GRANT (Routine Privileges)
– GRANT (Schema Privileges)
– GRANT (Sequence Privileges)
– GRANT (Server Privileges)
– GRANT (Table, View, or Nickname Privileges)
– GRANT (Table Space Privileges)
– GRANT (Workload Privileges)
– GRANT (XSR Object Privileges)

ACCESSCTRL authority can only be granted by a user with SECADM authority. The ACCESSCTRL authority cannot be granted to PUBLIC.

• DATAACCESS (data access authority)

The DATAACCESS authority level provides the following privileges and authorities.

– LOAD authority
– SELECT, INSERT, UPDATE, DELETE privilege on tables, views, nicknames, and materialized query tables
– EXECUTE privilege on packages
– EXECUTE privilege on modules
– EXECUTE privilege on routines

Except on the audit routines: AUDIT_ARCHIVE, AUDIT_LIST_LOGS, AUDIT_DELIM_EXTRACT.

– READ privilege on all global variables and WRITE privilege on all global variables except variables which are read-only
– USAGE privilege on all XSR objects
– USAGE privilege on all sequences

It can be granted only by a user who holds SECADM authority. The DATAACCESS authority cannot be granted to PUBLIC.

• Database authorities (non-administrative)

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).

Privileges

A privilege is a permission to perform an action or a task. 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 apples to tables, views, nicknames, indexes, and packages.

If a different user requires the CONTROL privilege to that object, a user with SECADM or ACCESSCTRL 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.

Individual privileges: Individual privileges can be granted to allow a user to carry out specific tasks on specific objects. Users with the administrative authorities ACCESSCTRL or SECADM, or with 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 SECADM authority, ACCESSCTRL 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 (except on the audit routines: AUDIT_ARCHIVE, AUDIT_LIST_LOGS, AUDIT_DELIM_EXTRACT).

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.

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.

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 ACCESSCTRL or SECADM 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 those privileges are with the WITH GRANT OPTION, where supported. Therefore the object owner can provide these privileges to other users by using the GRANT statement. For example, if USER1 creates a table space, USER1 automatically has the USEAUTH privilege with the WITH GRANT OPTION on this table space and can grant the USEAUTH privilege to other users. 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 ACCESSCTRL or SECADM 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 you use the **BIND** command to create a package and you specify the **OWNER** *authorization id* option, the owner of objects created by the static SQL statements in the package is the value of *authorization id*. 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*.

System catalog views

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 might involve the execution of different programs, or different executions of the same program.

More than one application process can request access to the same data at the same time. *Locking* is the mechanism that is 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 that were acquired during a unit of work and also commits database changes that were made during the unit of work.

A *unit of work* (UOW) is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process starts, or when the previous UOW ends because of something other than the termination of the application process. A unit of work ends with a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes that were made within the UOW that is ending.

The database manager provides a means of backing out of uncommitted changes that were 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 lock timeout situation. An application process can explicitly request that its database changes be cancelled. This is done using a *rollback* operation.

As long as these changes remain uncommitted, other application processes are unable to see them, and the changes can be rolled back. This is not true, however, if the prevailing isolation level is uncommitted read (UR). After they are committed, these database changes are accessible to other application processes and can no longer be rolled back.

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 that are acquired by the database manager on behalf of an application process are held until the end of a UOW, except 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).

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 might affect the operation of a concurrent transaction.

The initiation and the termination of a UOW define *points of consistency* within an application process. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first 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 UOW, thereby making the changes available to other application processes. If a failure occurs before the UOW ends, the database manager will roll back any uncommitted changes to restore data consistency.

Point of consistency   New point of consistency

← one unit of work →

TIME LINE   database updates

Begin unit of work   Commit End unit of work
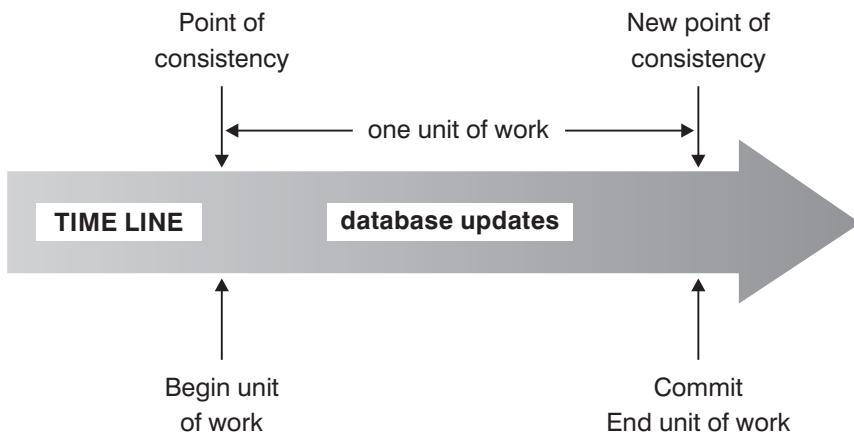
Figure 4. Unit of work with a COMMIT statement

Point of consistency   New point of consistency

← one unit of work →

TIME LINE   database updates   back out updates

Begin unit of work   Failure; Begin rollback   Data is returned to its initial state; End unit of work
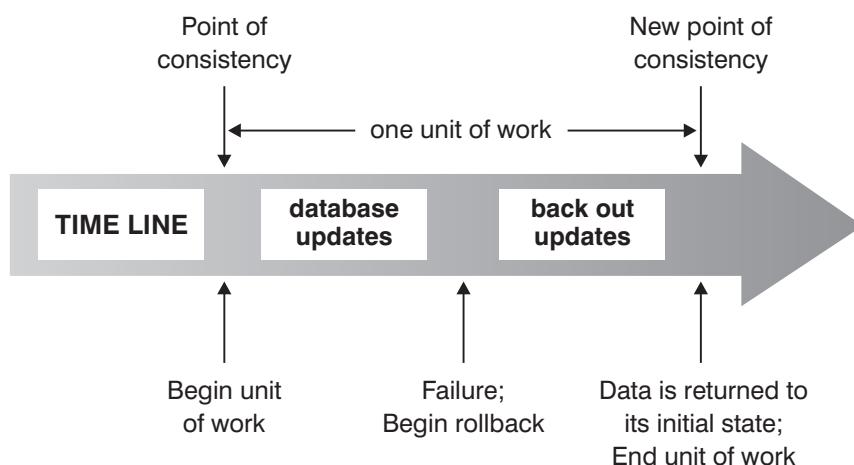
Figure 5. Unit of work with a ROLLBACK statement

Isolation levels

The *isolation level* that is associated with an application process determines the degree to which the data that is being accessed by that process is locked or isolated from other concurrently executing processes. The isolation level is in effect for the duration of a unit of work.

The isolation level of an application process therefore specifies:
- The degree to which rows that are read or 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 that package. The isolation level is specified during the program preparation process by setting the ISOLATION bind or precompile option. For dynamic SQL statements, the default isolation level is the isolation level that was specified for the package preparing the statement. Use the SET CURRENT ISOLATION statement to specify a different isolation level for dynamic SQL statements that are 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".

Isolation levels are enforced by locks, and the type of lock that is used 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 (S)

Under an S lock, concurrent application processes are limited to read-only operations on the data.

Update (U)

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

Exclusive (X)

Under an X lock, concurrent application processes are prevented from accessing the data in any way. This does not apply to application processes with an isolation level of uncommitted read (UR), which can read but not modify the data.

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 an application process during a unit of work is not changed by any other application process until the unit of work is complete.

The database manager supports four isolation levels.
- "Repeatable read (RR)" on page 23
- "Read stability (RS)" on page 23

- "Cursor stability (CS)" on page 24
- "Uncommitted read (UR)" on page 25

Note: Some host database servers support the *no commit (NC)* isolation level. On other database servers, this isolation level behaves like the uncommitted read isolation level.

Note: The *lost updates (LU)* concurrency issue is not allowed by any of the DB2 product's four isolation levels.

A detailed description of each isolation level follows, in decreasing order of performance impact, but in increasing order of the care that is required when accessing or updating data.

Repeatable read (RR)

The *repeatable read* isolation level locks all the rows that an application references during a unit of work (UOW). If an application issues a SELECT statement twice within the same unit of work, the same result is returned each time. Under RR, lost updates, access to uncommitted data, non-repeatable reads, and phantom reads are not possible.

Under RR, an application can retrieve and operate on the rows as many times as necessary until the UOW completes. However, no other application can update, delete, or insert a row that would affect the result set until the UOW completes. Applications running under the RR isolation level cannot see the uncommitted changes of other applications. This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

Every referenced row is locked, not just the rows that are retrieved. For example, if you scan 10 000 rows and apply predicates to them, locks are held on all 10 000 rows, even if, say, only 10 rows qualify. Another application cannot insert or update a row that would be added to the list of rows referenced by a query if that query were to be executed again. This prevents phantom reads.

Because RR can acquire a considerable number of locks, this number might exceed limits specified by the `locklist` and `maxlocks` database configuration parameters. To avoid lock escalation, the optimizer might elect to acquire a single table-level lock for an index scan, if it appears that lock escalation is likely. If you do not want table-level locking, use the read stability isolation level.

While evaluating referential constraints, the DB2 server might occasionally upgrade the isolation level used on scans of the foreign table to RR, regardless of the isolation level that was previously set by the user. This results in additional locks being held until commit time, which increases the likelihood of a deadlock or a lock timeout. To avoid these problems, create an index that contains only the foreign key columns, which the referential integrity scan can use instead.

Read stability (RS)

The *read stability* isolation level locks only those rows that an application retrieves during a unit of work. RS ensures that any qualifying row read during a UOW cannot be changed by other application processes until the UOW completes, and that any change to a row made by another application process cannot be read until the change is committed by that process. Under RS, access to uncommitted data

and non-repeatable reads are not possible. However, phantom reads are possible. Phantom reads might also be introduced by concurrent updates to rows where the old value did not satisfy the search condition of the original application but the new updated value does.

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 a DB2 pureScale environment, an application running at this isolation level might reject a previously committed row value if the row is updated concurrently on a different member. To override this behavior, specify the WAIT_FOR_OUTCOME option.

This isolation level ensures that all returned data remains unchanged until the time the application sees the data, even when temporary tables or row blocking is used.

The RS isolation level provides both a high degree of concurrency and a stable view of the data. To that end, the optimizer ensures that table-level locks are not obtained until lock escalation occurs.

The RS isolation level is suitable for an application that:
• Operates in a concurrent environment
• Requires qualifying rows to remain stable for the duration of a unit of work
• Does not issue the same query more than once during a unit of work, or does not require the same result set when a query is issued more than once during a unit of work

Cursor stability (CS)

The *cursor stability* isolation level locks any row being accessed during a transaction while the cursor is positioned on that row. This lock remains in effect until the next row is fetched or the transaction terminates. However, if any data in the row was changed, the lock is held until the change is committed.

Under this isolation level, no other application can update or delete a row while an updatable cursor is positioned on that row. Under CS, access to the uncommitted data of other applications is not possible. However, non-repeatable reads and phantom reads are possible.

CS is the default isolation level. It is suitable when you want maximum concurrency and need to see only committed data.

In a DB2 pureScale environment, an application running at this isolation level may return or reject a previously committed row value if the row is concurrently updated on a different member. The WAIT FOR OUTCOME option of the concurrent access resolution setting can be used to override this behavior.

Note: Under the *currently committed* semantics introduced in Version 9.7, only committed data is returned, as was the case previously, but now readers do not

wait for updaters to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.

Uncommitted read (UR)

The *uncommitted read* isolation level allows an application to access the uncommitted changes of other transactions. Moreover, UR does not prevent another application from accessing a row that is being read, unless that application is attempting to alter or drop the table.

Under UR, access to uncommitted data, non-repeatable reads, and phantom reads are possible. This isolation level is suitable if you run queries against read-only tables, or if you issue SELECT statements only, and seeing data that has not been committed by other applications is not a problem.

UR works differently for read-only and updatable cursors.
- Read-only cursors can access most of the uncommitted changes of other transactions.
- Tables, views, and indexes that are being created or dropped by other transactions are not available while the transaction is processing. Any other changes by other transactions can be read before they are committed or rolled back. Updatable cursors operating under UR behave as though the isolation level were CS.

If an uncommitted read application uses ambiguous cursors, it might use the CS isolation level when it runs. The ambiguous cursors can be escalated to CS if the value of the BLOCKING option on the **PREP** or **BIND** command is UNAMBIG (the default). To prevent this escalation:
- Modify the cursors in the application program to be unambiguous. Change the SELECT statements to include the FOR READ ONLY clause.
- Let the cursors in the application program remain ambiguous, but precompile the program or bind it with the BLOCKING ALL and STATICREADONLY YES options to enable the ambiguous cursors to be treated as read-only when the program runs.

Comparison of isolation levels

Table 1 summarizes the supported isolation levels.

Table 1. Comparison of isolation levels

	UR	CS	RS	RR
Can an application see uncommitted changes made by other application processes?	Yes	No	No	No
Can an 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? [1]	Yes	Yes	Yes	No [2]
Can updated rows be updated by other application processes? [3]	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

Table 1. Comparison of isolation levels (continued)

	UR	CS	RS	RR
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? [4]	Yes	Yes	No	No
Can accessed rows be read by other application processes?	Yes	Yes	Yes	Yes
Can the current row be updated or deleted by other application processes? [5]	Yes/No [6]	Yes/No [6]	No	No

Note:

1. An example of the *phantom read phenomenon* is as follows: Unit of work UW1 reads the set of *n* rows that satisfies some search condition. Unit of work UW2 inserts one or more rows that satisfy the same search condition and then commits. If UW1 subsequently repeats its read with the same search condition, it sees a different result set: the rows that were read originally plus the rows that were inserted by UW2.

2. If your label-based access control (LBAC) credentials change between reads, results for the second read might be different because you have access to different rows.

3. The isolation level offers no protection to the application if the application is both reading from and writing to 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 might see inconsistent data when more rows are fetched from the open cursor.

4. An example of the *non-repeatable read phenomenon* is as follows: Unit of work UW1 reads a row. Unit of work UW2 modifies that row and commits. If UW1 subsequently reads that row again, it might see a different value.

5. An example of the *dirty read phenomenon* is as follows: Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 commits. If UW1 subsequently rolls the changes back, UW2 has read nonexisting data.

6. Under UR or CS, if the cursor is not updatable, the current row can be updated or deleted by other application processes in some cases. For example, buffering might cause the current row at the client to be different from the current row at the server. Moreover, when using currently committed semantics under CS, a row that is being read might have uncommitted updates pending. In this case, the currently committed version of the row is always returned to the application.

Summary of isolation levels

Table 2 lists the concurrency issues that are associated with different isolation levels.

Table 2. Summary of isolation levels

Isolation level	Access to uncommitted data	Non-repeatable reads	Phantom reads
Repeatable read (RR)	Not possible	Not possible	Not possible
Read stability (RS)	Not possible	Not possible	Possible
Cursor stability (CS)	Not possible	Possible	Possible
Uncommitted read (UR)	Possible	Possible	Possible

The isolation level affects not only the degree of isolation among applications but also the performance characteristics of an individual application, because the

processing and memory resources that are required to obtain and free locks vary with the isolation level. The potential for deadlocks also varies with the isolation level. Table 3 provides a simple heuristic to help you choose an initial isolation level for your application.

Table 3. Guidelines for choosing an isolation level

Application type	High data stability required	High data stability not required
Read-write transactions	RS	CS
Read-only transactions	RR or RS	UR

Table spaces

A *table space* is a storage structure containing tables, indexes, large objects, and long data. They are used to organize data in a database into logical storage groupings that relate to where data is stored on a system. Table spaces are stored in database partition groups.

Using table spaces to organize storage offers a number of benefits:

Recoverability
Putting objects that must be backed up or restored together into the same table space makes backup and restore operations more convenient, since you can backup or restore all the objects in table spaces with a single command. If you have partitioned tables and indexes that are distributed across table spaces, you can backup or restore only the data and index partitions that reside in a given table space.

More tables
There are limits to the number of tables that can be stored in any one table space; if you have a need for more tables than can be contained in a table space, you need only to create additional table spaces for them.

Automatic storage management
With automatic storage table spaces table spaces, storage is managed automatically. The database manager creates and extends containers as needed.

Ability to isolate data in buffer pools for improved performance or memory utilization
If you have a set of objects (for example, tables, indexes) that are queried frequently, you can assign the table space in which they reside a buffer pool with a single CREATE or ALTER TABLESPACE statement. You can assign temporary table spaces to their own buffer pool to increase the performance of activities such as sorts or joins. In some cases, it might make sense to define smaller buffer pools for seldom-accessed data, or for applications that require very random access into a very large table; in such cases, data need not be kept in the buffer pool for longer than a single query

Table spaces consist of one or more *containers*. A container can be a directory name, a device name, or a file name. A single table space can have several containers. It is possible for multiple containers (from one or more table spaces) to be created on the same physical storage device (although you will get the best performance if each container you create uses a different storage device). If you are using automatic storage table spaces, the creation and management of containers is

handled automatically by the database manager. If you are not using automatic storage table spaces, you must define and manage containers yourself.

Figure 6 illustrates the relationship between tables and table spaces within a database, and the containers associated with that database.
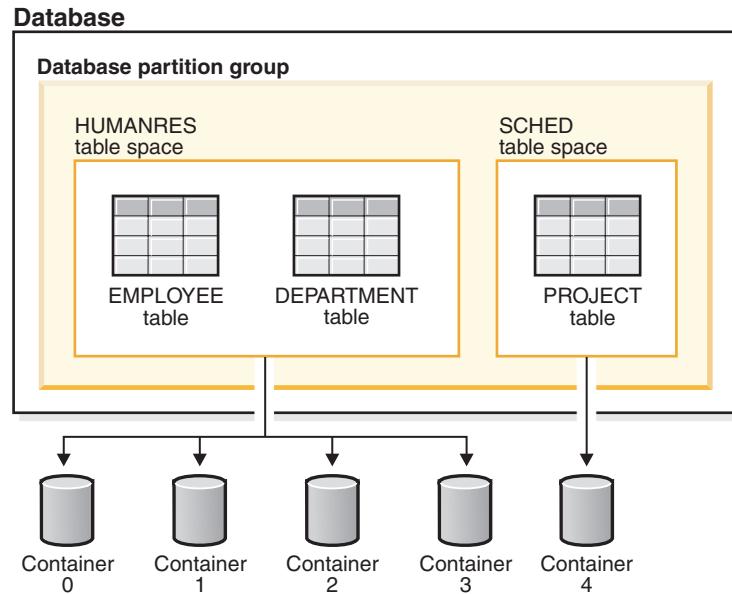
Figure 6. Table spaces and tables in a database

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 7 on page 29 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.
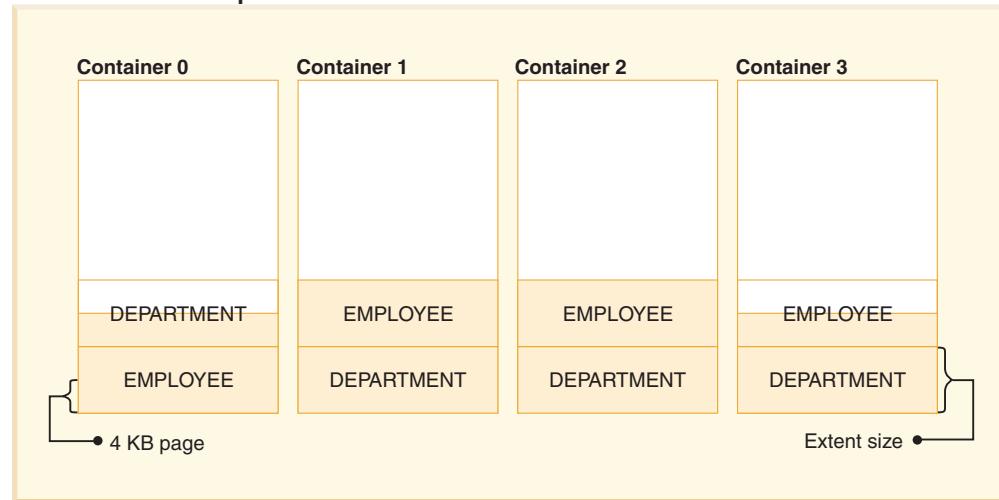
HUMANRES table space

Figure 7. Containers and extents in a table space

Character conversion

A *string* is a sequence of bytes that might represent characters. All the characters within a string have a common coding representation. In some cases, it might be necessary to convert these characters to a different coding representation, a process known as *character conversion*.

When character conversion is required, it is automatic. Applications do not need to explicitly invoke character conversion, because the DB2 database server and client perform all necessary character conversion automatically.

Character conversion can occur when an SQL statement is executed remotely. Consider, for example, the following scenarios in which the coding representations might 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.

Character conversion

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 multibyte characters. *Bit data* (columns defined as FOR BIT DATA, or BLOBs, or binary strings) is not associated with any character set.

code page: pp1 (ASCII)

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

code point: 2F
character set ss1 (in code page pp1)

code page: pp2 (EBCDIC)

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

character set ss1 (in code page pp2)

Figure 8. 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.

Application code page

The code page under which the application runs. This code page 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 value 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.

Multicultural 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 multibyte 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 blank padding in graphic strings and in other places.
- 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 4. 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'FDFE'
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'
1392	X'A3A5'	X'A3DF'	-	-

Code Page 5488 is equivalent to 1392 for the EXPORT, IMPORT, and LOAD utilities.

For Unicode databases, the GRAPHIC space is X'0020', which is different from the GRAPHIC space of X'3000' used for eucJP (Extended UNIX Code - Japan) and eucTW (Extended UNIX Code - 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.

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.

Remote unit of work for distributed relational databases

The *remote unit of work facility* provides for the remote preparation and execution of SQL statements.

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

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

Remote unit of work for distributed relational databases

At any given time, an application process is in one of four possible *connection states*:

- Connectable and connected

  An application process is connected to an application server, and CONNECT statements can be executed.

  If implicit connect is available:
  - The application process enters this state when a CONNECT TO statement or a CONNECT without operands statement is successfully executed from the connectable and unconnected state.
  - The application process might enter this state from the implicitly connectable state if any SQL statement other than CONNECT RESET, DISCONNECT, SET CONNECTION, or RELEASE is issued.

  Whether or not implicit connect is available, this state is entered when:
  - A CONNECT TO statement is successfully executed from the connectable and unconnected state.
  - A COMMIT or ROLLBACK statement is successfully issued, or a forced rollback occurs from the unconnectable and connected state.

- Unconnectable and connected

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

- Connectable and unconnected

  An application process is not connected to an application server. CONNECT TO is the only SQL statement that can be executed; otherwise, an error (SQLSTATE 08003) is raised.

  Whether or not implicit connect is available, the application process enters this state if an error occurs when a CONNECT TO statement is issued, or an error occurs within a unit of work, causing the loss of a connection and a rollback. An error that occurs because the application process is not in the connectable state, or because the server name is not listed in the local directory, does not cause a transition to this state.

  If implicit connect is not available:
  - The application process is initially in this state
  - The CONNECT RESET and DISCONNECT statements cause a transition to this state.

- Implicitly connectable (if implicit connect is available).

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

Availability of implicit connect is determined by installation options, environment variables, and authentication settings.

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

It is also an error to execute any SQL statement other than CONNECT TO, CONNECT RESET, CONNECT with no operand, SET CONNECTION, RELEASE, COMMIT, or ROLLBACK, and then to execute a CONNECT TO statement. To avoid this error, a CONNECT RESET, DISCONNECT (preceded by a COMMIT or ROLLBACK statement), COMMIT, or ROLLBACK statement should be executed before the CONNECT TO statement.
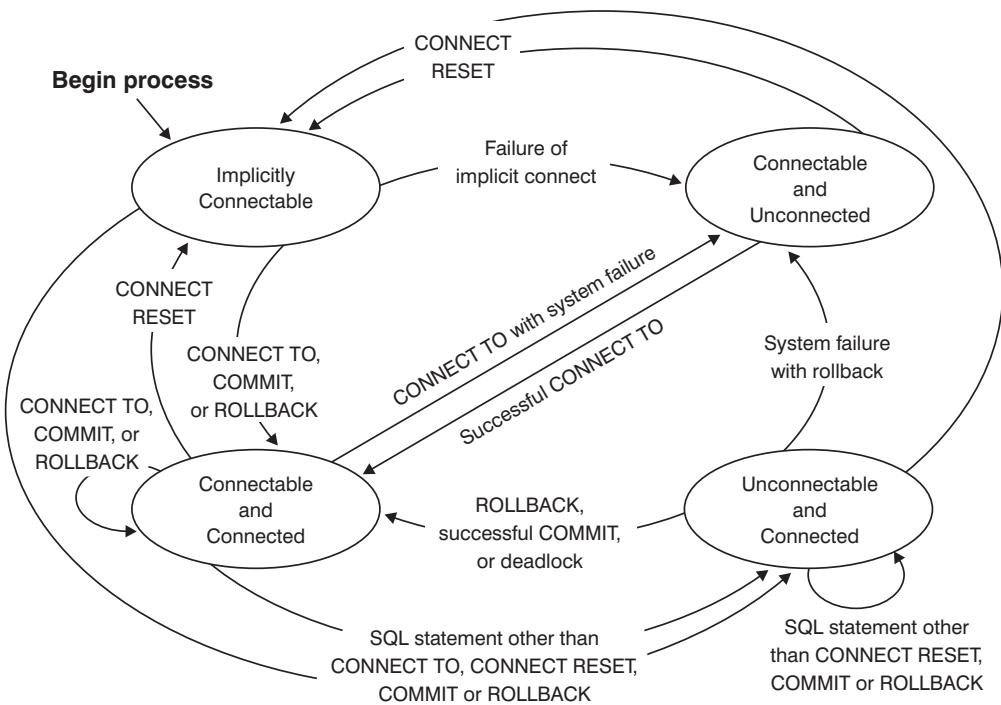
Figure 9. Connection State Transitions If Implicit Connect Is Available

Figure 10. Connection State Transitions If Implicit Connect Is Not Available

Application-directed distributed unit of work

The *application-directed distributed unit of work facility* provides for the remote preparation and execution of SQL statements.

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

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

A type 2 application process:
- Is always connectable
- Is either in the connected state or in the unconnected state
- Has zero or more connections.

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

An individual connection always has one of the following connection states:
- current and held
- current and release-pending

- dormant and held
- dormant and release-pending

A type 2 application process is initially in the unconnected state, and does not have any connections. A connection is initially in the current and held state.
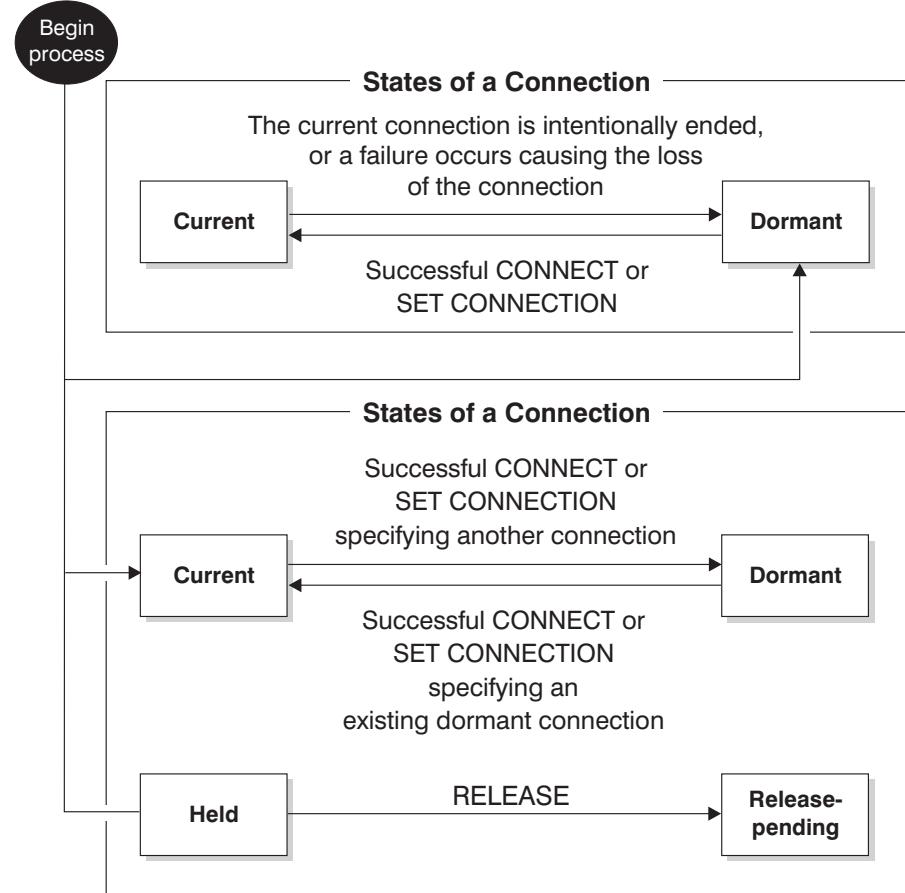
Figure 11. Application-Directed Distributed Unit of Work Connection State Transitions

Application process connection states

There are certain rules that apply to the execution of a CONNECT statement.

The following rules apply to the execution of a CONNECT statement:
- A context cannot have more than one connection to the same application server at the same time.
- When an application process executes a SET CONNECTION statement, the specified location name must be an existing connection in the set of connections for the application process.
- When an application process executes a CONNECT statement, and the SQLRULES(STD) option is in effect, the specified server name must *not* be an existing connection in the set of connections for the application process. For a description of the SQLRULES option, see "Options that govern unit of work semantics" on page 40.

If an application process has a current connection, the application process is in the *connected* state. The CURRENT SERVER special register contains the name of

the application server for the current connection. The application process can execute SQL statements that refer to objects managed by that application server.

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

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

An application process in the *connected state* enters the *unconnected state* when its current connection intentionally ends, or when an SQL statement fails, causing a rollback operation at the application server and loss of the connection. Connections end intentionally following the successful execution of a DISCONNECT statement, or a COMMIT statement when the connection is in release-pending state. (If the DISCONNECT precompiler option is set to AUTOMATIC, all connections end. If it is set to CONDITIONAL, all connections that do not have open WITH HOLD cursors end.)

Connection states

There are two types of connection states: "held and release-pending states" and "current and dormant states".

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

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

Held and release-pending states. The RELEASE statement controls whether a connection is in the held or the release-pending state. The *release-pending* state means that a disconnect is to occur at the next successful commit operation. (A rollback has no effect on connections.) The *held* state means that a disconnect is *not* to occur at the next commit operation.

All connections are initially in the held state and can be moved to the release-pending state using the RELEASE statement. Once in the release-pending state, a connection cannot be moved back to the held state. A connection remains in release-pending state across unit of work boundaries if a ROLLBACK statement is issued, or if an unsuccessful commit operation results in a rollback operation.

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

Current and dormant states. Regardless of whether a connection is in the held state or the release-pending state, it can also be in the current state or the dormant state. A connection in the *current* state is the connection being used to execute SQL statements while in this state. A connection in the *dormant* state is a connection that is not current.

The only SQL statements that can flow on a dormant connection are COMMIT, ROLLBACK, DISCONNECT, or RELEASE. The SET CONNECTION and CONNECT statements change the connection state of the specified server to current, and any existing connections are placed or remain in dormant state. At any point in time, only one connection can be in current state. If a dormant connection becomes current in the same unit of work, the state of all locks, cursors, and prepared statements is the same as the state they were in the last time that the connection was current.

When a connection ends

When a connection ends, all resources that were acquired by the application process through the connection, and all resources that were used to create and maintain the connection are de-allocated. For example, if the application process executes a RELEASE statement, any open cursors are closed when the connection ends during the next commit operation.

A connection can also end because of a communications failure. If this connection is in current state, the application process is placed in unconnected state.

All connections for an application process end when the process ends.

Options that govern unit of work semantics

The semantics of type 2 connection management are determined by a set of precompiler options. These options are summarized in the following list, with default values indicated by bold and underlined text.

- CONNECT (**1** | 2). Specifies whether CONNECT statements are to be processed as type 1 or type 2.
- SQLRULES (**DB2** | STD). Specifies whether type 2 CONNECTs are to be processed according to the DB2 rules, which allow CONNECT to switch to a dormant connection, or the SQL92 Standard rules, which do not allow this.
- DISCONNECT (**EXPLICIT** | CONDITIONAL | AUTOMATIC). Specifies what database connections are to be disconnected when a commit operation occurs:
  - Those that have been explicitly marked for release by the SQL RELEASE statement (EXPLICIT)
  - Those that have no open WITH HOLD cursors, and those that are marked for release (CONDITIONAL)
  - All connections (AUTOMATIC).
- SYNCPOINT (**ONEPHASE** | TWOPHASE | NONE). Specifies how COMMITs or ROLLBACKs are to be coordinated among multiple database connections. This option is ignored, and is included for backwards compatibility only.
  - Updates can only occur against one database in the unit of work, and all other databases are read-only (ONEPHASE). Any update attempts to other databases raise an error (SQLSTATE 25000).
  - A transaction manager (TM) is used at run time to coordinate two-phase COMMITs among those databases that support this protocol (TWOPHASE).
  - Does not use a TM to perform two-phase COMMITs, and does not enforce single updater, multiple reader (NONE). When a COMMIT or a ROLLBACK statement is executed, individual COMMITs or ROLLBACKs are posted to all databases. If one or more ROLLBACKs fail, an error (SQLSTATE 58005) is raised. If one or more COMMITs fail, another error (SQLSTATE 40003) is raised.

To override any of the previously listed options at run time, use the **SET CLIENT** command or the sqlesetc application programming interface (API). Their current settings can be obtained using the **QUERY CLIENT** command or the sqleqryc API. Note that these are not SQL statements; they are APIs defined in the various host languages and in the command line processor (CLP).

Data representation considerations

Different systems represent data in different ways. When data is moved from one system to another, data conversion must sometimes be performed.

Products supporting DRDA automatically perform any necessary conversions at the receiving system.

To perform conversions of numeric data, the system needs to know the data type and how it is represented by the sending system. Additional information is needed to convert character strings. String conversion depends on both the code page of the data and the operation that is to be performed on that data. Character conversions are performed in accordance with the IBM Character Data Representation Architecture (CDRA). For more information about character conversion, see the *Character Data Representation Architecture: Reference & Registry* (SC09-2190-00) manual.

Event monitors that write to tables, files, and pipes

Some event monitors can be configured to write information about database events to tables, pipes, or files.

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.

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.

Note: The deprecated detailed deadlock event monitor, DB2DETAILDEADLOCK, is created by default for each database and starts when the database is activated. Avoid the overhead this event monitor incurs by dropping it. The use of the DB2DETAILDEADLOCK monitor element is no longer recommended. This

deprecated event monitor might be removed in a future release. Use the CREATE EVENT MONITOR FOR LOCKING statement to monitor lock-related events, such as lock timeouts, lock waits, and deadlocks.

Database partitioning across multiple database partitions

The database manager allows great flexibility in spreading data across multiple database partitions 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 is to 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 can 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.

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 might 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. 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 AMS), 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,
```

Large object behavior in partitioned tables

```
        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.
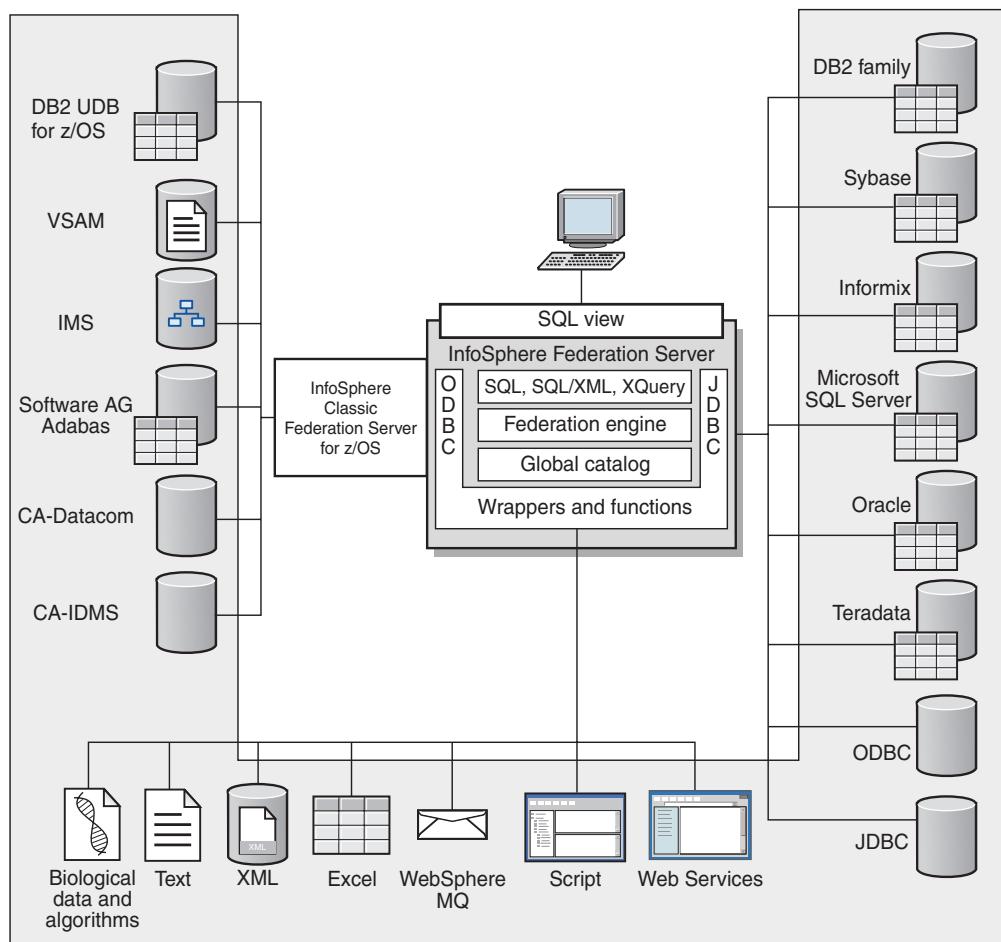
Figure 12. The components of a federated system

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 an XML tagged file).

Through some data sources you can access other data sources. For example, with the ODBC wrapper you can access IBM InfoSphere Classic Federation Server for z/OS data sources such as DB2 for z/OS, IMS™, CA-IDMS, CA-Datacom, 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 the DB2 database system. 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 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 IBM InfoSphere 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 specify the following information in the CREATE USER MAPPING statement:
- local authorization ID
- local name of the remote data source server as specified in the server definition
- 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.

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

In InfoSphere Federation Server Version 9.7 Fix Pack 2 and later, you can also create public user mappings to allow all local database users to access a data source through a single remote user ID and password.

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 a *data 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 complier 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.

IBM InfoSphere 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 IBM InfoSphere 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.

Table 5. 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, 5.3	None	Y	Y
DB2 for Linux, UNIX, and Windows	8.1.x, 8.2.x, 9.1, 9.5, 9.7, 9.8	None	Y	Y
DB2 for z/OS	7.x, 8.x, 9.x	DB2 Connect™ V9.7	Y	Y
DB2 for System i	5.2, 5.3, 5.4, 6.1	DB2 Connect V9.7	Y	Y
DB2 Server for VSE and VM	7.2 , 7.4	DB2 Connect V9.7	Y	Y
Flat files		None	Y	Y
Informix	Informix XPS 8.50, 8.51 and Informix IDS IDS 7.31, IDS 9.40, IDS 10.0, 11.1, 11.5, 11.7	Informix Client SDK version 2.81.TC2 or later; 3.0 required for SLES 10 on Power	Y	Y
JDBC	3.0 or later	JDBC drivers that comply with JDBC 3.0 or later	Y	Y
Microsoft Excel	2000, 2002, 2003, 2007	None		Y
Microsoft SQL Server	Microsoft SQL Server 2000/SP4, 2005, 2008, 2008R2	For Windows, Microsoft SQL Server Client ODBC 3.0 (or later) driver. For Unix, DataDirect ODBC 5.3, 6	Y	Y
MQ	MQ7	MQ7	Y	Y
ODBC	3.0	ODBC drivers that comply with ODBC 3.0, or later**	Y	Y
OLE DB	2.7, 2.8	OLE DB 2.0, or later	Y	Y
Oracle	10g, 10gR2, 11g, 11gR1, 11gR2	Oracle NET client 10.0 - 10.1, 10.2.0.1 with patch 3807408, 10.1.0.3 with patch 3807408, 11.1.0.6.0	Y	Y

Supported data sources

Table 5. Supported data sources, client software requirements, and support from 32-bit operating systems. (continued)

Data source	Supported versions	Client software	32-bit hardware architecture and operating system	
			X86-32	X86-32
			Linux, RedHat Enterprise Linux (RHEL), SUSE	Windows
Sybase	Sybase ASE 12.5, 15.0, 15.5	Sybase Open Client 12.5 - 15.5	Y	Y
Teradata	2.5, 2.6, 12, 13	Shared Common Components for Internationalization for Teradata (tdicu) version 1.01 or later, Teradata Generic Security Services (TeraGSS) version 6.01 or later, and the software on the following operating systems: For Windows, Teradata API library Call-Level Interface, Version 2, Release 4.8.0 or later For UNIX and Linux, Teradata Call-Level Interface, Version 2, Release 4.8.0 or later	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 6.20.UC5 and 6.3, and InfoSphere Classic Federation Server for z/OS V8.2 and above , among other data sources.

Table 6. 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
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
Informix	Y		Y	Y	Y	Y	Y
JDBC	Y	Y	Y	Y	Y	Y	Y

Table 6. 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							
Microsoft Excel							
Microsoft SQL Server	Y	Y	Y	Y		Y	Y
MQ	Y	N	Y	Y	Y	Y	Y
ODBC	Y	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		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 6.20.UC5 and 6.3, IBM InfoSphere Classic Federation Server for z/OS Version 8.2 and above using 32-bit and 64-bit clients, and Netezza.

For additional information about supported data sources, see the Data source requirements page on the Web.

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:
- 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 the DB2 database system 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.

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

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

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

Table 7. Special characters

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

Identifiers

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

* SQL identifiers

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

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

    Examples

        WKLYSAL   WKLY_SAL

  – A *delimited identifier* is a sequence of one or more characters enclosed by double quotation marks. Leading blanks in the sequence are significant. Trailing blanks in the sequence are not significant, although they are stored with the identifier. Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. 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 a database object name depend on the type of the object designated by the name. A name may consist of a single SQL identifier or it may be qualified with one or more identifiers that more specifically identify the object. A period must separate each identifier.

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

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

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

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

schema-name) followed by a period and an SQL identifier. If the array type is defined in a module and used outside of the same module, it must be qualified by the module-name.

authorization-name

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

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

bufferpool-name

An identifier that designates a buffer pool.

column-name

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

component-name

An identifier that designates a security label component.

condition-name

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

constraint-name

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

correlation-name

An identifier that designates a result table.

cursor-name

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

cursor-type-name

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

cursor-variable-name

A qualified or unqualified name that designates a global variable, local

variable or an SQL parameter of a cursor type. An unqualified cursor variable name in an SQL statement is implicitly qualified, depending on context.

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

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

descriptor-name
A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). For the description of a host identifier, see "References to host variables" on page 77. Note that a descriptor name never includes an indicator variable.

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

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

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

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

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

group-name

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

host-variable

A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, explained in "References to host variables" on page 77.

index-name

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

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

method-name

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

module-name

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

nickname

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

package-name

A 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.

partition-name

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

period-name

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

procedure-name

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

remote-authorization-name

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

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

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

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

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

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

role-name
> An identifier that designates a role.

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

savepoint-name
> An identifier that designates a savepoint.

schema-name
> An identifier that provides a logical grouping for SQL objects. A schema name used as a qualifier for the name of an object may be implicitly determined:
> - from the value of the CURRENT SCHEMA special register
> - from the value of the QUALIFIER precompile/bind option
> - on the basis of a resolution algorithm that uses the CURRENT PATH special register
> - on the basis of the schema name for another object in the same SQL statement.
>
> To avoid complications, it is recommended that the name SESSION not be used as a schema, except as the schema for declared global temporary tables (which *must* use the schema name SESSION).

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

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

sequence-name
> An identifier that designates a sequence.

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

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

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

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

storagegroup-name
> An identifier that designates a storage group.

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

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

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

tablespace-name
> An identifier that designates a table space.

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

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

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

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

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

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

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

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

wrapper-name
>   An identifier that designates a wrapper.

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

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

Aliases for database objects

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

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

Aliases are also known as synonyms.

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

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

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

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

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

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

effectively becomes

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

Authorization IDs and authorization names

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

Authorization IDs are used by the database manager to provide:
- Authorization checking of SQL statements
- A default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. The authorization ID is also included in the default CURRENT PATH special register and the FUNCPATH precompile/bind option.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID that is used during program binding. The authorization ID that applies to a dynamic SQL statement is based on the DYNAMICRULES option supplied at bind time, and on the current runtime environment for the package issuing the dynamic SQL statement:
- In a package that has bind behavior, the authorization ID used is the authorization ID of the package owner.
- In a package that has define behavior, the authorization ID used is the authorization ID of the corresponding routine's definer.
- In a package that has run behavior, the authorization ID used is the current authorization ID of the user executing the package.
- In a package that has invoke behavior, the authorization ID used is the authorization ID currently in effect when the routine is invoked. This is called the runtime authorization ID.

For more information, see "Dynamic SQL characteristics at run time" on page 68.

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 60.

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

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

Table 8. How DYNAMICRULES and the runtime environment determine dynamic SQL statement behavior (continued)

DYNAMICRULES value	Behavior of dynamic SQL statements in a stand-alone program environment	Behavior of dynamic SQL statements in a routine environment
INVOKEBIND	Bind behavior	Invoke behavior
INVOKERUN	Run behavior	Invoke behavior

Run behavior

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

Bind behavior

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

Define behavior

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

Invoke behavior

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

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

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

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

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

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

Authorization IDs and statement preparation

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

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

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

Column names

The meaning of a *column name* depends on its context. A column name can be used to:
- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:

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

Qualified column names

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

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

Where a qualifier is optional, it can serve two purposes. They are described under "Column name qualifiers to avoid ambiguity" on page 73 and "Column name qualifiers in correlated references" on page 75.

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.

Identifiers

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

Example

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

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

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

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

```
FROM EMPLOYEE E, DEPARTMENT
```

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

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

1. Given the FROM clause:

   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```

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

2. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```

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

3. Given the FROM clause:

   ```
   FROM EMPLOYEE, EMPLOYEE
   ```

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

4. Given the following statement:

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

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

5. Given the FROM clause:

    FROM EMPLOYEE, X.EMPLOYEE

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

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

Given the FROM clause:

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

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

Column name qualifiers to avoid ambiguity

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

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

Table designators

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

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

Table designators in the FROM clause are established as follows:

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

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

For example, assume that the current schema is CORPDATA.

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

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

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

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

Now assume that the current schema is REGION.

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

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

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

Avoiding undefined or ambiguous references

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

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

operand of a right outer join or a full outer join and the column name does not refer to a column of a *table-reference* within the nested table expression's fullselect. The reference is undefined.

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

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

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

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

   is a valid statement.

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

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

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

Column name qualifiers in correlated references

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

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

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

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

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

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

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

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

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

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

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

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

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

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

References to variables

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

host variable

Host variables are defined by statements of a host language. For more information about how to refer to host variables, see "References to host variables."

transition variable

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

SQL variable

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

global variable

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

module variable

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

SQL parameter

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

parameter marker

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

References to host variables

A *host variable* is either:
• A variable in a host language such as a C variable, a C++ variable, a COBOL data item, a FORTRAN variable, or a Java variable

or:
- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions
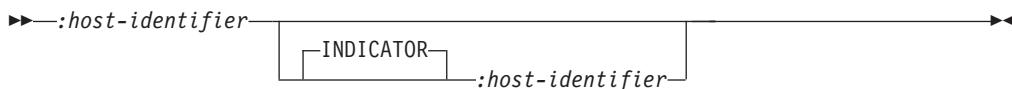
that is referenced in an SQL statement. Host variables are either directly defined by statements in the host language or are indirectly defined using SQL extensions.

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

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

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

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

```
►►──:host-identifier─────────────────────────────────────►◄
                     ┌─INDICATOR─┐
                     └───────────┴─:host-identifier─┘
```

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

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

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

The normal indicator variable has the following purposes:
- Specify a non-null value. A 0 (zero), or positive value of the indicator variable specifies that the associated, first, *host-identifier* provides the value of this host variable reference.
- Specify the null value. A negative value of the indicator variable specifies the null value.
- On output, indicate that a numeric conversion error (such as division by 0 or overflow) has occurred, if the **dft_sqlmathwarn** database configuration parameter is set to "yes" (or was set to "yes" during binding of a static SQL statement). A -2 value of the indicator variable indicates a null result because of either numeric truncation or friendly arithmetic warnings.

- On output, report the original length of a truncated string (if the source of the value is not a large object type).
- On output, report the seconds portion of a time if the time is truncated on assignment to a host variable.

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

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

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

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

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

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

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

the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

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

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

Example

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

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
  INTO :PNAME, :STAFF :STAFF_IND, :MAJPROJ :MAJPROJ_IND
  FROM PROJECT
  WHERE PROJNO = 'IF1000'
```

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

Variables in dynamic SQL

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

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

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

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

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

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

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

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

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

References to LOB variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see "References to LOB locator variables"), and LOB file reference variables (see "References to LOB file reference variables " on page 82) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

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

References to LOB locator variables

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

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

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

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

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

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

References to LOB file reference variables

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

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

A file reference variable has the following properties:

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

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

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

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

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

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

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

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

- Input (from client to server)

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

- Output (from server to client)

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

SQL_FILE_OVERWRITE (Overwrite)

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

SQL_FILE_APPEND

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

Data Length

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

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

Example of an output file reference variable (in C)

Given a declare section coded as:

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

Following preprocessing this would be:

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

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

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

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

Example of an input file reference variable (in C)

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

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

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

References to structured type host variables

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

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

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

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

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

Example

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

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

SQL path

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

For example, if the SQL path is SYSIBM. SYSFUN, SYSPROC, SYSIBMADM, SMITH, XGRAPHICS2 and an unqualified distinct type name MYTYPE was

specified, the database manager looks for MYTYPE first in schema SYSIBM, then SYSFUN, then SYSPROC, then SYSIBMADM, then SMITH, and then XGRAPHICS2.

The SQL path used depends on the SQL statement:

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

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

Qualification of unqualified object names

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

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

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

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

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

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

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

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

    – For global variables, the database manager searches the SQL path and selects the first schema in the SQL path such that the global variable exists in the schema.

    – For procedure names, the database manager uses the SQL path in conjunction with procedure resolution.

    – For function names, the database manager uses the SQL path in conjunction with function resolution .

    – For specific names specified for sourced functions, see "CREATE FUNCTION (Sourced)".

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

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

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

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

To run an unqualified user-defined routine instead of a new SYSIBM function with the same name, rename the user-defined routine or fully qualify the name before you run it. Alternatively, place in the SQL path the schema in which the user-defined routine exists before the schema in which the built-in functions and SQL administrative routines exist. However, promoting the schema in the SQL path increases the resolution time for all built-in functions and SQL administrative routines because the system schemas are considered first.

Resolving qualified object names

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

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

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

- If the context of the reference is within a module and the qualifier matches the module name, the database manager searches the module for the object, applying the appropriate resolution for the type of object to find a match among published and unpublished module objects. If no match is found, the search continues as specified in the next bullets.

- Assume that the qualifier is a schema name and, if the schema exists, resolve the object in the schema.

- If the qualifier is not an existing schema or the object is not found in the schema that matches the qualifier and the qualifier did not match the context module name, search for the first module that matches the qualifier in the schemas on the SQL path. If authorized to the matching module, resolve to the object in that module, considering only published module objects.

- If the qualifier is not found as a module on the SQL path and the qualifier did not match the context module name, check for a module public synonym that matches the qualifier. If found, resolve the object in the module identified by the module public synonym, considering only published module objects.

Data types

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

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

A number of built-in data types are supported. Figure 13 on page 89 shows the supported built-in data types. Support for the following user-defined data types is also provided:
- Array
- Cursor
- Distinct
- Row
- Structured
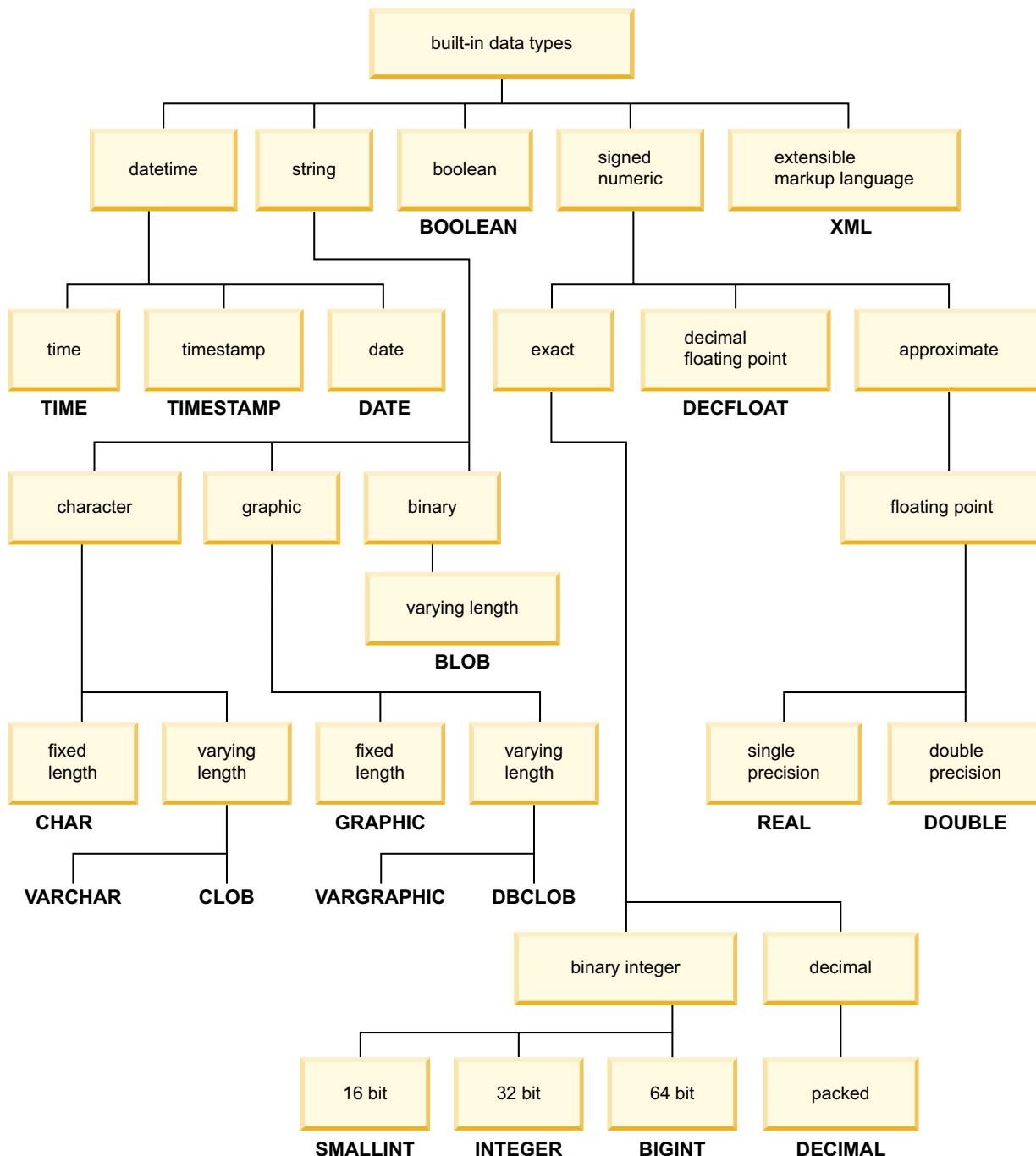
Figure 13. Built-in Data Types

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.

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

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 warning
- Signalling NaN - a value that represents undefined results and that causes an invalid number warning if used in any numeric operation

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

Subnormal numbers and underflow

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

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

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

Numbers

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

Character strings

A *character string* is a sequence of code units. The length of the string is the number of code units in the sequence. If the length is zero, the value is called the *empty string*. 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 1 - 254, inclusive, unless the string unit is CODEUNITS32 which has a range of 1 - 63, inclusive.

Varying-length character strings

There are two types of varying-length character strings:
- A VARCHAR value can be up to 32 672 bytes long. If the string unit is CODEUNITS32, the length can be up to 8 168 string units.
- A CLOB (character large object) value can be up to 2 gigabytes minus 1 byte (2 147 483 647 bytes) long. If the string unit is CODEUNITS32, the length can be up to 536 870 911 string units. A CLOB is used to store large SBCS or mixed (SBCS and MBCS) character-based data (such as documents written with a single character set) and, therefore, has an SBCS or mixed code page that is associated with it.

Special restrictions apply to expressions that result in a CLOB data type, and to structured type columns; such expressions and columns are not permitted in:
- A SELECT list that is preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, or IN predicate
- An aggregate function
- VARGRAPHIC, TRANSLATE, and datetime scalar functions
- The pattern operand in a LIKE predicate, or the search string operand in a POSSTR function
- The string representation of a datetime value.

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

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

Each character string is further defined as one of:

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

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

Character strings

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

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

String units specification for character strings

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

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

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

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

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

String units in built-in functions

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

CODEUNITS16

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

CODEUNITS32

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

OCTETS

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

The calculated length of a string computed using OCTETS (bytes) might differ from that computed using CODEUNITS16 or CODEUNITS32. When you are using OCTETS, the length of the string is determined by simply counting the number of bytes in the string. When you are using CODEUNITS16 or CODEUNITS32, the length of the string is determined by counting the number of 16-bit or 32-bit code units necessary to represent the string in UTF-16 or UTF-32. A length that is determined using CODEUNITS16 and CODEUNITS32 is identical unless the data contains supplementary characters (see "Difference between CODEUNITS16 and CODEUNITS32" on page 96).

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

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

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

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

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

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

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

```
Format    Representation of the name 'Jürgen'
--------  --------------------------------------
UTF-8   X'4AC3BC7267656E'
UTF-16BE  X'004A00FC007200670065006E'
UTF-32BE  X'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, the string units, or the code page of the result of the function. If necessary, the data is converted to Unicode for evaluation when CODEUNITS16 or CODEUNITS32 is specified.

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

Difference between CODEUNITS16 and CODEUNITS32

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

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

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

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

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

SELECT CHARACTER_LENGTH(C1,CODEUNITS32) FROM T1    1

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

Graphic strings

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

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

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

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

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

Fixed-length graphic strings (GRAPHIC)

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

Varying-length graphic strings

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

Special restrictions apply to an expression that results in a DBCLOB data type. These restrictions are the same as the restrictions specified in "Varying-length character strings" on page 93.

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

String units specification for graphic strings

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

Double bytes
Indicates that the units for the length attribute are double bytes. This unit

of length applies to all graphic string data types in a non-Unicode database. In a Unicode database, CODEUNITS16 is used.

CODEUNITS16

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

CODEUNITS32

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

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

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

National character strings

A national character string is a sequence of bytes that represents character data in UTF-8 or UTF16BE encoding in a Unicode database.

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

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

nchar_mapping is CHAR_CU32 in a Unicode database:

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

nchar_mapping is GRAPHIC_CU32 in a Unicode database:

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

nchar_mapping is GRAPHIC_CU16 in a Unicode database:

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

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

Binary strings

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

Character strings of the FOR BIT DATA subtype may be used for similar purposes. Binary strings are not associated with a code page. Binary strings have the same restrictions as character strings (for details, see "Varying-length character strings" on page 93). Only character strings of the FOR BIT DATA subtype are compatible with binary strings.

Binary large object (BLOB)

A *binary large object* is a varying-length binary string that can be up to 2 gigabytes minus 1 byte (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. In a Unicode database, NCLOB can be used as a synonym for DBCLOB.

LOB values are subject to restrictions, as described in "Varying-length character strings" on page 93. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

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

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

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

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

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

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

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

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

Datetime values

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

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

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

String representations of datetime values

Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the user. Date, time, and timestamp values can, however, also be represented by strings. This is useful because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Before it can be retrieved, a datetime value must be assigned to a string variable. The CHAR function or the GRAPHIC function (for Unicode databases only) can be used to change a datetime value to a string representation. The string

representation is normally the default format of datetime values associated with the territory code of the application, unless overridden by specification of the DATETIME option when the program is precompiled or bound to the database.

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

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

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

Date strings

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 9. 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.

Table 10. 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

Datetime values

Table 10. Formats for String Representations of Times (continued)

Format Name	Abbreviation	Time Format	Example
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* or *yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn*, where the number of digits for fractional seconds can range from 0 to 12. Trailing blanks may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp. Trailing zeros can be truncated or entirely omitted from the fractional seconds. If a string representation of a timestamp is implicitly cast to a value with a TIMESTAMP data type, the timestamp precision of the result of the cast is determined by the precision of the TIMESTAMP operand in an expression or the precision of the TIMESTAMP target in an assignment. Digits in the string beyond the timestamp precision of the cast are truncated or any missing digits to reach the timestamp precision of the cast are assumed to be zeros. For example, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000000000.

A string representation of a timestamp can be given a different timestamp precision by explicitly casting the value to a timestamp with a specified precision. If the string is a constant, an alternative is to precede the string constant with the TIMESTAMP keyword. For example, TIMESTAMP '2007-03-28 14:50:35.123' has the TIMESTAMP(3) data type.

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.nnnnnnnnnnnn*, where the number of digits for fractional seconds can range from 0 to 12..

Boolean values

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

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

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

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

Cursor values

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

The CURSOR type is a built-in data type that can only be used as the data type of:
- A local variable in a compound SQL (compiled) statement
- A parameter of an SQL routine
- The returns type of an SQL function
- A global variable

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

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

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

XML values

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

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

Special restrictions apply to expressions that result in an XML data type value; such expressions and columns are not permitted in (SQLSTATE 42818):
- A SELECT list preceded by the DISTINCT clause
- A GROUP BY clause
- An ORDER BY clause
- A subselect of a set operator other than UNION ALL
- A basic, quantified, BETWEEN, IN, or LIKE predicate
- An aggregate function with DISTINCT

Array values

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

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

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

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

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

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

Anchored types

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

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

User-defined types

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

There are six user-defined types:
- Distinct type
- Structured type
- Reference type
- Array type
- Row type
- Cursor type

Each of these types is described in the following sections.

Distinct type

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

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

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

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

Strongly typed distinct type

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

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

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

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

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

User-defined types

Weakly typed distinct type

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

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

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

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

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

Using distinct types can provide benefits in the following categories:

Extensibility

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

Flexibility

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

Consistent and inherited behavior

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

Encapsulation

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

Performance

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

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

Structured type

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

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

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

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

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:
- *Directly uses*: A type **A** is said to directly use another type **B**, if and only if one of the following statements is true:
  1. Type **A** has an attribute of type **B**.
  2. Type **B** is a subtype of **A** or a supertype of **A**.
- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following statements is true:
  1. Type **A** directly uses type **B**.
  2. Type **A** directly uses some type **C** and type **C** indirectly uses type **B**.

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

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

Reference type

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

User-defined types

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

Array type

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

Row type

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

Cursor type

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

Promotion of data types

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

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

Promotion of data types is used when:
- Performing function resolution
- Casting user-defined types
- Assigning user-defined types to built-in data types

Table 11 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 11. Data Type Precedence Table

Data Type	Data Type Precedence List (in best-to-worst order)
SMALLINT	SMALLINT, INTEGER, BIGINT, decimal, real, double, DECFLOAT
INTEGER	INTEGER, BIGINT, decimal, real, double, DECFLOAT
BIGINT	BIGINT, decimal, real, double, DECFLOAT
decimal	decimal, real, double, DECFLOAT
real	real, double, DECFLOAT
double	double, DECFLOAT
DECFLOAT	DECFLOAT
CHAR	CHAR, VARCHAR, CLOB
VARCHAR	VARCHAR, CLOB
CLOB	CLOB
GRAPHIC	GRAPHIC, VARGRAPHIC, DBCLOB
VARGRAPHIC	VARGRAPHIC, DBCLOB
DBCLOB	DBCLOB
BLOB	BLOB
DATE	DATE, TIMESTAMP
TIME	TIME
TIMESTAMP	TIMESTAMP
BOOLEAN	BOOLEAN
CURSOR	CURSOR
ARRAY	ARRAY
udt	udt (same name) or a supertype of udt
REF(T)	REF(S) (provided that S is a supertype of T)
ROW	ROW

Promotion of data types

Table 11. Data Type Precedence Table (continued)

Data Type	Data Type Precedence List (in best-to-worst order)

Note:

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

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

2. For a Unicode database, the following data types are considered to be equivalent:
   - CHAR and GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

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

Casting between data types

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

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

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

The supported casts between built-in data types are shown in Table 12 on page 119. The first column represents the data type of the cast operand (source data type), and the data types across the header row represent the target data type of the cast operation. A 'Y' indicates that the CAST specification can be used for the combination of source and target data types. Cases in which only the XMLCAST specification can be used are noted.

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

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

The following casts involving strongly typed distinct types are supported (using the CAST specification unless noted otherwise):
- Cast from distinct type *DT* to its source data type *S*
- Cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- Cast from distinct type *DT* to the same distinct type *DT*
- Cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT*
- Cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- Cast from a DOUBLE to distinct type *DT* with a source data type REAL
- Cast from a DECFLOAT to distinct type DT with a source data type of DECFLOAT
- Cast from a 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

Casting between data types

- Cast from an XML to a distinct type *DT* with a source data type of any built-in data type, using the XMLCAST specification depending on the XML schema data type of the XML value

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

FOR BIT DATA character types cannot be cast to CLOB.

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

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

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

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

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

The following casts involving reference types are supported:
- cast from reference type *RT* to its representation data type *S*
- cast from the representation data type *S* of reference type *RT* to reference type *RT*
- cast from reference type *RT* with target type *T* to a reference type *RS* with target type *S* where *S* is a supertype of *T*.
- cast from a data type *A* to reference type *RT*, where *A* is promotable to the representation data type *S* of reference type *RT*.

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

Table 12. Supported Casts between Built-in Data Types

	Target Data Type																				
Source Data Type	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	DECFLOAT	CHAR	CHAR FB D²	VARCHAR	VARCHAR FB D²	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BLOB	DATE	TIME	TIMESTAMP	XML	BOOLEAN
SMALLINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	Y⁷
INTEGER	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	Y⁷
BIGINT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	Y⁷
DECIMAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	-
REAL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	-
DOUBLE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	Y³	-
DECFLOAT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y¹	Y	Y	Y	Y	Y⁴	-
CHAR FOR BIT DATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y³	-
VARCHAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y¹	Y	Y	Y	Y	Y⁴	-
VARCHAR FOR BIT DATA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y	Y	Y³	-
CLOB	-	-	-	-	-	-	-	Y	-	Y	-	Y	Y¹	Y¹	Y¹	Y	-	-	-	Y⁴	-
GRAPHIC	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	-	Y¹	-	Y¹	Y	Y	Y	Y	Y¹	Y¹	Y¹	Y³	-
VARGRAPHIC	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	Y¹	-	Y¹	-	Y¹	Y	Y	Y	Y	Y¹	Y¹	Y¹	Y³	-
DBCLOB	-	-	-	-	-	-	-	Y¹	-	Y¹	-	Y¹	Y	Y	Y	Y	-	-	-	Y³	-
BLOB	-	-	-	-	-	-	-	-	Y	-	Y	-	-	-	-	Y	-	-	-	Y⁴	-
DATE	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y¹	Y¹	-	-	Y	-	Y	Y³	-
TIME	-	Y	Y	Y	-	-	-	Y	Y	Y	Y	-	Y¹	Y¹	-	-	-	Y	-	Y³	-
TIMESTAMP	-	-	Y	Y	-	-	-	Y	Y	Y	Y	-	Y¹	Y¹	-	-	Y	Y	Y	Y³	-
XML	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y⁵	Y	-
BOOLEAN	Y⁷	Y⁷	Y⁷	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y⁷

Casting between data types

Table 12. Supported Casts between Built-in Data Types (continued)

Source Data Type	Target Data Type																				
	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	DECFLOAT	CHAR	CHAR FOR BIT DATA²	VARCHAR	VARCHAR FOR BIT DATA²	CLOB	GRAPHIC	VARGRAPHIC	DBCLOB	BLOB	DATE	TIME	TIMESTAMP	XML	BOOLEAN

Notes

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

[1] Cast is only supported for Unicode databases.

[2] FOR BIT DATA

[3] Cast can only be performed using XMLCAST.

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

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

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

[7] Only supported using the CAST specification. No cast function exists.

Table 13 shows where to find information about the rules that apply when casting to the identified target data types.

Table 13. Rules for Casting to a Data Type

Target Data Type	Rules
SMALLINT	If the source type is BOOLEAN, then TRUE is cast to 1 and FALSE is cast to 0. For all other source types, see "SMALLINT" on page 623.
INTEGER	If the source type is BOOLEAN, then TRUE is cast to 1 and FALSE is cast to 0. For all other source types, see "INTEGER or INT" on page 502.
BIGINT	If the source type is BOOLEAN, then TRUE is cast to 1 and FALSE is cast to 0. For all other source types, see "BIGINT" on page 403.
DECIMAL	"DECIMAL or DEC" on page 447
NUMERIC	"DECIMAL or DEC" on page 447

Table 13. Rules for Casting to a Data Type (continued)

Target Data Type	Rules
REAL	"REAL" on page 583
DOUBLE	"DOUBLE_PRECISION or DOUBLE" on page 459
DECFLOAT	"DECFLOAT" on page 443
CHAR	"CHAR" on page 410
VARCHAR	"VARCHAR" on page 692
CLOB	"CLOB" on page 420
GRAPHIC	"GRAPHIC" on page 474
VARGRAPHIC	"VARGRAPHIC" on page 709
DBCLOB	"DBCLOB" on page 440
BLOB	"BLOB" on page 407
DATE	"DATE" on page 432
TIME	"TIME" on page 649
TIMESTAMP	If the source type is a character string, see "TIMESTAMP" on page 650, where one operand is specified. If the source data type is a DATE, the timestamp is composed of the specified date and a time of 00:00:00.
BOOLEAN	If the source type is numeric, then 0 is cast to FALSE and 1 is cast to TRUE. NULL is cast to NULL.

Casting non-XML values to XML values

Table 14. Supported Casts from Non-XML Values to XML Values

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
SMALLINT	Y	xs:short
INTEGER	Y	xs:int
BIGINT	Y	xs:long
DECIMAL or NUMERIC	Y	xs:decimal
REAL	Y	xs:float
DOUBLE	Y	xs:double
DECFLOAT	N	-
CHAR	Y	xs:string
VARCHAR	Y	xs:string
CLOB	Y	xs:string
GRAPHIC	Y	xs:string
VARGRAPHIC	Y	xs:string
DBCLOB	Y	xs:string
DATE	Y	xs:date
TIME	Y	xs:time
TIMESTAMP	Y	xs:dateTime[1]

Casting between data types

Table 14. Supported Casts from Non-XML Values to XML Values (continued)

Source Data Type	Target Data Type	
	XML	Resulting XML Schema Type
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

Notes

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

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

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

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

Casting XML values to non-XML values

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

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

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

Table 15. Supported Casts from XML Values to Non-XML Values

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

Table 15. Supported Casts from XML Values to Non-XML Values (continued)

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

Notes

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

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

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

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

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

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

Casting between data types

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

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

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

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

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

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

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

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

Assignments and comparisons

The basic operations of SQL are assignment and comparison.

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

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

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

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

Table 16. Data type compatibility for assignments and comparisons

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Binary String	Date	Time	Time-stamp	Boolean	UDT
Binary Integer	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	[2]
Decimal Number	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	[2]
Floating point	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	[2]
Decimal Floating-point	Yes	Yes	Yes	Yes	Yes	Yes [5]	No	No	No	No	No	[2]
Character String	Yes	Yes	Yes	Yes	Yes	Yes [5,6]	Yes [3]	Yes	Yes	Yes	No	[2]
Graphic String	Yes [5]	Yes [5]	Yes [5]	Yes [5]	Yes [5,6]	Yes	No	Yes [5]	Yes [5]	Yes [5]	No	[2]
Binary String	No	No	No	No	Yes [3]	No	Yes	No	No	No	No	[2]
Date	No	No	No	No	Yes	Yes [5]	No	Yes	No	Yes	No	[2]
Time	No	No	No	No	Yes	Yes [5]	No	No	Yes	[1]	No	[2]
Time-stamp	No	No	No	No	Yes	Yes [5]	No	Yes	[1]	Yes	No	[2]
Boolean	No	No	No	No	No	No	No	No	No	No	Yes [7]	[2]
UDT	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	Yes

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

[2] For detailed user-defined type information see "User-defined type assignments" on page 132 and "User-defined type comparisons" on page 141.

[3] Support for assignment only (not comparison) and only for character strings defined as FOR BIT DATA.

[4] For information about assignment and comparison of reference types, see "Reference type assignments" on page 136 and "Reference type comparisons" on page 142.

[5] Only supported for Unicode databases.

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

[7] Variables of Boolean data type cannot be directly compared; comparison only can be done with the literal values TRUE, FALSE, NULL.

Numeric assignments

For numeric assignments, overflow is not allowed.

- When assigning to an exact numeric data type, overflow occurs if any digit of the whole part of the number would be eliminated. If necessary, the fractional part of a number is truncated.

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

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

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

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

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

Assignments to integer

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

Assignments to decimal

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

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

When a floating-point number is assigned to a decimal column or variable, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number between 1 and -1 that is less than the smallest positive number or greater than the largest negative number that can be

represented in the decimal column or variable is reduced to 0. The scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

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

Assignments to floating-point

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

Assignments to decimal floating-point

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

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

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

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

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

Assignments from strings to numeric

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

String assignments

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

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

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

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

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

Retrieval assignment

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

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

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

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

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

Conversion rules for string assignments

A character string or graphic string assigned to a column, variable, or parameter is first converted, if necessary, to the code page of the target. Character conversion is necessary only if all of the following are true:
- The code pages are different.
- The string is neither null nor empty.
- Neither string has a code page value of 0 (FOR BIT DATA).

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

MBCS considerations for character string assignments

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

DBCS considerations for graphic string assignments

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

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

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

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

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

Assignments from numeric to strings

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

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

Datetime assignments

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

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

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

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

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

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

When the target is a string host variable, the following rules apply:
- **For a DATE:** If the length of the host variable is less than 10 characters, an error is returned.
- **For a TIME:** If the USA format is used, the length of the host variable must not be less than 8 characters; in other formats the length must not be less than 5 characters.
  If ISO or JIS formats are used, and if the length of the host variable is less than 8 characters, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.
- **For a TIMESTAMP:** If the length of the host variable is less than 19 characters, an error is returned. If the length is less than 32 characters, but greater than or equal to 19 characters, trailing digits of the fractional seconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

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

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

XML assignments

The general rule for XML assignments is that only an XML value can be assigned to XML columns or to XML variables. There are exceptions to this rule, as follows.
- *Processing of input XML host variables:* This is a special case of the XML assignment rule, because the host variable is based on a string value. To make the assignment to XML within SQL, the string value is implicitly parsed into an XML value using the setting of the CURRENT IMPLICIT XMLPARSE OPTION

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

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

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

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

User-defined type assignments

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

Strongly typed distinct type assignments

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

Assignments to host variables
> The assignment of a strongly typed distinct type value to a host variable

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

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

```
CREATE TYPE AGE AS SMALLINT
```

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

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

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

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

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

Assignments other than to host variables

A strongly typed distinct type can be either the source or target of an assignment. Assignment is based on whether the data type of the value to be assigned is castable to the data type of the target (see "Casting between data types" on page 117 for the casts supported when a distinct type is involved). A strongly typed distinct type value can be assigned to any target other than host a variable in the following cases:

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

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

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

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

```
CREATE TYPE AGE AS SMALLINT
```

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

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

Using the following SQL statement and substituting various values for X and Y to insert values into various columns of TABLE1 from TABLE2. The database manager uses assignment rules in this INSERT statement to determine if X can be assigned to Y. Table 17 on page 134 shows whether the assignments are valid.

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

Table 17. Assessment of various assignments

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

Weakly typed distinct type assignments

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

Structured type assignments

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

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

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

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

Array type assignments

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

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

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

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

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

is equivalent to:

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

And the statement:

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

is equivalent to:

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

The following are valid assignments that involve array type values:

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

Row type assignments

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

Cursor type assignments

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

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

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

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

Assignments and comparisons

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

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

Boolean type assignments

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

- TRUE
- FALSE
- NULL

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

Reference type assignments

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

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

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

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

Numeric comparisons

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

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

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

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

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

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

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

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

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

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

The comparison and ordering rules for special values are as follows:
- (+/-) INFINITY compares equal only to (+/-) INFINITY of the same sign.
- (+/-) NAN compares equal only to (+/-) NAN of the same sign.
- (+/-) SNAN compares equal only to (+/-) SNAN of the same sign.

The ordering among different special values is as follows:
- -NAN < -SNAN < -INFINITY < 0 < INFINITY < SNAN < NAN

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

String comparisons

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.

Assignments and comparisons

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

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

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

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

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

then the following is valid:

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

Examples:

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

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

```
'a' < 'A' < 'á' < 'Á'
```

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

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

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

```
 'A' < 'a' < 'Á' < 'á'
```

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

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

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

```
'A' < 'a' < 'Á' < 'á'
```

DBCS characters in CHAR columns sort a sequence of bytes, according to their weights (first pass) and then according to their code points to break the tie (second pass). First bytes have equal weights, so the code points (0xC1 and 0xE1) break the

tie. Therefore, characters D1 and D2 sort before characters D3 and D4. Then the second bytes are compared in similar way, and the final result is as follows:

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

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

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

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

Conversion rules for comparison

When two strings are compared, one of the strings is first converted, if necessary, to the encoding scheme 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 137. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

MBCS considerations for string comparisons

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

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

Example:

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

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

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

and

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

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

Assignments and comparisons

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

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

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

and

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

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

Datetime comparisons

A date, time, or timestamp value can be compared with another value of the same data type, a datetime constant of the same data type, or with a string representation of a value of that data type. A date value or a string representation of a date can also be compared with a TIMESTAMP, where the missing time information for the date value is assumed to be all zeros. All comparisons are chronological, which means the further a point in time is from January 1, 0001, the greater the value of that point in time. The time 24:00:00 is greater than the time 00:00:00.

Comparisons that involve time values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds are implied.

Comparisons that involve timestamp values are evaluated according to the following rules:

* When comparing timestamp values with different precisions, the higher precision is used for the comparison and any missing digits for fractional seconds are assumed to be zero.
* When comparing a timestamp value with a string representation of a timestamp, the string representation is first converted to TIMESTAMP(12).
* Timestamp comparisons are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

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

User-defined type comparisons

Information about comparisons involving user-defined types is in the sections that follow.

Strongly typed distinct type comparisons

Values with a strongly typed distinct type only can be compared with values of exactly the same strongly typed distinct type.

For example, given the following YOUTH distinct type and CAMP_DB_ROSTER table:

```
CREATE TYPE YOUTH AS INTEGER

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

The following comparison is valid:

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

The following comparison is not valid:

```
SELECT * FROM CAMP_DB_ROSTER
    WHERE AGE > ATTENDEE_NUMBER
```

However, AGE can be compared to ATTENDEE_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB_ROSTER
    WHERE INTEGER(AGE) > ATTENDEE_NUMBER

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

SELECT * FROM CAMP_DB_ROSTER
    WHERE AGE > YOUTH(ATTENDEE_NUMBER)

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

Weakly typed distinct type comparisons

Values with a weakly typed distinct type are compared according to the comparison rules for the source type of the distinct type. Data type constraints have no impact on the comparison. It is valid to compare different weakly typed distinct types if the underlying source types are comparable.

Structured type comparisons

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Array type comparisons

Comparisons of array type values are not supported. Elements of arrays can be compared based on the comparison rules for the data type of the elements.

Row type comparisons

A row variable cannot be compared to another row variable even if the row type name is the same. Individual fields within a row type can be compared to other values and the comparison rules for the data type of the field apply.

Cursor type comparisons

A cursor variable cannot be compared to another cursor variable even if the cursor type name is the same.

Boolean type comparisons

A Boolean value can be compared with a Boolean literal value. A value of TRUE is greater than a value of FALSE.

Reference type comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the SQL path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

XML comparisons in a non-Unicode database

When performed in a non-Unicode database, comparisons between XML data and character or graphic string values require a code page conversion of one of the two sets of data being compared. Character or graphic values used in an SQL or XQuery statement, either as a query predicate or as a host variable with a character or graphic string data type, are converted to the database code page before comparison. If any characters included in this data have code points that are not part of the database code page, substitution characters are added in their place, potentially causing unexpected results for the query.

For example, a client with a UTF-8 code page is used to connect to a database server created with the Greek encoding ISO8859-7. The expression $\Sigma_G\Sigma_M$ is sent as the predicate of an XQuery statement, where Σ_G represents the Greek sigma character in Unicode (U+03A3) and Σ_M represents the mathematical symbol sigma in Unicode (U+2211). This expression is first converted to the database code page, so that both "Σ" characters are converted to the equivalent code point for sigma in the Greek database code page, 0xD3. We may denote this code point as Σ_A. The newly converted expression $\Sigma_A\Sigma_A$ is then converted again to UTF-8 for comparison with the target XML data. Since the distinction between these two code points was lost as a result of the code page conversion required to pass the predicate expression into the database, the two initially distinct values Σ_G and Σ_M are passed to the XML parser as the expression $\Sigma_G\Sigma_G$. This expression then fails to match when compared to the value $\Sigma_G\Sigma_M$ in an XML document.

One way to avoid the unexpected query results that may be caused by code page conversion issues is to ensure that all characters used in a query expression have

matching code points in the database code page. Characters that do not have matching code points can be included through the use of a Unicode character entity reference. A character entity reference will always bypass code page conversion. For example, using the character entity reference ࢣ in place of the Σ_M character ensures that the correct Unicode code point is used for the comparison, regardless of the database code page.

Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This topic explains those rules.

These rules apply to:
- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression and the DECODE and NVL2 scalar functions
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause
- Expression values for the elements in an array constructor
- Arguments of a BETWEEN predicate (except if the data types of all operands are numeric)
- Arguments for the aggregation group ranges in OLAP specifications

These rules are applied subject to other restrictions on long strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types. The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 bytes. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

Character strings

A character string value is compatible with another character string value. Character strings include data types CHAR, VARCHAR, and CLOB.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
CHAR(x)	VARCHAR(y)	VARCHAR(z) where z = max(x,y)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = max(x,y)

If one operand is...	And the other operand is...	The data type of the result is...
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where z = max(x,y)

The code page of the result character string will be derived based on the rules for string conversions.

In a Unicode database, if either operand has string units CODEUNITS32 and the derived code page is not 0, the string units of the result character string is CODEUNITS32. If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA.Otherwise, the string units of the result character string is OCTETS. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x OCTETS) with x>63	CHAR(y CODEUNITS32)	VARCHAR(z CODEUNITS32) where z = max(x,y)
VARCHAR(x OCTETS) with x>8168	CHAR(y CODEUNITS32) or VARCHAR(y CODEUNITS32)	Error
CLOB(x OCTETS) with x>536870911	CHAR(y CODEUNITS32), VARCHAR(y CODEUNITS32), or CLOB(y CODEUNITS32)	CLOB(536870911 CODEUNITS32)

Graphic strings

A graphic string value is compatible with another graphic string value. Graphic strings include data types GRAPHIC, VARGRAPHIC, and DBCLOB.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max (x,y)

The code page of the result graphic string will be derived based on the rules for string conversions.

In a Unicode database, if either operand has string units CODEUNITS32 and the derived code page is not 0, the string units of the result character string is CODEUNITS32. If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA.Otherwise, the string units of the result character string is OCTETS. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS

with a length attribute that exceeds the data type maximum in CODEUNITS32. In a Unicode database, if either operand has string units CODEUNITS32, or CODEUNITS16, the string units of the result graphic string is CODEUNITS32. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is CODEUNITS16 with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x CODEUNITS16) with x>63	GRAPHIC(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)
VARGRAPHIC(x CODEUNITS16) with x>8168	GRAPHIC(y CODEUNITS32) or VARGRAPHIC(y CODEUNITS32)	Error
DBCLOB(x CODEUNITS16) with x>536870911	GRAPHIC(y CODEUNITS32), VARGRAPHIC(y CODEUNITS32) or DBCLOB(y CODEUNITS32)	DBCLOB(536870911 CODEUNITS32)

Character and graphic strings in a Unicode database

In a Unicode database, a character string value is compatible with a graphic string value.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	CHAR(y) or GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	CHAR(y) or VARCHAR(y)	VARGRAPHIC(z) where z = max(x,y)
VARCHAR(x)	GRAPHIC(y) or VARGRAPHIC	VARGRAPHIC(z) where z = max(x,y)
DBCLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	DBCLOB(z) where z = max(x,y)
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where z = max(x,y)

The string units of the result graphic string will be CODEUNITS32 if either operand has string units CODEUNITS32, or CODEUNITS16. Special cases apply when the string units of one operand is CODEUNITS32 and the string units of the other operand is OCTETS or CODEUNITS16 with a length attribute that exceeds the data type maximum in CODEUNITS32.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x OCTETS) with x>63	GRAPHIC(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x CODEUNITS16) with x>63	CHAR(y CODEUNITS32)	VARGRAPHIC(z CODEUNITS32) where z = max(x,y)
VARCHAR(x OCTETS) with x>8168	GRAPHIC(y CODEUNITS32) or VARGRAPHIC(y CODEUNITS32)	Error
VARGRAPHIC(x CODEUNITS16) with x>8168	CHAR(y CODEUNITS32) or VARCHAR(y CODEUNITS32)	Error
CLOB(x OCTETS) with x>536870911	GRAPHIC(y CODEUNITS32), VARGRAPHIC(y CODEUNITS32), or DBCLOB(y CODEUNITS32)	DBCLOB(536870911 CODEUNITS32)
DBCLOB(x CODEUNITS16) with x>536870911	CHAR(y CODEUNITS32), VARCHAR(y CODEUNITS32), or CLOB(y CODEUNITS32)	DBCLOB(536 870 911 CODEUNITS32)

Binary 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 data types, character-string data types (except CLOB), and in a Unicode database, graphic-string data types (except DBCLOB). Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE, and DECFLOAT.

Table 18. Operands and the resulting data type

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
SMALLINT	String	DECFLOAT(34)
INTEGER	SMALLINT or INTEGER	INTEGER
INTEGER	String	DECFLOAT(34)
BIGINT	SMALLINT, INTEGER, or BIGINT	BIGINT
BIGINT	String	DECFLOAT(34)
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where $p = x+max(w-x,5)$[1]
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where $p = x+max(w-x,11)$[1]

Rules for result data types

Table 18. Operands and the resulting data type (continued)

If one operand is...	And the other operand is...	The data type of the result is...
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where $p = x+max(w-x,19)$[1]
DECIMAL(w,x)	DECIMAL(y,z)	DECIMAL(p,s) where $p = max(x,z)+max(w-x,y-z)$[1] $s = max(x,z)$
DECIMAL(w,x)	String	DECFLOAT(34)
REAL	REAL	REAL
REAL	SMALLINT, INTEGER, BIGINT, or DECIMAL	DOUBLE
REAL	String	DECFLOAT(34)
DOUBLE	SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, or DOUBLE	DOUBLE
DOUBLE	String	DECFLOAT(34)
DECFLOAT(n)	SMALLINT, INTEGER, DECIMAL (<=16,s), REAL, or DOUBLE	DECFLOAT(n)
DECFLOAT(n)	BIGINT or DECIMAL (>16,s)	DECFLOAT(34)
DECFLOAT(n)	DECFLOAT(m)	DECFLOAT(MAX(n,m))
DECFLOAT(n)	String	DECFLOAT(34)

[1] Precision cannot exceed 31.

Datetime

Datetime data types are compatible with other operands of the same data type or any CHAR or VARCHAR expression that contains a valid string representation of the same data type. In addition, DATE is compatible with TIMESTAMP and the other operand of a TIMESTAMP can be the string representation of a timestamp or a date. In a Unicode database, character and graphic strings are compatible which implies that GRAPHIC or VARGRAPHIC string representations of datetime values are compatible with other datetime operands.

Table 19. Result data types with datetime operands

If one operand is...	And the other operand is...	The data type of the result is...
DATE	DATE, CHAR(y), or VARCHAR(y)	DATE
TIME	TIME, CHAR(y), or VARCHAR(y)	TIME
TIMESTAMP(x)	TIMESTAMP(y)	TIMESTAMP($max(x,y)$)
TIMESTAMP(x)	DATE, CHAR(y), or VARCHAR(y)	TIMESTAMP(x)

XML

An XML value is compatible with another XML value. The data type of the result is XML.

Boolean

A Boolean value is compatible with another Boolean value. The data type of the result is BOOLEAN.

User-defined types

Distinct types

A strongly typed distinct type value is compatible only with another value of the same distinct type. The data type of the result is the distinct type.

If both operands have the same weakly typed distinct type, the result is the distinct type. Otherwise, if any operand is a weakly typed distinct type then the data type of the operand is considered to be the source data type and the result data type is determined based on the combination of built-in data type operands.

Array data types

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

Cursor data types

A CURSOR value is compatible with another CURSOR value. The result data type is CURSOR. A user-defined cursor data type value is compatible only with another value of the same user-defined cursor data type. The data type of the result is the user-defined cursor data type.

Row data types

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

Reference types

A reference type value is compatible with another value of the same reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.

Structured types

A structured type value is compatible with another value of the same structured type provided that they have a common supertype. The static data type of the resulting structured type column is the structured type that is the least common supertype of either column.

For example, consider the following structured type hierarchy,

```
   A
  / \
 B   C
```

Rules for result data types

```
    / \
   D   E
  / \
 F   G
```

Structured types of the static type E and F are compatible with the resulting static type of B, which is the least common super type of E and F.

Nullable attribute of result

With the exception of INTERSECT and EXCEPT, the result allows nulls unless both operands do not allow nulls.

- For INTERSECT, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For EXCEPT, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).

Rules for string conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This topic explains those rules.

These rules apply to:

- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression and the DECODE scalar function
- Arguments of the scalar function COALESCE (also NVL and VALUE)
- Arguments of the scalar functions GREATEST, LEAST, MAX, and MIN
- The *source-string* and *insert-string* arguments of the scalar function OVERLAY (and INSERT)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to 'W' in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:

- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- 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 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 string, every non-supplementary character, including the halfwidth underscore (U+005F) and halfwidth percent (U+0025), is two bytes in width. For Unicode databases, the special characters in the pattern are interpreted as follows:

- For character strings, a halfwidth underscore (X'5F') or a fullwidth underscore (X'EFBCBF') refers to one Unicode character. A halfwidth percent (X'25') or a fullwidth percent (X'EFBC85') refers to zero or more Unicode characters.
- For graphic strings, a halfwidth underscore (U+005F) or a fullwidth underscore (U+FF3F) refers to one Unicode character. A halfwidth percent (U+0025) or a fullwidth percent (U+FF05) refers to zero or more Unicode characters.

Note: Two underscores are needed to match a Unicode supplementary graphic character because such a character is represented by two UCS-2 characters in a graphic string. Only one underscore is needed to match a Unicode supplementary character in a character string.

For the optional "escape expression", which specifies a character to be used to modify the special meaning of the underscore and percent sign characters, the expression can be specified by any one of:
- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the previously mentioned operands
- An expression concatenating any of the previously mentioned operands or functions

with the restrictions that:
- No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- For character strings, the result of the expression must be one character or a FOR BIT DATA string containing exactly one (1) byte (SQLSTATE 22019). For graphic strings, the result of the expression must be one character (SQLSTATE 22019).

Resolving the anchor object for an anchored type

The anchor object of an anchored type that does not specify ROW is specified with a name that could represent an SQL variable, an SQL parameter, a global variable, a module variable, a column of a table, or a column of a view.

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.

- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, or a global variable.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, the column of a table, or the column of a view.
- If the anchor object name is specified with 3 identifiers, the name represents a column of a schema-qualified table, a column of a schema-qualified view, a global variable qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body.

Resolving the anchor object name that has 1 identifier is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.
5. If not yet found, then search for a global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:

1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.
3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.

4. If not yet found, then search for a table or view column in the current schema using the first identifier as a table or view name and the second identifier as a column name.

5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.

6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.

7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.

2. If not found in the previous step or the step is not applicable, then search for a table or view column using the first identifier as a schema name, the second identifier as a table or view name and the third identifier as a column name.

3. If not found in the previous step and the first identifier is the same as the current server name, then search for a global variable using the second identifier as a schema name, and the third identifier as a global variable name.

4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published module variable name in the module.

Resolving the anchor object for an anchored row type

The anchor object of an anchored type that includes the ROW keyword is specified with a name that could represent a variety of objects, depending on the context and the number of identifiers in the name and the context of the ANCHOR clause.

The objects include the following:
- An SQL variable
- An SQL parameter
- A global variable
- A module variable
- A table
- A view

The way the anchor object is resolved depends on the number of identifiers in the anchor object name and the context of the statement using the ANCHOR clause.
- If the anchor object name is specified with 1 identifier, the name could represent an SQL variable, an SQL parameter, a module variable, a global variable, a table, or a view.
- If the anchor object name is specified with 2 identifiers, the name could represent a label-qualified SQL variable, a routine-qualified SQL parameter, a schema-qualified global variable, a module variable, a schema-qualified table, or a schema-qualified view.
- If the anchor object name is specified with 3 identifiers, the name could represent a global variable qualified by the current server name and a schema, a table qualified by the current server name and a schema, a view qualified by the current server name and a schema, or a schema-qualified module variable.

An SQL variable is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement. An SQL parameter is a candidate for an anchor object name only if the ANCHOR clause is used in an SQL variable declaration within a compound statement used in an SQL routine body. Resolving the anchor object name that has 1 identifier is done using the following steps:
1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching SQL variable name starting from the innermost nested compound to the outermost compound.
2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine.
3. If the ANCHOR clause is used in defining a module object, then search for a matching module variable name within the module.
4. If not yet found, then search for a table or view with a matching name in the current schema.
5. If not yet found, then search for a schema global variable with a matching global variable name on the SQL path.

Resolving the anchor object name that has 2 identifiers is done using the following steps:
1. If the ANCHOR clause is in an SQL variable declaration of a compound statement, search for a matching qualified SQL variable name starting from the innermost nested compound to the outermost compound.

2. If the ANCHOR clause is in an SQL variable declaration of a compound statement within a routine body, search for a matching SQL parameter name for the routine if the first identifier of the anchor object name matches the name of the routine.

3. If the ANCHOR clause is used in defining a module object and if the first identifier matches the module name of that module, then search for a module variable name within the module that matches the second identifier.

4. If not yet found, then search for a table or view using the first identifier as the schema name and the second identifier as the table or view name.

5. If not yet found, then search for a global variable using the first identifier as a schema name and the second identifier as a global variable name.

6. If not found and a module was not searched in step 3, then search for a module on the SQL path with a name that matches the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module.

7. If a module is not found using the SQL path in step 6, check for a module public alias that matches the name of the first identifier. If found, then use the second identifier to search for a matching published module variable name in the module identified by the module public alias.

Resolving the anchor object name that has 3 identifiers is done using the following steps:

1. If the ANCHOR clause is used in defining a module object and if the first 2 identifiers match the schema name and the module name of that module, then search for a module variable with a name that matches the last identifier.

2. If not found and the first identifier is the same as the current server name, then search for a table or view using the second identifier as the schema name and the third identifier as the table or view name.

3. If not found and the first identifier is the same as the current server name, then search for a global variable using the second identifier as the schema name and the third identifier as the global variable name.

4. If not found and a module was not searched in step 1, then search for a published module variable using the first identifier as a schema name, the second identifier as a module name and, if such a module exists, use the third identifier to search for a matching published variable name in the module.

Database partition-compatible data types

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

Table 20 shows the compatibility of data types in database partitions.

Database partition compatibility has the following characteristics:
- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with character or graphic data types.
- Partition compatibility is not affected by the nullability of a column.
- Partition compatibility is affected by collation. Locale-sensitive UCA-based collations require an exact match in collation, except that the strength (S) attribute of the collation is ignored. All other collations are considered equivalent for the purposes of determining partition compatibility.
- Character columns defined with FOR BIT DATA are only compatible with character columns without FOR BIT DATA when a collation other than a locale-sensitive UCA-based collation is used.
- Null values of compatible data types are treated identically. Different results might be produced for null values of non-compatible data types.
- Base data type of the UDT is used to analyze database partition compatibility.
- Timestamps of the same value in the distribution key are treated identically, even if their timestamp precisions differ.
- Decimals of the same value in the distribution key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.
- When a locale-sensitive UCA-based collation is used, CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC are compatible data types. When other collations are used, CHAR and VARCHAR are compatible types and GRAPHIC and VARGRAPHIC are compatible types, but CHAR and VARCHAR are not compatible types with GRAPHIC and VARGRAPHIC. CHAR or VARCHAR of different lengths are compatible data types.
- DECFLOAT values that are equal are treated identically even if their precision differs. DECFLOAT values that are numerically equal are treated identically even if they have a different number of significant digits.
- Data types that are not supported as part of a distribution key are not applicable for database partition compatibility. This includes columns whose data type is BLOB, CLOB, DBCLOB, XML, distinct type based on any of these data types, or structured type.

Table 20. Database Partition Compatibilities

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Date	Time	Time-stamp	Distinct Type
Binary Integer	Yes	No	No	No	No	No	No	No	No	[1]
Decimal Number	No	Yes	No	No	No	No	No	No	No	[1]

Table 20. Database Partition Compatibilities (continued)

Operands	Binary Integer	Decimal Number	Floating-point	Decimal Floating-point	Character String	Graphic String	Date	Time	Time-stamp	Distinct Type
Floating-point	No	No	Yes	No	No	No	No	No	No	[1]
Decimal Floating-point	No	No	No	Yes	No	No	No	No	No	[1]
Character String	No	No	No	No	Yes[2]	2, 3	No	No	No	[1]
Graphic String	No	No	No	No	2, 3	Yes[2]	No	No	No	[1]
Date	No	No	No	No	No	No	Yes	No	No	[1]
Time	No	No	No	No	No	No	No	Yes	No	[1]
Timestamp	No	No	No	No	No	No	No	No	Yes	[1]
Distinct Type	[1]	[1]	[1]	[1]	[1]	[1]	[1]	[1]	[1]	[1]

Note:

[1]   A distinct type value is database partition compatible with the source data type of the distinct type or with any other distinct type with the same source data type. The source data type of the distinct type must be a data type that is supported as part of a distribution key. A user-defined distinct type (UDT) value is database partition compatible with the source type of the UDT or any other UDT with a database partition compatible source type. A distinct type cannot be based on BLOB, CLOB, DBCLOB, or XML.

[2]   Character and graphic string types are compatible when they have compatible collations.

[3]   Character and graphic string types are compatible when a locale-sensitive UCA-based collation is in effect. Otherwise, they are not compatible types.

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the NOT NULL attribute.

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

User-defined types have strong typing, except for weakly typed distinct types.. This means that a strongly typed user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a strongly typed user-defined type and a constant is only possible if the user-defined type has been cast to the constant's built-in type, or if the constant has been cast to the user-defined type. For example, using the table and distinct type in "User-defined type comparisons" on page 141, the following comparisons with the constant 14 are valid:

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

The following comparison is not valid:

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

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2 147 483 647, and not -2 147 483 648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9 223 372 036 854 775 807, and not -9 223 372 036 854 775 808, which is the limit for big integer values.

Examples:

```
   64      -15      +100      32767      720176      12345678901
```

In syntax diagrams, the term 'integer' is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double-precision. The value of the constant is the product

of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of bytes in the constant must not exceed 30.

Examples:
```
15E1    2.E5    2.2E-1    +5.E+2
```

Decimal constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be within the range of decimal numbers. The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples:
```
25.5    1000.    -15.    +37589.3333333333
```

Decimal floating-point constants

There are no decimal floating-point constants except for the decimal floating-point special values, which are interpreted as DECFLOAT(34).

These special values are: INFINITY, NAN, and SNAN. INFINITY represents infinity, a number whose magnitude is infinitely large. INFINITY can be preceded by an optional sign. INF can be specified in place of INFINITY. NAN represents Not a Number (NaN) and is sometimes called quiet NaN. It is a value that represents undefined results which does not cause a warning or exception. SNAN represents signaling NaN (sNaN). It is a value that represents undefined results which will cause a warning or exception if used in any operation that is defined in any numeric operation. Both NAN and SNAN can be preceded by an optional sign, but the sign is not significant for arithmetic operations.. SNAN can be used in non-numeric operations without causing a warning or exception, for example in the VALUES list of an INSERT or as a constant compared in a predicate.
```
SNAN    -INFINITY
```

When one of the special values (INFINITY, INF, NAN, or SNAN) is used in a context where it could be interpreted as an identifier, such as a column name, cast a string representation of the special value to decimal floating-point. Examples:
```
CAST ('snan' AS DECFLOAT)
CAST ('INF' AS DECFLOAT)
CAST ('Nan' AS DECFLOAT)
```

All non-special values are interpreted as integer, floating-point or decimal constants, in accordance with the rules specified previously. To obtain a numeric decimal floating-point value, use the DECFLOAT cast function with a character string constant. It is not recommended to use floating-point constants as arguments to the DECFLOAT function, because floating-point is not exact and the resulting decimal floating-point value might be different than the decimal digit characters that make up the argument. Instead, use character constants as arguments to the DECFLOAT function.

For example, DECFLOAT('6.0221415E23', 34) returns the decimal floating-point value 6.0221415E+23, but DECFLOAT(6.0221415E23, 34) returns the decimal floating-point value 6.0221415000000003E+23.

Character string constants

A *character string constant* specifies a varying-length character string. The constant value string units are determined by the environment default string units. There are three forms of a character string constant:

- A sequence of characters that starts and ends with a string delimiter, which is an apostrophe ('). The number of bytes between the string delimiters cannot be greater than 32 672. When the environment string unit is CODEUNITS32, the number of code units cannot be greater than 8 168. 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. When the environment string unit is CODEUNITS32, the number of code units that the hexadecimal constant represents cannot be greater than 8 168. Two consecutive string delimiters are used to represent one string delimiter within the character string. A hexadecimal digit is a digit 0 through 9 or any of the letters A through F (uppercase or lowercase). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of a character string constant allows you to specify characters that do not have a keyboard representation.

- U& followed by a sequence of characters that starts and ends with a string delimiter and that is optionally followed by the UESCAPE clause. This form of a character string constant is also called a *Unicode string constant*. The number of bytes between the string delimiters cannot be greater than 32 672. When the environment string unit is CODEUNITS32, the number of code units that the Unicode string constant represents cannot be greater than 8 168. 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&'\0141ód\017A is a city in Poland'   U&'c:\\temp'   U&'@+01D11E' UESCAPE '@'
```

The rightmost string on the second line in the example represents the VARCHAR pattern of the ASCII string 'Frank'. The last line corresponds to: '△ód△ 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.

Non-Unicode databases
> In a non-Unicode database, a graphic string constant consists of a sequence of double-byte characters that starts and ends with a single-byte apostrophe ('), and that is preceded by a single-byte G or N. The characters between the apostrophes must represent an even number of bytes, and the length of the graphic string must not exceed 16 336 double bytes. The apostrophe must not appear as part of an MBCS character to be considered a delimiter. For example:
> ```
> G'double-byte character string'
> N'double-byte character string'
> ```

Unicode databases
> In a Unicode database, a graphic string constant consists of a sequence of characters that starts and ends with an apostrophe ('), and that is preceded by a G or N character. The constant value string units are determined by the environment default string units. The characters between the apostrophes are converted to code page 1200 and the length of the graphic string must not exceed 16 336 double bytes. When the environment string unit is CODEUNITS32, the number of code units must not exceed 8 168.

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 32 672. When the environment string unit is CODEUNITS32, the number of code units that the hexadecimal graphic string constant represents must not exceed 8 168; otherwise, an error is returned (SQLSTATE 54002). If a hexadecimal graphic string constant is improperly formed, an error is returned (SQLSTATE 42606). Each group of four digits represents a single graphic character. In 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.

Datetime constants

A *datetime constant* specifies a date, time, or timestamp.

Typically, character-string constants are used to represent constant datetime values in assignments and comparisons. However, the associated data type name can be used preceding specific formats of the character-string constant to specifically denote the constant as a datetime constant instead of a character-string constant. The format for the three datetime constants are:

DATE '*yyyy-mm-dd***'**
>   The data type of the value is DATE.

TIME '*hh:mm:ss***'**
or
TIME '*hh:mm***'**
>   The data type of the value is TIME.

TIMESTAMP '*yyyy-mm-dd hh:mm:ss.nnnnnnnnnnnn***'**
or
TIMESTAMP '*yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnnn***'**
>   where the number of digits of fractional seconds can vary from 0 to 12 and the period character can be omitted if there are no fractional seconds. The data type of the value is TIMESTAMP(p), where p is the number of digits of fractional seconds.

Leading zeros can be omitted from the month, day, and hour part of the character-string constant portion, where applicable, in each of these datetime constants. Leading zero characters must be included for minutes and seconds elements of TIME or TIMESTAMP constants. Trailing blanks can be included and are ignored.

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'.

Boolean constants

A Boolean constant specifies the keyword TRUE or FALSE, representing the truth values true and false respectively. The unknown truth value can be specified using CAST(NULL AS BOOLEAN).
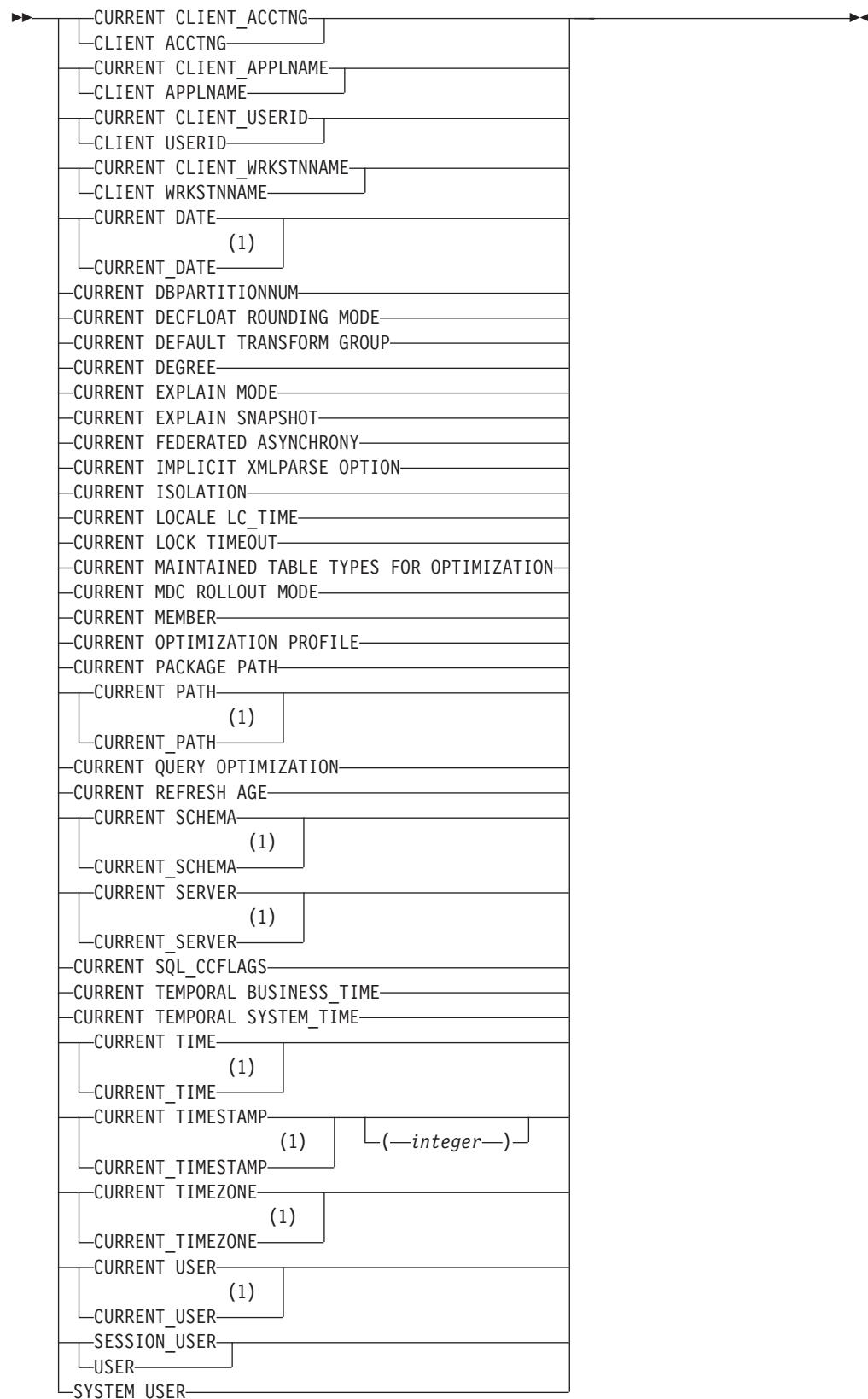
Special registers

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

A reference to a special register is a reference to a value provided by the current server. If the value is a string, its CCSID is a default CCSID of the current server.

The special registers can be referenced as follows:

Special registers

```
>>─────┬─CURRENT CLIENT_ACCTNG──────────────┬──────────────────────>◄
       ├─CLIENT ACCTNG──────────────┘       │
       ├─CURRENT CLIENT_APPLNAME────────────┤
       ├─CLIENT APPLNAME────────────┘       │
       ├─CURRENT CLIENT_USERID──────────────┤
       ├─CLIENT USERID──────────────┘       │
       ├─CURRENT CLIENT_WRKSTNNAME──────────┤
       ├─CLIENT WRKSTNNAME──────────┘       │
       ├─CURRENT DATE───────────────────────┤
       │              (1)                    │
       ├─CURRENT_DATE───────────────┘       │
       ├─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 LOCALE LC_TIME──────────────┤
       ├─CURRENT LOCK TIMEOUT────────────────┤
       ├─CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION─┤
       ├─CURRENT MDC ROLLOUT MODE────────────┤
       ├─CURRENT MEMBER──────────────────────┤
       ├─CURRENT OPTIMIZATION PROFILE────────┤
       ├─CURRENT PACKAGE PATH────────────────┤
       ├─CURRENT PATH───────────────────────┤
       │              (1)                    │
       ├─CURRENT_PATH───────────────┘       │
       ├─CURRENT QUERY OPTIMIZATION──────────┤
       ├─CURRENT REFRESH AGE─────────────────┤
       ├─CURRENT SCHEMA─────────────────────┤
       │              (1)                    │
       ├─CURRENT_SCHEMA─────────────┘       │
       ├─CURRENT SERVER─────────────────────┤
       │              (1)                    │
       ├─CURRENT_SERVER─────────────┘       │
       ├─CURRENT SQL_CCFLAGS─────────────────┤
       ├─CURRENT TEMPORAL BUSINESS_TIME──────┤
       ├─CURRENT TEMPORAL SYSTEM_TIME────────┤
       ├─CURRENT TIME───────────────────────┤
       │              (1)                    │
       ├─CURRENT_TIME───────────────┘       │
       ├─CURRENT TIMESTAMP──────────┬───────┤
       │              (1)           └─(─integer─)─┘
       ├─CURRENT_TIMESTAMP──────────┘       │
       ├─CURRENT TIMEZONE───────────────────┤
       │              (1)                    │
       ├─CURRENT_TIMEZONE───────────┘       │
       ├─CURRENT USER───────────────────────┤
       │              (1)                    │
       ├─CURRENT_USER───────────────┘       │
       ├─SESSION_USER───────────────┐       │
       ├─USER───────────────────────┘       │
       └─SYSTEM_USER─────────────────────────┘
```

Notes:

1   The SQL2008 Core standard uses the form with the underscore.

Some special registers can be updated using the SET statement. The following table shows which of the special registers can be updated as well as indicating which special register can be the null value.

Table 21. Updatable and nullable special registers

Special Register	Updatable	Nullable
CURRENT CLIENT_ACCTNG	No	No
CURRENT CLIENT_APPLNAME	No	No
CURRENT CLIENT_USERID	No	No
CURRENT CLIENT_WRKSTNNAME	No	No
CURRENT DATE	No	No
CURRENT DBPARTITIONNUM	No	No
CURRENT DECFLOAT ROUNDING MODE	No	No
CURRENT DEFAULT TRANSFORM GROUP	Yes	No
CURRENT DEGREE	Yes	No
CURRENT EXPLAIN MODE	Yes	No
CURRENT EXPLAIN SNAPSHOT	Yes	No
CURRENT FEDERATED ASYNCHRONY	Yes	No
CURRENT IMPLICIT XMLPARSE OPTION	Yes	No
CURRENT ISOLATION	Yes	No
"CURRENT LOCALE LC_MESSAGES" on page 183	Yes	No
CURRENT LOCALE LC_TIME	Yes	No
CURRENT LOCK TIMEOUT	Yes	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Yes	No
CURRENT MDC ROLLOUT MODE	Yes	No
CURRENT MEMBER	No	No
CURRENT OPTIMIZATION PROFILE	Yes	Yes
CURRENT PACKAGE PATH	Yes	No
CURRENT PATH	Yes	No
CURRENT QUERY OPTIMIZATION	Yes	No
CURRENT REFRESH AGE	Yes	No
CURRENT SCHEMA	Yes	No
CURRENT SERVER	No	No
CURRENT SQL_CCFLAGS	Yes	No
CURRENT TEMPORAL BUSINESS_TIME	Yes	Yes
CURRENT TEMPORAL SYSTEM_TIME	Yes	Yes
CURRENT TIME	No	No
CURRENT TIMESTAMP	No	No
CURRENT TIMEZONE	No	No
CURRENT USER	No	No
SESSION_USER	Yes	No
SYSTEM_USER	No	No
USER	Yes	No

Special registers

When a special register is referenced in a routine, the value of the special register in the routine depends on whether the special register is updatable or not. For non-updatable special registers, the value is set to the default value for the special register. For updatable special registers, the initial value is inherited from the invoker of the routine and can be changed with a subsequent SET statement inside the routine.

CURRENT CLIENT_ACCTNG

The CURRENT CLIENT_ACCTNG (or CLIENT ACCTNG) special register contains the value of the accounting string from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the accounting string can be changed by using the Set Client Information (sqleseti) API or the wlm_set_client_info procedure.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the current value of the accounting string for this connection.

```
VALUES (CURRENT CLIENT_ACCTNG)
  INTO :ACCT_STRING
```

CURRENT CLIENT_APPLNAME

The CURRENT CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name from the client information specified for this connection.

The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the application name can be changed by using the Set Client Information (sqleseti) API or the wlm_set_client_info procedure.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: 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 or the wlm_set_client_info procedure.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: 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 or the wlm_set_client_info procedure.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the workstation name being used for this connection.

```
VALUES (CURRENT CLIENT_WRKSTNNAME)
  INTO :WS_NAME
```

CURRENT DATE

The CURRENT DATE (or CURRENT_DATE) special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT DATE is not inherited from the invoking statement.

In a federated system, CURRENT DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT DATE register at the federated server, not from the data sources.

Examples

1. Run the following command from the DB2 CLP to obtain the current date.

       db2 values **CURRENT DATE**

2. Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

       UPDATE PROJECT
         SET PRENDATE = **CURRENT DATE**
         WHERE PROJNO = 'MA2111'

CURRENT DBPARTITIONNUM

The CURRENT DBPARTITIONNUM special register specifies an INTEGER value that identifies the coordinator database partition number for the statement.

For statements issued from an application, the coordinator is the database partition number to which the application connects.

For statements issued from a routine, the coordinator is the database partition number from which the routine is invoked.

When used in an SQL statement inside a routine, CURRENT DBPARTITIONNUM is never inherited from the invoking statement.

CURRENT DBPARTITIONNUM returns 0 if the database instance is not defined to support database partitioning. For a partitioned database, the db2nodes.cfg file exists and contains database partition and database partition number definitions.

In a database partitioning environment, the CURRENT DBPARTITIONNUM special register can be changed through the CONNECT statement, but only under certain conditions.

Examples

Example 1: Set the host variable APPL_DBPNUM (integer) to the number of the database partition to which the application is connected.

```
VALUES CURRENT DBPARTITIONNUM
  INTO :APPL_DBPNUM
```

Example 2: The following command is issued on member 0 and on a 4 member system in a partitioned database environment. This query will retrieve the currently connected database partition number.

```
VALUES CURRENT DBPARTITIONNUM

1
-----------
          0
```

CURRENT DECFLOAT ROUNDING MODE

The CURRENT DECFLOAT ROUNDING MODE special register specifies the rounding mode that is used for DECFLOAT values.

The data type is VARCHAR(128). The following rounding modes are supported:

- ROUND_CEILING rounds the value toward positive infinity. If all of the discarded digits are zero or if the sign is negative, the result is unchanged (except for the removal of the discarded digits). Otherwise, the result coefficient is incremented by 1.
- ROUND_DOWN rounds the value toward 0 (truncation). The discarded digits are ignored.
- ROUND_FLOOR rounds the value toward negative infinity. If all of the discarded digits are zero or if the sign is positive, the result is unchanged (except for the removal of the discarded digits). Otherwise, the sign is negative and the result coefficient is incremented by 1.
- ROUND_HALF_EVEN rounds the value to the nearest value. If the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent more than half of the value of a number in the next left position, the result coefficient is incremented by 1. If they represent less than half, the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise, the result coefficient is unaltered if its rightmost digit is even, or incremented by 1 if its rightmost digit is odd (to make an even digit).
- ROUND_HALF_UP rounds the value to the nearest value. If the values are equidistant, rounds the value up. If the discarded digits represent half or more than half of the value of a number in the next left position, the result coefficient is incremented by 1. Otherwise, the discarded digits are ignored.

The value of the DECFLOAT rounding mode on a client can be confirmed to match that of the server by invoking the SET CURRENT DECFLOAT ROUNDING MODE statement. However, this statement cannot be used to change the rounding mode of the server. The initial value of CURRENT DECFLOAT ROUNDING MODE is determined by the `decflt_rounding` database configuration parameter and can only be changed by changing the value of this database configuration parameter.

CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR(18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs.

This special register does not specify the transform groups used in static SQL statements, or in the exchange of parameters and results with external functions or methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a length of zero).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using the SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Examples

- Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

  SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1

- Retrieve the name of the default transform group assigned to this special register.

  VALUES (CURRENT DEFAULT TRANSFORM GROUP)

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intrapartition parallelism for the execution of dynamic SQL statements. (For static SQL, the DEGREE bind option provides the same control.)

The data type of the register is CHAR(5). Valid values are ANY or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intrapartition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32 767 when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism with the specified degree.

If the value of CURRENT DEGREE is ANY when an SQL statement is dynamically prepared, the execution of that statement can involve intrapartition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:
- The value of the maximum query degree (`max_querydegree`) configuration parameter
- The application runtime degree
- The SQL statement compilation degree.

If the `intra_parallel` database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intrapartition parallelism.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE is determined by the `dft_degree` database configuration parameter.

The value in the CURRENT DEGREE special register and the `intra_parallel` setting can be overridden in a workload by setting the MAXIMUM DEGREE workload attribute.

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements.

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables. This information does not include the Explain snapshot. Possible values are YES, EXPLAIN, NO, REOPT, RECOMMEND INDEXES, and EVALUATE INDEXES. (For static SQL, the **EXPLAIN** bind option provides the same control. In the case of the **PREP** and **BIND** commands, the **EXPLAIN** option values are: YES, NO, and ALL).

YES Enables the Explain facility and causes Explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN
Enables the facility, but dynamic statements are not executed.

NO Disables the Explain facility.

REOPT
Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

RECOMMEND INDEXES
Recommends a set of indexes for each dynamic query. Populates the ADVISE_INDEX table with the set of indexes.

EVALUATE INDEXES
Enables the SQL compiler to evaluate virtual recommended indexes for dynamic queries. Queries executed in this explain mode will be compiled and optimized using fabricated statistics based on the virtual indexes. The statements are not executed. The indexes to be evaluated are read from the ADVISE_INDEX table if the USE_INDEX column contains 'Y'. Existing non-unique indexes can also be ignored by setting the USE_INDEX column to 'I' and the EXISTS column to 'Y'. If a combination of USE_INDEX='I' and EXISTS='N' is given then index evaluation for the query will continue normally but the index in question will not be ignored.

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN MODE statement.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option. RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
  INTO :EXPL_MODE
```

CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value that controls the behavior of the Explain snapshot facility. This facility generates compressed information, including access plan information, operator costs, and bind-time statistics.

Only the following statements consider the value of this register: CALL, Compound SQL (Dynamic), DELETE, INSERT, MERGE, REFRESH, SELECT, SELECT INTO, SET INTEGRITY, UPDATE, VALUES, or VALUES INTO. Possible values are YES, EXPLAIN, NO, and REOPT. (For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO, and ALL.)

YES   Enables the Explain snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.

EXPLAIN
          Enables the Explain snapshot facility, but dynamic statements are not executed.

NO   Disables the Explain snapshot facility.

REOPT
          Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, global variables, or parameter markers).

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN SNAPSHOT statement.

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option.

Example: Set the host variable EXPL_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

```
VALUES CURRENT EXPLAIN SNAPSHOT
  INTO :EXPL_SNAP
```

CURRENT FEDERATED ASYNCHRONY

The CURRENT FEDERATED ASYNCHRONY special register specifies the degree of asynchrony for the execution of dynamic SQL statements. The FEDERATED_ASYNCHRONY bind option provides the same control for static SQL.

The data type of the register is INTEGER. Valid values are ANY (representing -1) or an integer between 0 and 32 767, inclusive. If, when an SQL statement is dynamically prepared, the value of CURRENT FEDERATED ASYNCHRONY is:

- 0, the execution of that statement will not use asynchrony
- Greater than 0 and less than or equal to 32 767, the execution of that statement can involve asynchrony using the specified degree
- ANY (representing -1), the execution of that statement can involve asynchrony using a degree that is determined by the database manager

The value of the CURRENT FEDERATED ASYNCHRONY special register can be changed by invoking the SET CURRENT FEDERATED ASYNCHRONY statement.

The initial value of the CURRENT FEDERATED ASYNCHRONY special register is determined by the **federated_async** database manager configuration parameter if the dynamic statement is issued through the command line processor (CLP). The initial value is determined by the FEDERATED_ASYNCHRONY bind option if the dynamic statement is part of an application that is being bound.

Example: Set the host variable FEDASYNC (INTEGER) to the value of the CURRENT FEDERATED ASYNCHRONY special register.

```
VALUES CURRENT FEDERATED ASYNCHRONY INTO :FEDASYNC
```

CURRENT IMPLICIT XMLPARSE OPTION

The CURRENT IMPLICIT XMLPARSE OPTION special register specifies the whitespace handling options that are to be used when serialized XML data is implicitly parsed by the database server, without validation.

An implicit non-validating parse operation occurs when an SQL statement is processing an XML host variable or an implicitly or explicitly typed XML parameter marker that is not an argument of the XMLVALIDATE function. The data type of the register is VARCHAR(19).

The value of the CURRENT IMPLICIT XMLPARSE OPTION special register can be changed by invoking the SET CURRENT IMPLICIT XMLPARSE OPTION statement. Its initial value is 'STRIP WHITESPACE'.

Examples

- Retrieve the value of the CURRENT IMPLICIT XMLPARSE OPTION special register into a host variable named CURXMLPARSEOPT:

  ```
  EXEC SQL VALUES (CURRENT IMPLICIT XMLPARSE OPTION) INTO :CURXMLPARSEOPT;
  ```

- Set the CURRENT IMPLICIT XMLPARSE OPTION special register to 'PRESERVE WHITESPACE'.

  ```
  SET CURRENT IMPLICIT XMLPARSE OPTION = 'PRESERVE WHITESPACE'
  ```

  Whitespace is then preserved when the following SQL statement executes:

  ```
  INSERT INTO T1 (XMLCOL1) VALUES (?)
  ```

CURRENT ISOLATION

The CURRENT ISOLATION special register holds a CHAR(2) value that identifies the isolation level (in relation to other concurrent sessions) for any dynamic SQL statements issued within the current session.

The possible values are:

(blanks)
Not set; use the isolation attribute of the package.

UR   Uncommitted Read

CS   Cursor Stability

RR   Repeatable Read

RS   Read Stability

The value of the CURRENT ISOLATION special register can be changed by the SET CURRENT ISOLATION statement.

Until a SET CURRENT ISOLATION statement is issued within a session, or after RESET has been specified for SET CURRENT ISOLATION, the CURRENT ISOLATION special register is set to blanks and is not applied to dynamic SQL statements; the isolation level used is taken from the isolation attribute of the package which issued the dynamic SQL statement. Once a SET CURRENT ISOLATION statement has been issued, the CURRENT ISOLATION special register provides the isolation level for any subsequent dynamic SQL statement compiled within the session, regardless of the settings for the package issuing the statement. This will remain in effect until the session ends or until a SET CURRENT ISOLATION statement is issued with the RESET option.

Example: Set the host variable ISOLATION_MODE (CHAR(2)) to the value currently stored in the CURRENT ISOLATION special register.

```
VALUES CURRENT ISOLATION
  INTO :ISOLATION_MODE
```

CURRENT LOCALE LC_MESSAGES

The CURRENT LOCALE LC_MESSAGES special register identifies the locale that is used by EVMON_UPGRADE_TABLES as well as monitoring routines in the **monreport** module.

EVMON_UPGRADE_TABLES and the monitoring routines use the value of CURRENT LOCALE LC_MESSAGES to determine in which language the result set text output should be returned. User-defined routines that are coded to return messages could also use the value of CURRENT LOCALE LC_MESSAGES to determine what language to use for message text.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_MESSAGES is "en_US" for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_MESSAGES statement.

CURRENT LOCALE LC_TIME

The CURRENT LOCALE LC_TIME special register identifies the locale that is used for SQL statements that involve the datetime related built-in functions.

These functions include DAYNAME, MONTHNAME, NEXT_DAY, ROUND, ROUND_TIMESTAMP, TIMESTAMP_FORMAT, TRUNCATE, TRUNC_TIMESTAMP and VARCHAR_FORMAT. These functions use the value of CURRENT LOCALE LC_TIME if the *locale-name* argument is not explicitly specified.

The data type is VARCHAR(128).

The initial value of CURRENT LOCALE LC_TIME is "en_US" for English (United States). The value can be changed by invoking the SET CURRENT LOCALE LC_TIME statement.

CURRENT LOCK TIMEOUT

The CURRENT LOCK TIMEOUT special register specifies the number of seconds to wait for a lock before returning an error indicating that a lock cannot be obtained. This special register impacts row, table, index key, block index, 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. The initial value is null; in this case, the current value of the **locktimeout** database configuration parameter is used when waiting for a lock, and this value is returned for the special register.

CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION

The CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register specifies a VARCHAR(254) value that identifies the types of tables that can be considered when optimizing the processing of dynamic SQL queries. Materialized query tables are never considered by static embedded SQL queries.

The initial value of CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION is SYSTEM. Its value can be changed by the SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement.

CURRENT MDC ROLLOUT MODE

The CURRENT MDC ROLLOUT MODE special register specifies the behavior on multidimensional clustering (MDC) tables of DELETE statements that qualify for rollout processing.

The default value of this register is determined by the DB2_MDC_ROLLOUT registry variable. The value can be changed by invoking the SET CURRENT MDC ROLLOUT MODE statement. When the CURRENT MDC ROLLOUT MODE special register is set to a particular value, the execution behavior of subsequent DELETE statements that qualify for rollout is impacted. The DELETE statement does not need to be recompiled for the behavior to change.

CURRENT MEMBER

The CURRENT MEMBER special register specifies an INTEGER value that identifies the coordinator member for the statement.

For statements issued from an application, the coordinator is the member to which the application connects. For statements issued from a routine, the coordinator is the member from which the routine is invoked.

When used in an SQL statement inside a routine, CURRENT MEMBER is never inherited from the invoking statement.

CURRENT MEMBER returns 0 if the database instance is not defined to support database partitioning or the IBM DB2 pureScale Feature. The database instance is not defined for such support if there is no db2nodes.cfg file. For a partitioned database or a DB2 pureScale environment, the db2nodes.cfg file exists and contains database partition and member definitions.

CURRENT MEMBER can be changed through the CONNECT statement, but only under certain conditions.

For compatibility with previous versions of DB2 and with other database products, NODE can be specified in place of MEMBER.

Examples

Example 1: Set the host variable APPL_NODE (integer) to the number of the member to which the application is connected.

```
VALUES CURRENT MEMBER
  INTO :APPL_NODE
```

Example 2: The following command is issued on member 0 and on a 4 member system in a partitioned database environment. This query will retrieve the currently connected database member number.

```
VALUES CURRENT MEMBER

1
-----------
          0
```

CURRENT OPTIMIZATION PROFILE

The CURRENT OPTIMIZATION PROFILE special register specifies the qualified name of the optimization profile to be used by DML statements that are dynamically prepared for optimization.

The initial value is the null value. The value can be changed by invoking the SET CURRENT OPTIMIZATION PROFILE statement. An optimization profile that is not qualified with a schema name will be implicitly qualified with the value of the CURRENT DEFAULT SCHEMA special register.

Example 1: Set the optimization profile to 'JON.SALES'.

```
SET CURRENT OPTIMIZATION PROFILE = JON.SALES
```

Example 2: Get the current value of the optimization profile name for this connection.

```
VALUES (CURRENT OPTIMIZATION PROFILE) INTO :PROFILE
```

CURRENT PACKAGE PATH

The CURRENT PACKAGE PATH special register specifies a VARCHAR(4096) value that identifies the path to be used when resolving references to packages that are needed when executing SQL statements.

The value can be an empty or a blank string, or a list of one or more schema names that are delimited with double quotation marks and separated by commas. Any double quotation marks appearing as part of the string will need to be represented as two double quotation marks, as is common practice with delimited identifiers. The delimiters and commas contribute to the length of the special register.

This special register applies to both static and dynamic statements.

The initial value of CURRENT PACKAGE PATH in a user-defined function, method, or procedure is inherited from the invoking application. In other contexts, the initial value of CURRENT PACKAGE PATH is an empty string. The value is a list of schemas only if the application process has explicitly specified a list of schemas by means of the SET CURRENT PACKAGE PATH statement.

Examples

- An application will be using multiple SQLJ packages (in schemas SQLJ1 and SQLJ2) and a JDBC package (in schema DB2JAVA). Set the CURRENT PACKAGE PATH special register to check SQLJ1, SQLJ2, and DB2JAVA, in that order.

      SET CURRENT PACKAGE PATH = "SQLJ1", "SQLJ2", "DB2JAVA"

- Set the host variable HVPKLIST to the value currently stored in the CURRENT PACKAGE PATH special register.

      VALUES CURRENT PACKAGE PATH INTO :HVPKLIST

CURRENT PATH

The CURRENT PATH (or CURRENT_PATH) special register specifies a VARCHAR(2048) value that identifies the SQL path used when resolving unqualified function names, procedure names, data type names, global variable names, and module object names in dynamically prepared SQL statements. CURRENT FUNCTION PATH is a synonym for CURRENT PATH.

The initial value is the default value specified in a following paragraph. For static SQL, the FUNCPATH bind option provides an SQL path that is used for function and data type resolution.

The CURRENT PATH special register contains a list of one or more schema names that are enclosed by double quotation marks and separated by commas. For example, an SQL path specifying that the database manager is to look first in the FERMAT schema, then in the XGRAPHIC schema, and finally in the SYSIBM schema, is returned in the CURRENT PATH special register as:

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

The default value is "SYSIBM","SYSFUN","SYSPROC","SYSIBMADM",*X*, where *X* is the value of the USER special register, delimited by double quotation marks. The value can be changed by invoking the SET CURRENT PATH statement. The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed to be the first schema. SYSIBM does not take up any of the 2048 bytes if it is implicitly assumed.

A data type that is not qualified with a schema name will be implicitly qualified with the first schema in the SQL path that contains a data type with the same unqualified name. There are exceptions to this rule, as outlined in the descriptions of the following statements: CREATE TYPE (Distinct), CREATE FUNCTION, COMMENT, and DROP.

Example: Using the SYSCAT.ROUTINES catalog view, find all user-defined routines that can be invoked without qualifying the routine name, because the CURRENT PATH special register contains the schema name.

```
SELECT ROUTINENAME, ROUTINESCHEMA FROM SYSCAT.ROUTINES
  WHERE POSITION (ROUTINESCHEMA, CURRENT PATH, CODEUNITS16) <> 0
```

CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements.

The QUERYOPT bind option controls the class of query optimization for static SQL statements. The possible values range from 0 to 9. For example, if the query optimization class is set to 0 (minimal optimization), then the value in the special register is 0. The default value is determined by the **dft_queryopt** database configuration parameter. The value can be changed by invoking the SET CURRENT QUERY OPTIMIZATION statement.

Example: Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
  WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

CURRENT REFRESH AGE

The CURRENT REFRESH AGE special register specifies a timestamp duration value with a data type of DECIMAL(20,6).

It is the maximum duration since a particular time-stamped event occurred to a cached data object (for example, a REFRESH TABLE statement is 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. For example, 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 refresh deferred materialized query tables that are specified in CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION are considered when optimizing the processing of a dynamic SQL query.

The value of the CURRENT REFRESH AGE must be a value between 0 and 99 999 999 999 999 inclusive. The default value is determined by the **dft_refresh_age** database configuration parameter. 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 setting of CURRENT SCHEMA does not affect the Explain facility's selection of explain tables.

The QUALIFIER bind option controls the schema name used to qualify database object references, where applicable, for static SQL statements.

Example: Set the schema for object qualification to 'D123'.

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

CURRENT SERVER

The CURRENT SERVER (or CURRENT_SERVER) special register specifies a VARCHAR(18) value that identifies the current database server (sometimes referred to as the application server). The register contains the actual name of the database, not an alias.

CURRENT SERVER can be changed through the CONNECT statement, but only under certain conditions.

When used in an SQL statement inside a routine, CURRENT SERVER is not inherited from the invoking statement.

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

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

CURRENT SQL_CCFLAGS

The CURRENT SQL_CCFLAGS special register specifies the conditional compilation named constants that are defined for use during compilation of SQL statements.

The data type of the special register is VARCHAR(1024).

The CURRENT SQL_CCFLAGS special register contains a list of name and value pairs separated by a comma and a blank. The name is separated from the value in a pair using the colon character. The values in the list are a BOOLEAN constant, an INTEGER constant, or the keyword NULL. The names can be specified using any combination of uppercase or lowercase characters which are folded to all uppercase characters. For example, conditional compilation values defined for debug and tracing could appear in the special register as the string value:

```
CC_DEBUG:TRUE, CC_TRACE_LEVEL:2
```

The initial value of the special register is the value of the **sql_ccflags** database configuration parameter when the special register is first used. The first use can occur as a result of processing a statement with an inquiry directive or as a direct reference to the special register. If the value assigned to the **sql_ccflags** database configuration parameter is not valid, an error is returned on the first use (SQLSTATE 42815 or 428HV).

The value of the special register can be changed by executing the SET CURRENT SQL_CCFLAGS statement.

CURRENT TEMPORAL BUSINESS_TIME

The CURRENT TEMPORAL BUSINESS_TIME special register specifies a TIMESTAMP(12) value that is used in the default BUSINESS_TIME period specification for references to application-period temporal tables.

When an application-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is represented by *CTBT*, which is the non-null value, the following period specification is implicit:

```
FOR BUSINESS_TIME AS OF CTBT
```

When an application-period temporal table is the target of an UPDATE or DELETE statement and the value in effect for the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value, the following additional predicate is implicit:

```
bt_begin <= CURRENT TEMPORAL BUSINESS_TIME
    AND bt_end > CURRENT TEMPORAL BUSINESS_TIME
```

where `bt_begin` and `bt_end` are the begin and end columns of the BUSINESS_TIME period of the target table of the UPDATE statement.

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the null value.

The value of this special register can be changed by executing the SET CURRENT TEMPORAL BUSINESS_TIME statement.

The setting of the CURRENT TEMPORAL BUSINESS_TIME special register impacts the following compiled SQL objects when the associated package is bound with the BUSTIMESENSITIVE bind option set to YES:

- SQL procedures
- Compiled functions
- Compiled triggers
- Compound SQL (compiled) statements
- External UDFs

The setting for the BUSTIMESENSITIVE bind option determines whether references to application-period temporal tables and bitemporal tables in both static SQL statements and dynamic SQL statements in a package are affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register. The bind option can be set to YES or NO.

For the following examples, assume the table IN_TRAY is an application-period temporal table.

Example 1: Based on the state of the messages in IN_TRAY as of the date specified by the CURRENT TEMPORAL BUSINESS_TIME special register, list the user IDs and subject lines.

```
SELECT SOURCE, SUBJECT FROM IN_TRAY
```

Assuming that the CURRENT TEMPORAL BUSINESS_TIME special register was previously set to the value CURRENT TIMESTAMP-4 DAYS and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR BUSINESS_TIME AS OF CURRENT TIMESTAMP-4 DAYS
```

Example 2: List the user ID and subject line for the messages in IN_TRAY sent before the date specified by the CURRENT TEMPORAL BUSINESS_TIME special register.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    WHERE DATE(RECEIVED) < DATE(CURRENT TEMPORAL BUSINESS_TIME)
```

Assuming that the CURRENT TEMPORAL BUSINESS_TIME special register was previously set to '2011-01-01-00.00.00' and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR BUSINESS_TIME AS OF '2011-01-01-00.00.00'
    WHERE DATE(RECEIVED) < DATE('2011-01-01-00.00.00')
```

CURRENT TEMPORAL SYSTEM_TIME

The CURRENT TEMPORAL SYSTEM_TIME special register specifies a TIMESTAMP(12) value that is used in the default SYSTEM_TIME period specification for references to system-period temporal tables.

When a system-period temporal table is referenced and the value in effect for the CURRENT TEMPORAL SYSTEM_TIME special register is represented by *CTST*, which is the non-null value, , the following period specification is implicit:

```
FOR SYSTEM_TIME AS OF CTST
```

The initial value of the special register in a user-defined function or procedure is inherited from the invoking application. In other contexts the initial value of the special register is the null value.

The value of this special register can be changed by executing the SET CURRENT TEMPORAL SYSTEM_TIME statement.

The setting of the CURRENT TEMPORAL SYSTEM_TIME special register impacts the following compiled SQL objects when they have been bound with the SYSTIMESENSITIVE bind option set to YES:

- SQL procedures
- Compiled functions
- Compiled triggers
- Compound SQL (compiled) statements
- External UDFs)

The setting for the SYSTIMESENSITIVE bind option determines whether references to system-period temporal tables in both static SQL statements and dynamic SQL statements in a package are affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. The bind option can be set to YES or NO.

For the following examples, assume the table IN_TRAY is a system-period temporal table.

Example 1: Based on the state of the messages in IN_TRAY as of the date specified by the CURRENT TEMPORAL SYSTEM_TIME special register, list the user IDs and subject lines.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
```

Assuming that the CURRENT TEMPORAL SYSTEM_TIME special register was previously set to the value CURRENT TIMESTAMP-7 DAYS and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR SYSTEM_TIME AS OF CURRENT TEMPORAL SYSTEM_TIME
```

Example 2: List the user ID and subject line for the messages in IN_TRAY sent before the value specified by the CURRENT TEMPORAL SYSTEM_TIME special register.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    WHERE RECEIVED < CURRENT TEMPORAL SYSTEM_TIME
```

CURRENT TEMPORAL SYSTEM_TIME

Assuming that the CURRENT TEMPORAL SYSTEM_TIME special register was previously set to '2011-01-01-00.00.00' and is currently set to the null value, the following statement returns the same result.

```
SELECT SOURCE, SUBJECT
    FROM IN_TRAY
    FOR SYSTEM_TIME AS OF '2011-01-01-00.00.00'
    WHERE DATE(RECEIVED) < DATE('2011-01-01-00.00.00')
```

CURRENT TIME

The CURRENT TIME (or CURRENT_TIME) special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT TIME is not inherited from the invoking statement.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Examples

1. Run the following command from the DB2 CLP to obtain the current time.

   ```
   db2 values CURRENT TIME
   ```

2. Using the CL_SCHED table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

   ```
   SELECT CLASS_CODE FROM CL_SCHED
     WHERE STARTING > CURRENT TIME AND DAY = 3
   ```

CURRENT TIMESTAMP

The CURRENT TIMESTAMP (or CURRENT_TIMESTAMP) special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server.

If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading. It is possible for separate CURRENT TIMESTAMP special register requests to return the same value; if unique values are required, consider using the GENERATE_UNIQUE function, a sequence, or an identity column.

If a timestamp with a specific precision is desired, the special register can be referenced as CURRENT TIMESTAMP(*integer*), where *integer* can range from 0 to 12. The default precision is 6. The precision of the clock reading varies by platform and the resulting value is padded with zeros where the precision of the retrieved clock reading is less than the precision of the request.

When used in an SQL statement inside a routine, CURRENT TIMESTAMP is not inherited from the invoking statement.

In a federated system, CURRENT TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT TIMESTAMP register at the federated server, not from the data sources.

On a DB2 pureScale instance, with transaction workload balancing enabled, the CURRENT TIMESTAMP special register does not necessarily return increasing values across transactions if those transactions are executed on different members.

SYSDATE can also be specified as a synonym for CURRENT TIMESTAMP(0).

Example: Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
  VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

CURRENT TIMEZONE

The CURRENT TIMEZONE (or CURRENT_TIMEZONE) special register specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server.

The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. (The CURRENT TIMEZONE value is determined from C runtime functions.)

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used; for example, in time and timestamp arithmetic.

When used in an SQL statement inside a routine, CURRENT TIMEZONE is not inherited from the invoking statement.

Example: Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```
INSERT INTO IN_TRAY VALUES (
  CURRENT TIMESTAMP - CURRENT TIMEZONE,
  :source,
  :subject,
  :notetext )
```

CURRENT USER

The CURRENT USER (or CURRENT_USER) special register specifies the authorization ID that is used for statement authorization for the statement in which it was referenced.

For dynamic SQL statements, the value depends on the dynamic SQL statement behavior in effect for the package issuing the dynamic SQL statement in which this special register is referenced. See "Effect of DYNAMICRULES bind option on dynamic SQL" for details. The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

Example: Select table names whose schema matches the value of the CURRENT USER special register.

```
SELECT TABNAME FROM SYSCAT.TABLES
  WHERE TABSCHEMA = CURRENT USER AND TYPE = 'T'
```

If this statement is executed as a static SQL statement, it returns the tables whose schema name matches the binder of the package that includes the statement. If this statement is executed as a dynamic SQL statement using dynamic SQL statement run behavior, it returns the tables whose schema name matches the current value of the SESSION_USER special register.

SESSION_USER

The SESSION_USER special register specifies the current runtime authorization ID that is being used for the current session.

The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register. Its value can be changed by invoking the SET SESSION AUTHORIZATION statement.

SESSION_USER is a synonym for the USER special register.

Example: Determine what routines can be executed by current runtime authorization ID if it were to issue invocations through dynamic SQL.

```
SELECT SCHEMA, SPECIFICNAME FROM SYSCAT.ROUTINEAUTH
  WHERE GRANTEE = SESSION_USER
  AND EXECUTEAUTH IN ('Y', 'G')
```

SYSTEM_USER

The SYSTEM_USER special register specifies the authorization ID of the user that connected to the database.

The value of this register can only be changed by connecting as a user with a different authorization ID. The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

See "Example" in the description of the SET SESSION AUTHORIZATION statement.

USER

The USER special register specifies the runtime authorization ID that is used for the current session.

The data type of the register is VARCHAR(128). If the length of the authorization ID is less than 8 bytes, the special register value is padded with trailing blanks such that the length is 8 bytes.

USER is a synonym for the SESSION_USER special register. SESSION_USER is the preferred spelling.

Example: Select all notes from the IN_TRAY table that were placed there by the user.

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

Global variables

A global variable is a named memory variable that is retrieved or modified through SQL statements.

Global variables enable applications to share relational data among SQL statements, without the need for additional application logic to support this data transfer.

A global variable is defined within a schema. A global variable defined in a module within a schema is referred to as a *module global variable*. All other global variables are referred to as *schema global variables*.

The definitions for global variables are recorded in the system catalogs.

Types of global variables

There are three different ways to classify global variables: by the ownership of the variable, by the scope of the value, and by the method used to maintain the value.

Ownership of the variable

A global variable can be classified based on whether the variable is owned by the database manager, or if the variable is user-defined:

- The database manager creates *built-in global variables*. Built-in global variables are registered to the database manager in the system catalog. Built-in global variables belong to the following schema:
  - SYSIBM
  - SYSIBMADM

  Some built-in *module global variables* are located inside modules within the SYSIBMADM schema.
- A user creates a *user-defined global variable* by using an SQL DDL statement. User-defined global variables are registered to the database manager in the system catalog. A *user-defined schema global variable* is created by using the CREATE VARIABLE SQL statement. A *user-defined module global variable* is created using the ADD VARIABLE or PUBLISH VARIABLE option of the ALTER MODULE SQL statement.

Scope of the value

A global variable can be classified as either session or database based on the scope of the value:

- The value of a *session global variable* is uniquely associated with each session that uses this particular global variable. Session global variables are either built-in global variables or user-defined global variables.
- The value of a *database global variable* is a single value that remains the same for all sessions that use this particular global variable. Database global variables are always built-in global variables.

Method by which the value is maintained

A global variable can be classified based on how the variable is maintained:

- A *constant global variable* has a fixed value that is instantiated based on evaluation of the CONSTANT clause when the global variable is first referenced

in the session or the database, depending on the scope of the global variable. This type of global variable is created by using the CONSTANT clause in the CREATE VARIABLE statement. A value cannot be assigned to the global variable using an SQL statement. Constant global variables are read-only global variables.

- A *maintained-by-system global variable* has a value that is set by the database manager. A value cannot be assigned using an SQL statement. Only built-in global variables can be defined as maintained-by-system global variables, and most built-in global variables are defined as maintained-by-system global variables. Maintained-by-system global variables are read-only global variables.

- A *maintained-by-user global variable* can be assigned a value using an SQL statement; however, this assignment requires WRITE privilege on the global variable. This type of global variable is the default for user-defined global variables that are defined without using a CONSTANT clause. Built-in global variables can also be defined so values can be assigned using an SQL statement.

Authorization required for global variables

To access a global variable, the authorization ID requires certain privileges or DATAACCESS authority.

Schema global variables

To retrieve the value of a schema global variable, the authorization ID of the statement must have one of these authorizations:

- READ privilege on the schema global variable
- DATAACCESS authority

To specify a schema global variable as the target of a value assignment, the authorization ID of the statement must have one of these authorizations:

- WRITE privilege on the schema global variable
- DATAACCESS authority

Module global variables

If a module global variable is published and is then referenced from outside the module that defined it, the authorization ID of the statement must have one of these authorizations:

- EXECUTE privilege on the module in which the global variable is defined
- DATAACCESS authority

References to module global variables from objects that are defined within the same module as the global variable do not require any authorizations to be held by the authorization ID of the statement.

Resolution of global variable references

Global variable reference resolution depends on whether a global variable name is qualified and where the global variable is referenced.

The order of resolving a global variable reference in relation to the names for a column, SQL variable, SQL parameter, or row variable field is described in "References to SQL parameters, SQL variables, and global variables" in SQL Reference Volume 2.

Resolution of global variable references

The implicit qualification of an unqualified global variable name that is used as the main object of a CREATE, ALTER, COMMENT, DROP, GRANT, or REVOKE statement is described in "Unqualified user-defined type, function, procedure, specific, global variable and module names" on page 85.

A best practice is to fully qualify the name of the global variable when referencing the global variable in an SQL statement. This prevents a subsequent change in the SQL path from having an impact on the resolution of the global variable.

The resolution of a global variable reference by the database manager in all other contexts depends on whether the global variable name is qualified.

Qualified names

To resolve the name of a qualified global variable, the reference is evaluated according to the following process:
1. If the global variable reference is made from within a module and the qualifier matches the name of the module, the module is searched for a matching module global variable. The following rules are applied:
   - If the qualifier is a single identifier, the schema name of the module is ignored when the qualifier is compared to the module name.
   - If the qualifier is a two-part identifier, it is compared to the schema-qualified module name.

   If the name of a module global variable matches the unqualified global variable name in the reference, resolution is complete. If the qualifier does not match the name of the module or there is no matching module global variable, resolution continues with the next step.
2. The qualifier is now considered to be a schema name. That specified schema is searched for a matching schema global variable.
   - If a schema global variable name matches the unqualified global variable name in the reference, resolution is complete.
   - If the schema does not exist, then an error is returned.
   - If there are no matching schema global variables in the schema, and the qualifier matched the name of the module in the first step, then an error is returned.
   - Otherwise, resolution continues with the next step.
3. The qualifier is now considered to be a module name. The following rules are applied:
   - If the module name is qualified with a schema name, the module is searched for a matching published module global variable.
   - If the module name is not qualified with a schema name, the schema for the module is the first schema in the SQL path that has a matching module name. If the name of a module matches the schema name that is found in the SQL path, that module is searched for a matching published module global variable.
   - If the module is not found through the SQL path, the existence of a module public alias that matches the name of the global variable qualifier is considered. If a module public alias is found, the module that is associated with the module public alias is searched for a matching published module global variable.

   If the name of a published module global variable matches the unqualified global variable name in the global variable reference, resolution is complete. If

a matching module is not found or there is no matching module global variable in the matching module, an error is returned.

Unqualified names

To resolve the name of an unqualified global variable, the reference is evaluated according to the following process:

1. If an unqualified global variable reference is made from within a module, the module is searched for a matching module global variable. If a module global variable name matches the global variable name in the reference, resolution is complete. If there is no matching module global variable, resolution continues with the next step.

2. The schemas in the SQL path are searched in order from left to right for a matching schema global variable. If a schema global variable name matches the global variable name in the reference, resolution is complete. If no matching global variable is found after completing this step, an error is returned.

Using global variables

Using global variables requires an understanding of the usage restrictions, the rules for assignment to global variables, and the rules for retrieving global variable values.

Usage restrictions

Global variables can be referenced from within any SQL expression, unless the context of the expression requires that the expression be deterministic. The following situations are examples of contexts that require deterministic expressions and therefore preclude the use of global variables:

- In a table check constraint or data type check constraint
- In the definition of a generated expression column
- In a refresh-immediate materialized query table (MQT)

If the data type of the global variable is a cursor type, then the underlying cursor of the global cursor variable can be referenced anywhere that a *cursor-variable-name* can be specified.

If the data type of the global variable is a row type, a field of the global row variable can be referenced anywhere that a global variable with the same type as the field can be referenced. The global variable name that qualifies the field name is resolved in the same way as any other global variable name.

Assignment

The value of a global variable can be changed if both of the following conditions are true:

- The global variable is not a read-only variable.
- The authorization ID of the statement is authorized to write to the global variable.

A value can be assigned to a global variable using any of the following SQL statements:

- A SET variable statement with a global variable as the target variable

- An EXECUTE, FETCH, SELECT INTO, or VALUE INTO statement with a global variable as an assignment target in the INTO clause
- A CALL statement with a global variable as an argument for an OUT or INOUT parameter of the procedure
- A function invocation with a global variable as an argument for an OUT or INOUT parameter of the function (this is supported only for the source expression of a SET variable statement).

Retrieval

The value of a global variable is obtained by referencing the variable from within the SQL context where the value is needed.

The following table shows when the value of a global variable is read, for the indicated reference of that global variable.

Table 22. When the value of a global variable is read, based on the reference context

Context of the global variable reference	The reference uses the value of the global variable at the beginning of:
An SQL statement in a compound SQL (inlined) statement	The compound SQL (inlined) statement
An SQL statement in a compound SQL (compiled) statement	The SQL statement within the compound SQL (compiled) statement
An SQL statement, possibly with a function invocation or a trigger activation[1]	The SQL statement
An SQL statement in an invoked inlined SQL function	The SQL statement invoking the inlined SQL function
An SQL statement in an activated inlined trigger	The SQL statement activating the inlined trigger
An SQL statement in an invoked inlined SQL method	The SQL statement invoking the inlined SQL method
An SQL statement in an invoked compiled SQL function	The SQL statement in the compiled SQL function
An SQL statement in an activated compiled trigger	The SQL statement in the compiled trigger
An SQL statement in an invoked external routine	The SQL statement in the external routine
Note: [1] In this table, the SQL statement that might call a function or activate a trigger does not include the compound SQL (inlined) statement or the compound SQL (compiled) statement.	

Functions

A *function* is an operation denoted by a function name followed by one or more operands that are enclosed in parentheses.

A function represents a relationship between a set of input values and a set of result values. The input values to a function are called *arguments*. For example, the TIMESTAMP function can be passed arguments of type DATE and TIME, and the result is a TIMESTAMP.

There are several ways to classify functions.

One way is to classify functions as ether built-in or user-defined.

- *Built-in functions* are functions provided with the database manager. Built-in functions include aggregate functions (for example, AVG), operator functions (for example, +), casting functions (for example, DECIMAL), scalar functions (for example, CEILING), and table functions (for example, BASE_TABLE). Built-in functions are generally defined in schemas that begin with 'SYS' (for example, SYSIBM, SYSFUN, and SYSIBMADM) although some are also defined in schemas that begin with 'DB2' (for example DB2MQ).

- *User-defined functions* are functions that are created using an SQL data definition statement and registered to the database manager in the catalog. *User-defined schema functions* are created using the CREATE FUNCTION statement.*User-defined module functions* are created using the ALTER MODULE ADD FUNCTION or ALTER MODULE PUBLISH FUNCTION statements. A set of *user-defined module functions* is provided with the database manager in a set of modules in a schema called SYSIBMADM. A *user-defined function* resides in the schema in which it was created or in the module where it was added or published.

  User-defined functions extend the capabilities of the database system by adding function definitions (provided by users or third party vendors) that can be applied in the database engine itself. Extending database functions lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database.

Another way to classify a user-defined function is as an external function, an SQL function, 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 aggregate functions. An *SQL function* is defined to the database using only SQL statements, including at least one RETURN statement. It can return a scalar value, a row, or a table. SQL functions cannot be aggregate functions. A *sourced function* is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or aggregate functions. They are useful for supporting existing functions with user-defined types.

Another way to classify functions is as scalar, aggregate, row, or table function, depending on the input data values and result values. A *scalar function* is a function that returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

An *aggregate function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. An example of an aggregate function

is the built-in function AVG(). An external column UDF cannot be defined, but a column UDF, which is sourced upon one of the built-in aggregate functions, can be defined. This is useful for distinct types. For example, if there is a distinct type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE), which is sourced on the built-in function AVG(INTEGER), could be defined, and it would be an aggregate function.

A *row function* is a function that returns one row of values. It can be used in a context where a row expression is supported. It can also be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.
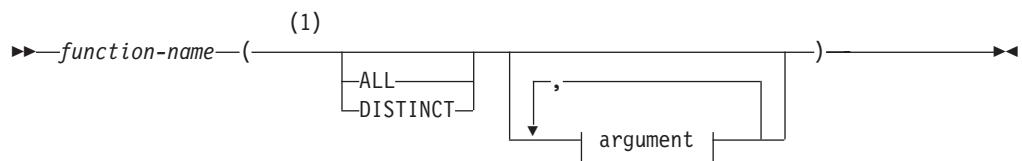
A *table function* is a function that returns a table to the SQL statement which references it. It may only be referenced in the FROM clause of a SELECT statement. Such a function can be used to apply SQL language processing power to data that does not reside in the database, or to convert such data into a table in the database. A table function can read a file, get data from the Web, or access a Lotus Notes® database and return a result table. This information can be joined with other tables in the database. A table function can be defined as an external function or as an SQL function. (A table function cannot be a sourced function.)

Function signatures
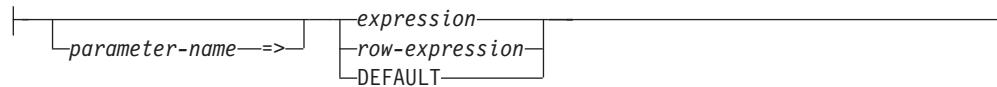
A schema function is identified by its schema name, a function name, the number of parameters, and the data types of its parameters. A module function is identified by its schema name, module name, a function name, the number of parameters, and the data types of its parameters. This identification of a schema function or a module function is called a *function signature*, which must be unique within the database; for example, TEST.RISK(INTEGER). There can be more than one function with the same name in a schema or a module, provided that the number of parameters or the data types of the parameters are different. A function name for which there are multiple function instances with the same number of parameters is called an *overloaded* function. A function name can be overloaded within a schema, in which case there is more than one function by that name with the same number of parameters in the schema. Similarly, a function name can be overloaded within a module, in which case there is more than one function by that name with the same number of parameters in the module. These functions must have different parameter data types. Functions can also be overloaded across the schemas of an SQL path, in which case there is more than one function by that name with the same number of parameters in different schemas of the SQL path. These functions do not necessarily have different parameter data types.

Function invocation

Each reference to a function conforms to the following syntax:

argument:

```
├──┬─────────────────────────┬──┬─expression──────────┬──────────────────┤
   └─parameter-name──=>──┘  ├─row-expression──┤
                               └─DEFAULT─────────┘
```

Notes:

1   The ALL or DISTINCT keyword can be specified only for an aggregate
     function or a user-defined function that is sourced on an aggregate function.

In the syntax shown previously, *expression* and *row-expression* cannot include an
aggregate function. See "Expressions" for other rules for *expression*.

A function is invoked by referring (in an allowable context) to its qualified or
unqualified function name followed by the list of arguments enclosed in
parentheses. The possible qualifiers for a function name are:
- A schema name
- An unqualified module name
- A schema-qualified module name

The qualifier used when invoking a function determines the scope used to search
for a matching function.
- If a schema-qualified module name is used as the qualifier, the scope is the
  specified module.
- If a single identifier is used as the qualifier, the scope includes:
  - The schema that matches the qualifier
  - One of the following modules:
    - The invoking module, if the invoking module name matches the qualifier
    - The first module in a schema in the SQL path that matches the qualifier
- If no qualifier is used, the scope includes the schemas in the SQL path and, if
  the function is invoked from within a module object, the same module from
  which the function is invoked.

For static SQL statements, the SQL path is specified using the **FUNCPATH** bind
option. For dynamic SQL statements, the SQL path is the value of the CURRENT
PATH special register.

When any function is invoked, the database manager must determine which
function to execute. This process is called *function resolution* and applies to both
built-in and user-defined functions. It is recommended that function invocations
intending to invoke a user-defined function be fully qualified. This improves
performance of function resolution and prevents unexpected function resolution
results as new functions are added or privileges granted.

An *argument* is a value passed to a function upon invocation or the specification of
DEFAULT. When a function is invoked in SQL, it is passed a list of zero or more
arguments. They are positional in that the semantics of an argument are
determined by its position in the argument list. A *parameter* is a formal definition
of an input to a function or an output from a function. When a function is defined
to the database, either internally (a built-in function) or by a user (a user-defined
function), its parameters (zero or more) are specified, and the order of their
definitions defines their positions and their semantics. Therefore, every parameter
is a particular positional input to a function or an output from a function. On
invocation, an argument is assigned to a parameter using either the positional
syntax or the named syntax. If using the positional syntax, an argument
corresponds to a particular parameter according to its position in the list of

arguments. If using the named syntax, an argument corresponds to a particular parameter by the name of the parameter. When an argument is assigned to a parameter using the named syntax, then all the arguments that follow it must also be assigned using the named syntax (SQLSTATE 4274K). The name of a named argument can appear only once in a function invocation (SQLSTATE 4274K). In cases where the data types of the arguments of the function invocation are not a match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution time using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter. In cases where the arguments of the function invocation are the specification of DEFAULT, the actual value used for the argument is the value specified as the default for the corresponding parameter in the function definition. If no default value was defined for the parameter, the null value is used. If an untyped expression (a parameter marker, a NULL keyword, or a DEFAULT keyword) is used as the argument, the data type associated with the argument is determined by the parameter data type of the parameter of the selected function.

Access to schema functions is controlled through the EXECUTE privilege on the schema functions. If the authorization ID of the statement invoking the function does not have EXECUTE privilege, the schema function will not be considered by the function resolution algorithm, even if it is a better match. Built-in functions in the SYSIBM and SYSFUN schemas have the EXECUTE privilege implicitly granted to PUBLIC.

Access to module functions is controlled through EXECUTE privilege on the module for all functions within the module. The authorization ID of the statement invoking the function might not have EXECUTE privilege on a module. In such cases, module functions within that module, unlike schema functions, are still considered by the function resolution algorithm even though they cannot be executed.

When the user-defined function is invoked, the value of each of its arguments is assigned, using storage assignment, to the corresponding parameter of the function. Control is passed to external functions according to the calling conventions of the host language. When execution of a user-defined scalar function or a user-defined aggregate function is complete, the result of the function is assigned, using storage assignment, to the result data type. For details on the assignment rules, see "Assignments and comparisons".

Table functions can be referenced only in the FROM clause of a subselect. For more information on referencing a table function, see "table-reference" on page 792.

Function resolution

After a function is invoked, the database manager must determine which function to execute. This process is called function resolution and applies for both built-in and user-defined functions.

The database manager first determines the set of candidate functions based on the following information:
- The qualification of the name of the invoked function
- The context that invokes the function
- The unqualified name of the invoked function
- The number of arguments specified

- Any argument names that are specified
- The authorization of schema functions.

See "Determining the set of candidate functions" for details.

The database manager then determines the best fit from the set of candidate functions based on the data types of the arguments of the invoked function as compared with the data types of the parameters of the functions in the set of candidate functions. The SQL path and number of parameters is also considered. See "Determining the best fit" on page 219 for details.

Once a function is selected, it is still possible for an error to be returned for one of the following reasons:

- If a module function is selected and either the function is invoked from outside a module or the function is invoked from within a module object and the qualifier does not match the context module name, the authorization ID of the statement that invoked the function must have EXECUTE privilege on the module that contains the selected function (SQLSTATE 42501).
- If a function is selected, its successful use depends on it being invoked in a context in which the returned result is allowed. For example, if the function returns a table where a table is not allowed, an error is returned (SQLSTATE 42887).
- If a cast function is selected, either built-in or user-defined, and any argument would need to be implicitly cast (not promoted) to the data type of the parameter, an error is returned (SQLSTATE 42884).
- If a function invocation involves an argument with an unnamed row type, an error is returned (SQLSTATE 42884) if either of the following conditions occur:
  - The number of fields of the argument does not match the number of fields of the parameter.
  - The data types of the fields of the argument are not assignable to the corresponding data type of the fields of the parameter.

Determining the set of candidate functions

- Let A be the number of arguments in a function invocation.
- Let P be the number of parameters in a function signature.
- Let N be the number of parameters in a function signature without a defined default.

Candidate functions for resolution of a function invocation are selected based on the following criteria:

- Each candidate function has a matching name and applicable number of parameters. An applicable number of parameters satisfies the condition $N \leq A \leq P$.
- Each candidate function has parameters such that for each named argument in the function invocation there exists a parameter with a matching name that does not already correspond to a positional (unnamed) argument.
- Each parameter of a candidate function that does not have a corresponding argument in the function invocation, specified by either position or name, is defined with a default.
- Each candidate function from a set of one or more schemas has the EXECUTE privilege associated with the authorization ID of the statement invoking the function.

- Each candidate function from a module other than the context module is a published module function.

The functions selected for the set of candidate functions are from one or more of the following search spaces.

1. The *context module*, that is, the module which contains the module object that invoked the function
2. A set of one or more schemas
3. A module other than the context module

The specific search spaces considered are affected by the qualification of the name of the invoked function.

- *Qualified function invocation*: When a function is invoked with a function name and a qualifier, the database manager uses the qualifier and, in some cases, the context of the invoked function to determine the set of candidate functions.

   1. If a function is invoked from within a module object using a function name with a qualifier, the database manager considers if the qualifier matches the context module name. If the qualifier is a single identifier, then the schema name of the module is ignored when determining a match. If the qualifier is a two-part identifier, then it is compared to the schema-qualified module name when determining a match. If the qualifier matches the context module name, the database manager searches the context module for candidate functions.

      If one or more candidate functions are found in the context module, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see "Determining the best fit"). Otherwise, continue to the next search space.

   2. If the qualifier is a single identifier, the database manager considers the qualifier as a schema name and searches that schema for candidate functions.

      If one or more candidate functions are found in the schema, then this set of candidate functions is processed for best fit without consideration of possible candidate functions in any other search space (see "Determining the best fit"). Otherwise, continue to the next search space, if applicable.

   3. If the function is invoked from outside a module or the qualifier does not match the context module name when it is invoked from within a module object, the database manager considers the qualifier as a module name. Without considering EXECUTE privilege on modules, the database manager then selects the first module that matches based on the following criteria:

      - If the module name is qualified with a schema name, select the module with that schema name and module name.

      - If the module name is not qualified with a schema name, select the module with that module name that is found in the schema earliest in the SQL path.

      - If the module is not found using the SQL path, select the module public alias with that module name.

      If a matching module is not found, then there are no candidate functions. If a matching module is found, the database manager searches the selected module for candidate functions.

      If one or more candidate functions are found in the selected modules, then this set of candidate functions is processed for best fit (see "Determining the best fit").

- *Unqualified function invocation*: When a function is invoked without a qualifier, the database manager considers the context of the invoked function to determine the sets of candidate functions.

  1. If a function is invoked with an unqualified function name from within a module object, the database manager searches the context module for candidate functions.

     If one or more candidate functions are found in the context module, then these candidate functions are included with any candidate functions from the schemas in the SQL path (see next item).

  2. If a function is invoked with an unqualified function name, either from within a module object or from outside a module, the database manager searches the list of schemas in the SQL path to resolve the function instance to execute. For each schema in the SQL path (see "SQL path"), the database manager searches the schema for candidate functions.

     If one or more candidate functions are found in the schemas in the SQL path, then these candidate functions are included with any candidate functions from the context module (see previous item). This set of candidate functions is processed for best fit (see "Determining the best fit").

If the database manager does not find any candidate functions, an error is returned (SQLSTATE 42884).

Determining the best fit

The set of candidate functions may contain one function or more than one function with the same name. In either case, the data types of the parameters, the position of the schema in the SQL path, and the total number of parameters of each function in the set of candidate functions are used to determine if the function meets the best fit requirements.

If the set of candidate functions contains more than one function and named arguments are used in the function invocation, the ordinal position of the parameter corresponding to a named argument must be the same for all candidate functions (SQLSTATE 4274K).

The term *set of parameters* is used to refer to all of the parameters at the same position in the parameter lists (where such a parameter exists) for the set of candidate functions. The corresponding argument of a parameter is determined based on how the arguments are specified in the function invocation. For positional arguments, the corresponding argument to a parameter is the argument in the same position in the function invocation as the position of the parameter in the parameter list of the candidate function. For named arguments, the corresponding argument to a parameter is the argument with the same name as the parameter. In this case, the order of the arguments in the function invocation is not considered while determining the best fit. If the number of parameters in a candidate function is greater than the number of arguments in the function invocation, each parameter that does not have a corresponding argument is processed as if it does have a corresponding argument that has the DEFAULT keyword as the value.

The following steps are used to determine the function that is the best fit:

Step 1: Considering arguments that are typed expressions

> The database manager determines the function, or set of functions, that

meet the best fit requirements for the invocation by comparing the data type of each parameter with the data type of the corresponding argument.

When determining whether the data type of a parameter is the same as the data type of its corresponding argument:

- Synonyms of data types match. For example, FLOAT and DOUBLE are considered to be the same.
- Attributes of a data type such as length, precision, scale, and code page are ignored. Therefore, CHAR(8) and CHAR(35) are considered to be the same, as are DECIMAL(11,2), and DECIMAL(4,3).

A subset of the candidate functions is obtained by considering only those functions for which the data type of each argument of the function invocation that is not an untyped expression matches or is promotable to the data type of the corresponding parameter of the function instance. If the argument of the function invocation is an untyped expression, the data type of the corresponding parameter can be any data type. The precedence list for the promotion of data types in "Promotion of data types" on page 115 shows the data types that fit (considering promotion) for each data type in best-to-worst order. If this subset is not empty, then the best fit is determined using the Promotable process on this subset of candidate functions. If this subset is empty, then the best fit is determined using the Castable process on the original set of candidate functions.

Promotable process

This process determines the best fit only considering whether arguments in the function invocation match or can be promoted to the data type of the corresponding parameter of the function definition. For the subset of candidate functions, the parameter lists are processed from left to right, processing the set of parameters in the first position from the subset of candidate functions before moving on to the set of parameters in the second position, and so on. The following steps are used to eliminate candidate functions from the subset of candidate functions (only considering promotion):

1. If one candidate function has a parameter where the data type of the corresponding argument fits (only considering promotion) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The precedence list for the promotion of data types in "Promotion of data types" on page 115 shows the data types that fit (considering promotion) for each data type in best-to-worst order.

2. If the data type of the corresponding argument is an untyped expression, no candidate functions are eliminated.

3. These steps are repeated for the next set of parameters from the remaining candidate functions until there are no more sets of parameters.

Castable process

This process determines the best fit first considering, for each parameter, if the data type of the corresponding argument in the function invocation matches or can be promoted to the data type of the parameter of the function definition. Then, for each set of parameters where no corresponding argument has a data type that was promotable, the database manager considers, for each

parameter, if the data type of the corresponding argument can be implicitly cast for function resolution to the data type of the parameter.

For the set of candidate functions, the parameters in the parameter lists are processed from left to right, processing the set of parameters in the first position from all the candidate functions before moving on to the set of parameters in the second position, and so on. The following steps are used to eliminate candidate functions from the set of candidate functions (only considering promotion):

1. If one candidate function has a parameter where the data type of the corresponding argument fits (only considering promotion) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The precedence list for the promotion of data types in "Promotion of data types" on page 115 shows the data types that fit (considering promotion) for each data type in best-to-worst order.

2. If the data type for the corresponding argument is not promotable (which includes the case when the corresponding argument is an untyped expression) to the data type of the parameter of any candidate function, no candidate functions are eliminated.

3. These steps are repeated for the next set of parameters from the remaining candidate functions until there are no more sets of parameters.

If at least one set of parameters has no corresponding argument that fit (only considering promotion) and the corresponding argument for the set of parameters has a data type, the database manager compares each such set of parameters from left to right. The following steps are used to eliminate candidate functions from the set of candidate functions (considering implicit casting).

1. If all the data types of the set of parameters for all remaining candidate functions do not belong to the same data type precedence list, as specified in "Promotion of data types" on page 115, an error is returned (SQLSTATE 428F5).

2. If the data type of the corresponding arguments cannot be implicitly cast to the data type of the parameters, as specified in Implicit casting for function resolution, an error is returned (SQLSTATE 42884).

3. If one candidate function has a parameter where the data type of the corresponding argument fits (considering implicit casting) the data type of the parameter better than other candidate functions, those candidate functions that do not fit the function invocation equally well are eliminated. The data type list in Implicit casting for function resolution shows the data type that fits (considering implicit casting) better.

4. These steps are repeated for the next set of parameters which has no corresponding argument that fit (only considering promotion) and the corresponding argument for the set of parameters has a data type until there are no more such sets of parameters or an error occurs.

Step 2: Considering SQL path
If more than one candidate function remains and a context module exists that still includes candidate functions, the database manager selects those functions. If there is no context module or no candidate functions remain in the context module, the database manager selects those candidate functions whose schema is earliest in the SQL path.

Step 3: Considering number of arguments in the function invocation
If more than one candidate function remains and if one candidate function has a number of parameters that is less than or equal to the number of parameters of the other candidate functions, those candidate functions that have a greater number of parameters are eliminated.

Step 4: Considering arguments that are untyped expressions
If more than one candidate function remains and at least one set of parameters has a corresponding argument that is an untyped expression, the database manager compares each such set of parameters from left to right. The following steps are used to eliminate candidate functions from the set of candidate functions:

1.   If all the data types of the set of parameters for all remaining candidate functions do not belong to the same data type precedence list, as specified in "Promotion of data types" on page 115, an error is returned (SQLSTATE 428F5).

2. If the data type of the parameter of one candidate function is further left in the data type ordering for implicit casting than other candidate functions, those candidate functions where the data type of the parameter is further right in the data type ordering are eliminated. The data type list in "Implicit casting for function resolution" shows the data type ordering for implicit casting.

If there are still multiple candidate functions, an error is returned (SQLSTATE 428F5).

Implicit casting for function resolution
Implicit casting for function resolution is not supported for arguments with a user-defined type, reference type, or XML data type. It is also not supported for built-in or user-defined cast functions. It is supported for the following cases:

- A value of one data type can be cast to any other data type that is in the same data type precedence list, as specified in "Promotion of data types" on page 115.

- A numeric or datetime data type can be cast to a character or graphic string data type, except for LOBs

- A character or graphic string type, except LOBs, can be cast to a numeric or datetime data type

- A character FOR BIT DATA can be cast to a BLOB and a BLOB can be cast to a character FOR BIT DATA

- A TIMESTAMP data type can be cast to a TIME data type

- An untyped argument can be cast to any data type.

Similar to the data type precedence list for promotion, for implicit casting there is an order to the data types that are in the group of related data types. This order is used when performing function resolution that considers implicit casting. Table 23 on page 223 shows the data type ordering for implicit casting for function resolution. The data types are listed in best-to-worst order (note that this is different than the ordering in

the data type precedence list for promotion). When function resolution selects a built-in function from the SYSIBM schema and implicit casting is necessary for some argument, if the built-in function supports both character input and graphic input for the parameter, the argument is implicitly cast to character.

Table 23. Data type ordering for implicit casting for function resolution

Data type group	Data type list for implicit casting for function resolution (in best-to-worst order)
Numeric data types	DECFLOAT, double, real, decimal, BIGINT, INTEGER, SMALLINT
Character and graphic string data types	VARCHAR or VARGRAPHIC, CHAR or GRAPHIC, CLOB or DBCLOB
Datetime data types	TIMESTAMP, DATE

Notes:

1. The lowercase types in the previous table are defined as follows:
   - decimal = DECIMAL (p,s) or NUMERIC(p,s)
   - real = REAL or FLOAT(n) where n is not greater than 24
   - double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(n), where n is greater than 24

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

2. For a Unicode database only, the following are considered to be equivalent data types:
   - CHAR or GRAPHIC
   - VARCHAR and VARGRAPHIC
   - CLOB and DBCLOB

Table 24. Derived length of an argument when invoking a built-in scalar function from the SYSIBM schema in cases where implicit casting is needed

	Target Type and Length								
Source Data Type	Char	Graphic	Varchar	Vargraphic	Clob	DBclob	Blob	Timestamp	Decfloat
UNTYPED	127	127	254	254	32767	32767	32767	12	34
SMALLINT	6	6	6	6	-	-	-	-	-
INTEGER	11	11	11	11	-	-	-	-	-
BIGINT	20	20	20	20	-	-	-	-	-
DECIMAL(p,s)	2+p	2+p	2+p	2+p	-	-	-	-	-
REAL	24	24	24	24	-	-	-	-	-
DOUBLE	24	24	24	24	-	-	-	-	-
DECFLOAT	42	42	42	42	-	-	-	-	-
CHAR(n)	-	-	-	-	-	-	min(n,254)	12	34
VARCHAR(n)	min(n,254)	min(n,127)	-	-	-	-	min(n,32672)	12	34
CLOB(n)	min(n,254)	min(n,127)	min(n,32672)	min(n,16336)	-	-	-	-	-
GRAPHIC(n)	-	-	-	-	-	-	-	12	34
VARGRAPHIC(n)	min(n,254)	min(n,127)	-	-	-	-	-	12	34
DBCLOB(n)	min(n,254)	min(n,127)	min(n,32672)	min(n,16336)	-	-	-	-	-
BLOB(n)	min(n,254)	-	min(n,32672)	-	-	-	-	-	-
TIME	8	8	8	8	-	-	-	-	-
DATE	10	10	10	10	-	-	-	-	-

Table 24. Derived length of an argument when invoking a built-in scalar function from the SYSIBM schema in cases where implicit casting is needed (continued)

Source Data Type	Target Type and Length								
	Char	**Graphic**	**Varchar**	**Vargraphic**	**Clob**	**DBclob**	**Blob**	**Timestamp**	**Decfloat**
TIMESTAMP(*p*)	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	if *p*=0 then 19 else *p*+20	-	-	-	-	-

Note

This table shows character string and graphic string data types in string units associated with a Unicode database environment where the string units default is SYSTEM. If the Unicode database environment has the string units set to CODEUNITS32, then any character string or graphic string length attributes that represent the data type maximum length should be considered to represent the data type maximum in CODEUNITS32. All character string or graphic string data types have the default string units of the database environment.

SQL path considerations for built-in functions

Function resolution applies to all functions, including schema functions and modules functions that are built-in or user-defined. If a function is invoked without its schema name, the SQL path is used to resolve the function invocation to a specific function.

The built-in functions in the SYSIBM schema are always considered during function resolution, even when SYSIBM is not explicitly included in the SQL path. Omission of SYSIBM from the path results in the assumption (for function and data type resolution) that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN","SYSIBM","SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN","SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH, because SYSIBM implicitly appears first in the path.

To minimize potential problems in this area:
- Never use the names of built-in functions for user-defined functions.
- If, for some reason, it is necessary to create a user-defined function with the same name as a built-in function, be sure to qualify any references to it.

Note: Some invocations of built-in functions do not support SYSIBM as an explicit qualifier and resolve directly to the built-in function without considering the SQL path. Specific cases are covered in the description of the built-in function.

Examples of function resolution

The following are examples of function resolution. (Note that not all required keywords are shown.)
- This is an example illustrating the SQL path considerations in function resolution. For this example, there are eight ACT functions, in three different schemas, registered as:

```
CREATE FUNCTION AUGUSTUS.ACT (CHAR(5), INT, DOUBLE) SPECIFIC ACT_1 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_2 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE, INT) SPECIFIC ACT_3 ...
CREATE FUNCTION JULIUS.ACT (INT, DOUBLE, DOUBLE) SPECIFIC ACT_4 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_5 ...
CREATE FUNCTION JULIUS.ACT (SMALLINT, INT, DOUBLE) SPECIFIC ACT_6 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_7 ...
CREATE FUNCTION NERO.ACT (INT, INT, DEC(7,2)) SPECIFIC ACT_8 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
SELECT ... ACT(I1, I2, D) ...
```

Assume that the application making this reference has an SQL path established as:

```
"JULIUS","AUGUSTUS","CAESAR"
```

Following through the algorithm...

– The function with specific name ACT_8 is eliminated as a candidate, because the schema NERO is not included in the SQL path.

– The function with specific name ACT_3 is eliminated as a candidate, because it has the wrong number of parameters. ACT_1 and ACT_6 are eliminated because, in both cases, the first argument cannot be promoted to the data type of the first parameter.

– Because there is more than one candidate remaining, the arguments are considered in order.

– For the first argument, the remaining functions, ACT_2, ACT_4, ACT_5, and ACT_7 are an exact match with the argument type. No functions can be eliminated from consideration; therefore the next argument must be examined.

– For this second argument, ACT_2, ACT_5, and ACT_7 are exact matches, but ACT_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation among ACT_2, ACT_5, and ACT_7.

– For the third and last argument, neither ACT_2, ACT_5, nor ACT_7 match the argument type exactly. Although ACT_2 and ACT_5 are equally good, ACT_7 is not as good as the other two because the type DOUBLE is closer to DECIMAL than is DECFLOAT. ACT_7 is eliminated..

– There are two functions remaining, ACT_2 and ACT_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis, ACT_5 is the function chosen.

• This is an example of a situation where function resolution will result in an error (SQLSTATE 428F5) since more than one candidate function fits the invocation equally well, but the corresponding parameters for one of the arguments do not belong to the same type precedence list.

For this example, there are only three function in a single schema defined as follows:

```
CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5))SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DATE)      SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)    SPECIFIC ACT_3 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and VC is a VARCHAR column):

```
SELECT ... ACT(I1, I2, VC) ...
```

Assume that the application making this reference has an SQL path established as:

```
"CAESAR"
```

Following through the algorithm ...

– Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:

- For the first argument, all the candidate functions have an exact match with the parameter type.

- For the second argument, ACT_1 is eliminated because INTEGER is not promotable to VARCHAR.

- For the third argument, both ACT_2 and ACT_3 are eliminated since VARCHAR is not promotable to DATE or DOUBLE, so no candidate functions remain.

– Since the subset of candidate functions is empty, the candidate functions are considered using the castable process:

- For the first argument, all the candidate functions have an exact match with the parameter type.

- For the second argument, ACT_1 is eliminated since INTEGER is not promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.

- For the third argument, the data type of the corresponding parameters of ACT_2 and ACT_3 do not belong to the same data type precedence list, so an error is returned (SQLSTATE 428F5).

• This example illustrates a situation where function resolution will succeed using the castable process. For this example, there are only three function in a single schema defined as follows:

```
CREATE FUNCTION CAESAR.ACT (INT, VARCHAR(5), VARCHAR(5))SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DECFLOAT)     SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)   SPECIFIC ACT_3 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and VC is a VARCHAR column):

```
SELECT ... ACT(I1, I2, VC) ...
```

Assume that the application making this reference has an SQL path established as:

```
"CAESAR"
```

Following through the algorithm ...

– Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:

- For the first argument, all the candidate functions have an exact match with the parameter type.

- For the second argument, ACT_1 is eliminated because INTEGER is not promotable to VARCHAR.

- For the third argument, both ACT_2 and ACT_3 are eliminated since VARCHAR is not promotable to DECFLOAT or DOUBLE, so no candidate functions remain.

– Since the subset of candidate functions is empty, the candidate functions are considered using the castable process:

- For the first argument, all the candidate functions have an exact match with the parameter type.

- For the second argument, ACT_1 is eliminated since INTEGER is not promotable to VARCHAR. ACT_2 and ACT_3 are better candidates.

- For the third argument, both DECFLOAT and DOUBLE are in the same data type precedence list and VARCHAR can be implicitly cast to both

DECFLOAT and DOUBLE. Since DECFLOAT is a better fit for the purpose of implicit casting, ACT_2 is the best fit

- This example illustrates that during function resolution using the castable process that promotion of later arguments takes precedence over implicit casting. For this example, there are only three function in a single schema defined as follows:

```
CREATE FUNCTION CAESAR.ACT (INT, INT, VARCHAR(5))SPECIFIC ACT_1 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DECFLOAT) SPECIFIC ACT_2 ...
CREATE FUNCTION CAESAR.ACT (INT, INT, DOUBLE)  SPECIFIC ACT_3 ...
```

The function reference is as follows (where I1 is an INTEGER column, and VC1 is a VARCHAR column and C1 is a CHAR column):

```
SELECT ... ACT(I1, VC1, C1) ...
```

Assume that the application making this reference has an SQL path established as:

```
"CAESAR"
```

Following through the algorithm:

- Each of the candidate functions is evaluated to determine if the data type of each input argument of the function invocation matches or is promotable to the data type of the corresponding parameter of the function instance:
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, all candidate functions are eliminated since VARCHAR is not promotable to INTEGER, so no candidate functions remain.

- Since the subset of candidate functions is empty, the candidate functions are considered using the castable process
  - For the first argument, all the candidate functions have an exact match with the parameter type.
  - For the second argument, none of the candidate functions have a parameter to which the corresponding argument can be promoted, so no candidate functions are eliminated.
  - Since the third argument can be promoted to the parameter of ACT_1, but not to the parameters of ACT_2 or ACT_3, ACT_1 is the best fit.

Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or *subject argument*) has the same type, or is a subtype of the subject type (also called *subject parameter*), of the method.

For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR, and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. *Observer methods* are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. *Mutator methods* are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described previously is an example of a mutator method for the type ADDRESS.

Explicitly defined methods, or *user-defined methods*, are methods that are registered to a database in SYSCAT.ROUTINES, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

User-defined methods for structured types extend the function of the database system by adding method definitions (provided by users or third party vendors) that can be applied to structured type instances in the database engine. Defining database methods lets the database exploit the same methods in the engine that an application uses, providing more synergy between application and database.

External and SQL user-defined methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, because they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as *type preserving* (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method signatures

A method is identified by its subject type, a method name, the number of parameters, and the data types of its parameters. This is called a *method signature*, and it must be unique within the database.

There can be more than one method with the same name for a structured type, provided that:

- The number of parameters or the data types of the parameters are different, or
- The methods are part of the same method hierarchy (that is, the methods are in an overriding relationship or override the same original method), or
- The same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name that has multiple method instances is called an *overloaded method*. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy. These methods must have different parameter types.

A method can be invoked by referring (in an allowable context) to the method name, preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. A list of arguments enclosed in parentheses must follow. Which method is actually invoked depends on the static type of the subject type, using the method resolution process described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution apply.

If function resolution results in a method defined WITH FUNCTION ACCESS, all subsequent steps of method invocation are processed.

Access to methods is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific method or a set of methods. The EXECUTE privilege (or DATAACCESS authority) is needed to invoke a method. The definer of the method automatically receives the EXECUTE privilege. The definer of an external method or an SQL method having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the method. The definer (or authorization ID with the ACCESSCTRL or SECADM authority) must then grant it to the user who wants to invoke the method from any SQL statement, or reference the method in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint). If the EXECUTE privilege is not granted to a user, the method will not be considered by the method resolution algorithm, even if it is a better match.

Method resolution

After method invocation, the database manager must decide which of the possible methods with the same name is the "best fit". Functions (built-in or user-defined) are not considered during method resolution.

An *argument* is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a method. When a method is defined to the database, either implicitly (system-generated for a type) or by a user (a user-defined method), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a

method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the method given in the invocation, EXECUTE privilege on the method, the number and data types of the arguments, all the methods with the same name for the subject argument's static type (and it's supertypes), and the data types of their corresponding parameters as the basis for deciding whether or not to select a method. The following are the possible outcomes of the decision process:

- A particular method is deemed to be the best fit. For example, given the methods named RISK for the type SITE with signatures defined as:

```
PROXIMITY(INTEGER) FOR SITE
PROXIMITY(DOUBLE) FOR SITE
```

the following method invocation (where ST is a SITE column, DB is a DOUBLE column):

```
SELECT ST..PROXIMITY(DB) ...
```

then, the second PROXIMITY will be chosen.

The following method invocation (where SI is a SMALLINT column):

```
SELECT ST..PROXIMITY(SI) ...
```

would choose the first PROXIMITY, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE, which is further down the precedence list.

When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.

- No method is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

```
SELECT ST..PROXIMITY(C) ...
```

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

- A particular method is selected based on the methods in the type hierarchy and the number and data types of the arguments passed on invocation. For example, given methods named RISK for the types SITE and DRILLSITE (a subtype of SITE) with signatures defined as:

```
RISK(INTEGER) FOR DRILLSITE
RISK(DOUBLE) FOR SITE
```

and the following method invocation (where DRST is a DRILLSITE column, DB is a DOUBLE column):

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

the second RISK will be chosen, because DRILLSITE can be promoted to SITE.

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

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

would choose the first RISK, because SMALLINT can be promoted to INTEGER, which is closer on the precedence list than DOUBLE, and DRILLSITE is a better match than SITE, which is a supertype.

Methods within the same type hierarchy cannot have the same signatures, considering parameters other than the subject parameter.

Determining the best fit

A comparison of the data types of the arguments with the defined data types of the parameters of the methods under consideration forms the basis for the decision of which method in a group of like-named methods is the "best fit". Note that the data types of the results of the methods under consideration do not enter into this determination.

For method resolution, whether the data type of the input arguments can be promoted to the data type of the corresponding parameter is considered when determining the best fit. Unlike function resolution, whether the input arguments can be implicitly cast to the data type of the corresponding parameter is not considered when determining the best fit. Modules are not considered during method resolution because methods cannot be defined in modules.

Method resolution is performed using the following steps:
1. First, find all methods from the catalog (SYSCAT.ROUTINES) such that all of the following are true:
   - The method name matches the invocation name, and the subject parameter is the same type or is a supertype of the static type of the subject argument.
   - The invoker has the EXECUTE privilege on the method.
   - The number of defined parameters matches the invocation.
   - Each invocation argument matches the method's corresponding defined parameter in data type, or is "promotable" to it.
2. Next, consider each argument of the method invocation, from left to right. The leftmost argument (and thus the first argument) is the implicit SELF parameter. For example, a method defined for type ADDRESS_T has an implicit first parameter of type ADDRESS_T. For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list corresponding to the argument data type for which there exists a function with a parameter of that data type. Length, precision, scale, and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, a DECFLOAT(34) argument is considered an exact match for a DECFLOAT(16) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

   The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).
3. At most, one candidate method remains after Step 2. This is the method that is chosen.
4. If there are no candidate methods remaining after step 2, an error is returned (SQLSTATE 42884).

Example of method resolution

Following is an example of successful method resolution.

There are seven FOO methods for three structured types defined in a hierarchy of GOVERNOR as a subtype of EMPEROR as a subtype of HEADOFSTATE, registered with the following signatures:

```
CREATE METHOD FOO (CHAR(5), INT, DOUBLE)   FOR HEADOFSTATE SPECIFIC FOO_1 ...
CREATE METHOD FOO (INT, INT, DOUBLE)       FOR HEADOFSTATE SPECIFIC FOO_2 ...
CREATE METHOD FOO (INT, INT, DOUBLE, INT) FOR HEADOFSTATE SPECIFIC FOO_3 ...
CREATE METHOD FOO (INT, DOUBLE, DOUBLE)    FOR EMPEROR     SPECIFIC FOO_4 ...
CREATE METHOD FOO (INT, INT, DOUBLE)       FOR EMPEROR     SPECIFIC FOO_5 ...
CREATE METHOD FOO (SMALLINT, INT, DOUBLE) FOR EMPEROR     SPECIFIC FOO_6 ...
CREATE METHOD FOO (INT, INT, DEC(7,2))     FOR GOVERNOR    SPECIFIC FOO_7 ...
```

The method reference is as follows (where I1 and I2 are INTEGER columns, D is a DECIMAL column and E is an EMPEROR column):

```
SELECT E..FOO(I1, I2, D) ...
```

Following through the algorithm...

- FOO_7 is eliminated as a candidate, because the type GOVERNOR is a subtype (not a supertype) of EMPEROR.
- FOO_3 is eliminated as a candidate, because it has the wrong number of parameters.
- FOO_1 and FOO_6 are eliminated because, in both cases, the first argument (not the subject argument) cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are considered in order.
- For the subject argument, FOO_2 is a supertype, while FOO_4 and FOO_5 match the subject argument.
- For the first argument, the remaining methods, FOO_4 and FOO_5, are an exact match with the argument type. No methods can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, FOO_5 is an exact match, but FOO_4 is not, so it is eliminated from consideration. This leaves FOO_5 as the method chosen.

Method invocation

Once the method is selected, there are still possible reasons why the use of the method may not be permitted.

Each method is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the method is invoked, an error will occur. For example, assume that the following methods named STEP are defined, each with a different data type as the result:

```
STEP(SMALLINT) FOR TYPEA RETURNS CHAR(5)
STEP(DOUBLE) FOR TYPEA RETURNS INTEGER
```

and the following method reference (where S is a SMALLINT column and TA is a column of TYPEA):

```
SELECT 3 + TA..STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement, because the result type is CHAR(5) instead of a numeric type, as required for an argument of the addition operator.

Starting from the method that has been chosen, the algorithm described in "Dynamic dispatch of methods" is used to build the set of dispatchable methods at compile time. Exactly which method is invoked is described in "Dynamic dispatch of methods".

Note that when the selected method is a type preserving method:
- the static result type following function resolution is the same as the static type of the subject argument of the method invocation
- the dynamic result type when the method is invoked is the same as the dynamic type of the subject argument of the method invocation.

This may be a subtype of the result type specified in the type preserving method definition, which in turn may be a supertype of the dynamic type that is actually returned when the method is processed.

In cases where the arguments of the method invocation were not an exact match to the data types of the parameters of the selected method, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter, but excludes the case where the dynamic type of the argument is a subtype of the parameter's static type.

Dynamic dispatch of methods

Methods provide the functionality and encapsulate the data of a type. A method is defined for a type and can always be associated with this type. One of the method's parameters is the implicit SELF parameter. The SELF parameter is of the type for which the method has been declared. The argument that is passed as the SELF argument when the method is invoked in a DML statement is called *subject*.

When a method is chosen using method resolution (see "Method resolution" on page 229), or a method has been specified in a DDL statement, this method is known as the "most specific applicable authorized method". If the subject is of a structured type, that method could have one or more overriding methods. A determination is then made to select which method to invoke, based on the dynamic type (most specific type) of the subject at run time. This determination is called "determining the most specific dispatchable method". That process is described here.

1. Find the original method in the method hierarchy that the most specific applicable authorized method is part of. This is called the *root method*.
2. Create the set of dispatchable methods, which includes the following:
   - The most specific applicable authorized method.
   - Any method that overrides the most specific applicable authorized method, and which is defined for a type that is a subtype of the subject of this invocation.
3. Determine the most specific dispatchable method, as follows:
   a. Start with an arbitrary method that is an element of the set of dispatchable methods and that is a method of the dynamic type of the subject, or of one of its supertypes. This is the initial most specific dispatchable method.
   b. Iterate through the elements of the set of dispatchable methods. For each method: If the method is defined for one of the proper subtypes of the type for which the most specific dispatchable method is defined, and if it is

> defined for one of the supertypes of the most specific type of the subject, then repeat step 2 with this method as the most specific dispatchable method; otherwise, continue iterating.

4. Invoke the most specific dispatchable method.

Example:

Given are three types, "Person", "Employee", and "Manager". There is an original method "income", defined for "Person", which computes a person's income. A person is by default unemployed (a child, a retiree, and so on). Therefore, "income" for type "Person" always returns zero. For type "Employee" and for type "Manager", different algorithms have to be applied to calculate the income. Hence, the method "income" for type "Person" is overridden in "Employee" and "Manager".

Create and populate a table as follows:

```
CREATE TABLE aTable (id integer, personColumn Person);
  INSERT INTO aTable VALUES (0, Person()), (1, Employee()), (2, Manager());
```

List all persons who have a minimum income of $40000:

```
SELECT id, person, name
  FROM aTable
  WHERE person..income() >= 40000;
```

The method "income" for type "Person" is chosen, using method resolution, to be the most specific applicable authorized method.

1. The root method is "income" for "Person" itself.
2. The second step of the previous algorithm is carried out to construct the set of dispatchable methods:
   - The method "income" for type "Person" is included, because it is the most specific applicable authorized method.
   - The method "income" for type "Employee", and "income" for "Manager" is included, because both methods override the root method, and both "Employee" and "Manager" are subtypes of "Person".

   Therefore, the set of dispatchable methods is: {"income" for "Person", "income" for "Employee", "income" for "Manager"}.
3. Determine the most specific dispatchable method:
   - For a subject whose most specific type is "Person":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject, "income" for type "Person" is the most specific dispatchable method.
   - For a subject whose most specific type is "Employee":
     a. Let the initial most specific dispatchable method be "income" for type "Person".
     b. Iterate through the set of dispatchable methods. Because method "income" for type "Employee" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Employee" is a better

match for the most specific dispatchable method. Repeat this step with method "income" for type "Employee" as the most specific dispatchable method.

c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Employee" and for a supertype of the most specific type of the subject, method "income" for type "Employee" is the most specific dispatchable method.

- For a subject whose most specific type is "Manager":

a. Let the initial most specific dispatchable method be "income" for type "Person".

b. Iterate through the set of dispatchable methods. Because method "income" for type "Manager" is defined for a proper subtype of "Person" and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method "income" for type "Manager" is a better match for the most specific dispatchable method. Repeat this step with method "income" for type "Manager" as the most specific dispatchable method.

c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of "Manager" and for a supertype of the most specific type of the subject, method "income" for type "Manager" is the most specific dispatchable method.

4. Invoke the most specific dispatchable method.

Conservative binding semantics

Object resolution takes place when defining an SQL object or processing a package bind operation.

The database manager chooses which particular defined SQL object to use for an SQL object referenced in a DDL statement or coded in an application.

At a later time, the database manager might resolve to a different SQL object, even though the original SQL object did not change in any way. This resolution to a different SQL object happens as a result of defining another SQL object (or adding a privilege to an existing function) that the object resolution algorithm defines as resolved ahead of the SQL object originally chosen. Examples of SQL objects and situations to which this resolution to a different SQL object applies include the following situations:

- Routines - a new routine could be defined that is a better fit or that is an equally good fit but earlier in the SQL path; or a privilege could be granted to an existing routine that is a better fit or that is an equally good fit but earlier in the SQL path
- User-defined data types - a new user-defined data type could be defined with the same name and in a schema that is earlier in the SQL path
- Global variables - a new global variable could be defined with the same name and in a schema that is earlier in the SQL path
- Tables or views that resolve using a public alias - an actual table, view, or private alias could be defined with the same name in the current schema
- Sequences that resolve using a public sequence alias - an actual sequence or private sequence alias could be defined with the same name in the current schema
- Modules that resolve to a public module alias - an actual module or private module alias could be defined with the same name in a schema that is in the SQL path

There are instances where the database manager must be able to repeat the resolution of SQL objects as originally resolved when the statement was processed. This is true when the following static objects are used:

- Static DML statements in packages
- Views
- Triggers
- Check constraints
- SQL routines
- Global variables with a user-defined type or default expression
- Routines with a user-defined parameter type or default expression

For static DML statements in packages, SQL object references are resolved during a bind operation. SQL object references in views, triggers, SQL routines, and check constraints are resolved when the SQL object is defined or revalidated. When an existing static object is used, *conservative binding semantics* are applied unless the object has been marked invalid or inoperative by a change in the database schema.

Conservative binding semantics ensure that SQL object references will be resolved using the same SQL path, default schema, and set of routines that were used when it was previously resolved. It also ensures that the timestamp of the definition of the SQL objects considered during conservative binding resolution is not later than

the timestamp when the statement was last bound or validated using *non-conservative binding* semantics. Non-conservative binding semantics use the same SQL path and default schema as the original generation of the package or statement, but does not consider the timestamp of the definition of the SQL objects and does not consider any previously resolved set of routines.

Some changes to the database schema, such as dropping objects, altering objects, or revoking privileges, can impact an SQL object so that the database manager can no longer resolve all dependent SQL objects of an existing SQL object using conservative binding semantics.

- When this happens for a static statement in an SQL package, the package is marked inoperative. The next use of a statement in this package will cause an implicit rebind of the package using non-conservative binding semantics in order to be able to resolve to SQL objects considering the latest changes in the database schema that caused that package to become inoperative.
- When this happens for a view, trigger, check constraint, or SQL routine, the SQL object is marked invalid. The next use of the object causes an implicit revalidation of the SQL object using non-conservative binding semantics.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). Assume the SQL path contains both schemas SCHEMA1 and SCHEMA2 (although their order within the SQL path does not matter). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that invokes BAR(INT_VAL), where INT_VAL is a column or global variable defined with the INTEGER data type. This function reference in the view resolves to the function SCHEMA2.BAR(DOUBLE) because USER1 does not have the EXECUTE privilege on SCHEMA1.BAR(INTEGER). If USER1 is granted the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created, the view will continue to use SCHEMA2.BAR(DOUBLE) unless a database schema change causes the view to be marked invalid. The view is marked invalid if a required privilege is revoked or an object it depends on gets dropped or altered.

For static DML in packages, packages can rebind implicitly, or by explicitly issuing the REBIND command (or corresponding API), or the BIND command (or corresponding API). The implicit rebind is performed with conservative binding semantics if the package is marked invalid, but uses non-conservative binding semantics when the package is marked inoperative. A package is marked invalid only if an index on which it depends is dropped or altered. The REBIND command provides the option to resolve with conservative binding semantics (RESOLVE CONSERVATIVE) or to resolve by considering new routines, data types, or global variables (RESOLVE ANY, which is the default option). The RESOLVE CONSERVATIVE option can be used only if the package has not been marked inoperative by the database manager (SQLSTATE 51028).

The description of conservative binding semantics in this topic has assumed that the database configuration parameter **auto_reval** has a setting other than DISABLED. The default setting for **auto_reval** for new databases is DEFERRED. The default setting for **auto_reval** for databases upgraded to Version 9.7 is DISABLED. If **auto_reval** is set to DISABLED, then conservative binding semantics, invalidation, and revalidation behavior are the same as in releases previous to Version 9.7. Under a **auto_reval** setting of DISABLED, conservative binding semantics only considers the timestamp of the definition of the SQL objects for functions, methods, user-defined types, and global variables. For invalidation and revalidation behavior, this means, in the case of the DROP,

Conservative binding semantics

REVOKE, and ALTER statements, that either the semantics are more restrictive or the impact on dependent objects is to cascade and drop the object. In the case of packages, most database schema changes result in marking the package invalid and using conservative binding semantics during implicit rebind. However, when the schema is changed by dropping a dependent function and **auto_reval** is set to DISABLED, the package dependent on the function is marked inoperative and there is no implicit rebind of the inoperative package.

Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. When repeatedly using similar complex expressions, an SQL function to encapsulate a common expression can be considered.

Authorization
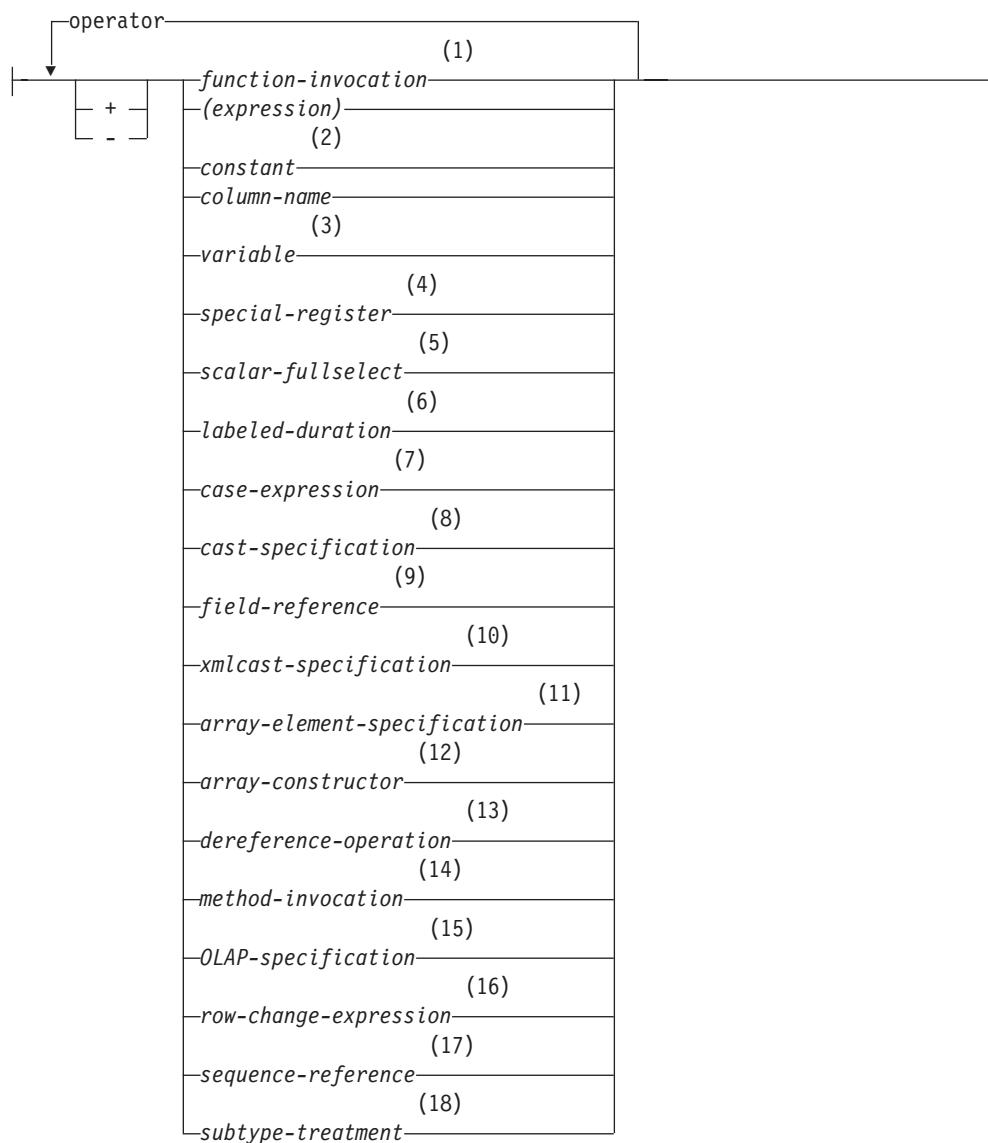
The use of some of the expressions, such as a scalar-subselect, sequence-reference, or function-invocation may require having the appropriate authorization. For these expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- scalar-subselect. For information about authorization considerations, see "SQL Queries".
- sequence-reference. The authorization to reference the sequence. For information about authorization considerations, see "Sequence authorization".
- function-invocation. The authorization to execute a user-defined function. For information about authorization considerations, see the "Function invocation" section in "Functions".
- variable. If the variable is a global variable, the authorization to reference the global variable is required. For information, see "Global variables".

In a Unicode database, an expression that accepts a character or graphic string will accept any string types for which conversion is supported.

expression:

Expressions

```
        ┌─operator─────────────────────────────────────┐
        ▼                                                            (1)
├──┬─────────┬──┬─function-invocation─────────────┐──┤
   │ ┌─ + ─┐ │  ├─(expression)────────────────────┤
   └─┤     ├─┘  │                      (2)         │
     └─ - ─┘    ├─constant────────────────────────┤
               ├─column-name─────────────────────┤
               │                      (3)         │
               ├─variable────────────────────────┤
               │                      (4)         │
               ├─special-register────────────────┤
               │                      (5)         │
               ├─scalar-fullselect───────────────┤
               │                      (6)         │
               ├─labeled-duration────────────────┤
               │                      (7)         │
               ├─case-expression─────────────────┤
               │                      (8)         │
               ├─cast-specification──────────────┤
               │                      (9)         │
               ├─field-reference─────────────────┤
               │                      (10)        │
               ├─xmlcast-specification───────────┤
               │                        (11)      │
               ├─array-element-specification─────┤
               │                      (12)        │
               ├─array-constructor───────────────┤
               │                      (13)        │
               ├─dereference-operation───────────┤
               │                      (14)        │
               ├─method-invocation───────────────┤
               │                      (15)        │
               ├─OLAP-specification──────────────┤
               │                      (16)        │
               ├─row-change-expression───────────┤
               │                      (17)        │
               ├─sequence-reference──────────────┤
               │                      (18)        │
               └─subtype-treatment───────────────┘
```

operator:

```
                 (19)
├──┬─CONCAT──┬────────────────────────────────────────────────┤
   ├─ / ────┤
   ├─ * ────┤
   ├─ + ────┤
   └─ - ────┘
```

Notes:

1 See "Function invocation" on page 214 for more information.

2 See "Constants" on page 160 for more information.

3 See "References to variables" on page 77 for more information.

4 See "Special registers" on page 165 for more information.

5 See "Scalar fullselect" on page 250 for more information.

6　　See "Durations" on page 252 for more information.

7　　See "CASE expression" on page 257 for more information.

8　　See "CAST specification" on page 260 for more information.

9　　See "Field reference" on page 265 for more information.

10　　See "XMLCAST specification" on page 266 for more information.

11　　See "ARRAY element specification" on page 268 for more information.

12　　See "Array constructor" on page 269 for more information.

13　　See "Dereference operation" on page 271 for more information.

14　　See "Method invocation" on page 273 for more information.

15　　See "OLAP specification" on page 275 for more information.

16　　See "ROW CHANGE expression" on page 284 for more information.

17　　See "Sequence reference" on page 286 for more information.

18　　See "Subtype treatment" on page 290 for more information.

19　　|| 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) combines two operands to form a *string expression*.

The first operand is an expression that returns a value of any string data type, any numeric data type, or any datetime data type. The second operand is also an expression that returns a value of any string data type, any numeric data type, or any datetime data type. However, some data types are not supported in combination with the data type of the first operand, as described in the remainder of this section.

The operands can be any combination of string (except binary string), numeric, and datetime values. When any operand is a non-string value, it is implicitly cast to VARCHAR. A binary string can only be concatenated with another binary string. However, through the castable process of function resolution, a binary string can be concatenated with a character string defined as FOR BIT DATA when the first operand is the binary string.

Concatenation involving both character string operands and graphic string operands is supported only in a Unicode database. Character operands are first converted to the graphic data type before the concatenation. Character strings defined as FOR BIT DATA cannot be cast to the graphic data type.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second. Note that no check is made for improperly formed mixed data when doing concatenation.

The length of the result is the sum of the lengths of the operands. In a Unicode database, the string unit of the result is the maximum string unit of the operands, as described in "Rules for result data types".

The data type and length attribute of the result is determined from that of the operands as shown in the following table unless an operand is defined with CODEUNITS32:

Table 25. Data Type and Length of Concatenated Operands

Operands	Combined Length Attributes[1]	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))

Table 25. Data Type and Length of Concatenated Operands (continued)

Operands	Combined Length Attributes[1]	Result
Notes [1] The numbers specified for the Combined Length Attributes are listed in OCTETS for character strings and CODEUNITS16 for graphic strings. Refer to the next table if an operand is defined with CODEUNITS32.		

If an operand is defined with CODEUNITS32, the other operand cannot be defined as FOR BIT DATA. Otherwise, when an operand is defined with CODEUNITS32, the data type and length attribute of the result is determined from that of the operands as shown in the following table:

Table 26. Data Type and Length of Concatenated Operands with CODEUNITS32

Operands	Combined Length Attributes	Result
CHAR(A) CHAR(B)	<64	CHAR(A+B)
CHAR(A) CHAR(B)	>63	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B, 8168))
VARCHAR(A) VARCHAR(B)	-	VARCHAR(MIN(A+B, 8168))
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 536870911))
CLOB(A) VARCHAR(B	-	CLOB(MIN(A+B, 536870911))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 536870911))
GRAPHIC(A) GRAPHIC(B)	<64	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>63	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B, 8168))
VARGRAPHIC(A) VARGRAPHIC(B)	-	VARGRAPHIC(MIN(A+B, 8168))
DBCLOB(A) CHAR(B)	-	DBCLOB(MIN(A+B, 536870911))
DBCLOB(A) VARCHAR(B)	-	DBCLOB(MIN(A+B, 536870911))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 536870911))

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG VARCHAR or LONG VARGRAPHIC data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands.

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as

those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

Example 1: If FIRSTNME is Pierre and LASTNAME is Fermat, then the following:
```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

returns the value `Pierre Fermat`.

Example 2: Given:
- `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 section code page. For more information about section code pages, see "Derivation of code page values".

Example 3: Given:
- `COLA` defined as CHAR(10)
- `COLB` defined as VARCHAR(5)

The parameter marker in the expression:
```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15), because `COLA CONCAT COLB` is evaluated first, giving a result that is the first operand of the second CONCAT operation.

User-defined types and the concatenation operator

A weakly typed distinct type is the only user-defined type that can be used with the concatenation operator. The source type of the weakly typed distinct type is used as the data type of the operand when processing the concatenation operator.

A strongly typed user-defined type cannot be used with the concatenation operator, even if it is a strongly typed distinct type with a source data type that is a string type. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE_DESCRIPTION, both of which had VARCHAR(25) data types, the following user-defined function, ATTACH, can be used to concatenate them.
```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator can be overloaded using a user-defined function to add the new data types.
```
CREATE FUNCTION CONCAT (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Expressions with arithmetic operators

If 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 253). For example, USER+2 is invalid. When any operand of an arithmetic operation is a weakly typed distinct type, the operation is performed assuming that the data type of the operand is the source type of the weakly typed distinct type. Sourced functions can be defined for arithmetic operations on strongly typed distinct types with a source type that is a signed numeric type.

The 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, except if the calculation is performed using decimal floating-point arithmetic. These operators can also be treated as functions. Thus, the expression "+"(a,b) is equivalent to the expression a+b. "operator" function.

Operands with a character or graphic string data type, except LOBs, are converted to DECFLOAT(34) using the rules for CAST specification, before performing the arithmetic operation. For more information, see "Casting between data types". Note that arithmetic involving graphic string operands is supported only in a Unicode database.

Operands with a string data type are converted to DECFLOAT(34) using the rules for CAST specification before performing the arithmetic operation. For more information, refer to "Casting between data types". The string must contain a valid representation of a number.

Arithmetic errors

If an arithmetic error such as divide by zero or a numeric overflow occurs during the processing of an non-decimal floating-point expression, an error is returned (SQLSTATE 22003 or 22012). For decimal floating-point expressions, a warning is returned (SQLSTATEs 0168C, 0168D, 0168E, or 0168F) 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 an aggregate function causes the row to be ignored in the determining the result of the aggregate function. If the arithmetic error was an overflow, this may significantly impact the result values.
- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows to not be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Examples:

- Add a case expression to check for zero divide and set the required value for such a situation
- Add additional predicates to handle nulls (like a check constraint on not nullable columns could become:

      check (c1*c2 is not null and c1*c2>5000)

  to cause the constraint to be violated on an overflow).

Two integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a *large integer* unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including 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 $31-p+s-s'$. 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 $31-p+s-s'$. If the parameter is set to YES, the scale is calculated as $\max(3, 31-p+s-s')$. 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 subtracted from -0
    - Operands with opposite signs are added, or operands with the same sign are subtracted; the result has a coefficient of 0, and the rounding mode is ROUND_FLOOR
    - Operands are multiplied or divided, the result has a coefficient of 0, and the signs of the operands are different
    - The first argument of the POWER function is -0, and the second argument is a positive odd number
    - The argument of the CEIL, FLOOR, or SQRT function is -0
    - The first argument of the ROUND or TRUNCATE function is -0

The following examples show special decimal floating-point values as operands:

```
INFINITY + 1          = INFINITY
INFINITY + INFINITY   = INFINITY
INFINITY + -INFINITY  = NAN          -- warning
NAN + 1               = NAN
NAN + INFINITY        = NAN
1 - INFINITY          = -INFINITY
INFINITY - INFINITY   = NAN          -- warning
-INFINITY - -INFINITY = NAN          -- warning
-0.0 - 0.0E1          = -0.0
-1.0 * 0.0E1          = -0.0
1.0E1 / 0             = INFINITY     -- warning
```

```
-1.0E5 / 0.0        = -INFINITY   -- warning
1.0E5 / -0          = -INFINITY   -- warning
INFINITY / -INFINITY = NAN        -- warning
INFINITY / 0        = INFINITY
-INFINITY / 0       = -INFINITY
-INFINITY / -0      = INFINITY
```

User-defined types as operands of arithmetic operators

Weakly typed distinct type operands can be used with arithmetic operators, provided that source type of the weakly typed distinct type is supported by the arithmetic operator. There is no need to create additional user-defined functions to support arithmetic operations for weakly typed distinct type operands.

A strongly typed user-defined type cannot be used with arithmetic operators, even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were strongly typed distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Precedence of operations

Expressions within parentheses and dereference operations are evaluated first from left to right. (Parentheses are also used in fullselects, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.) When the order of evaluation is not specified by parentheses, 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.
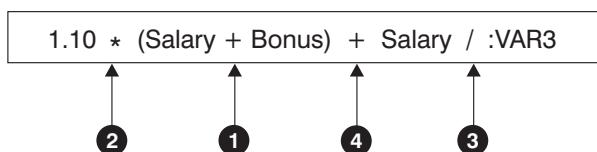
Figure 14. Precedence of Operations

Scalar fullselect

Scalar fullselect:

```
├──(─fullselect─)───────────────────────────────────────────┤
```

A *scalar fullselect* is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name

is based on the name of the column. The authorization required for a scalar
fullselect is the same as that required for an SQL query.

Datetime operations and durations
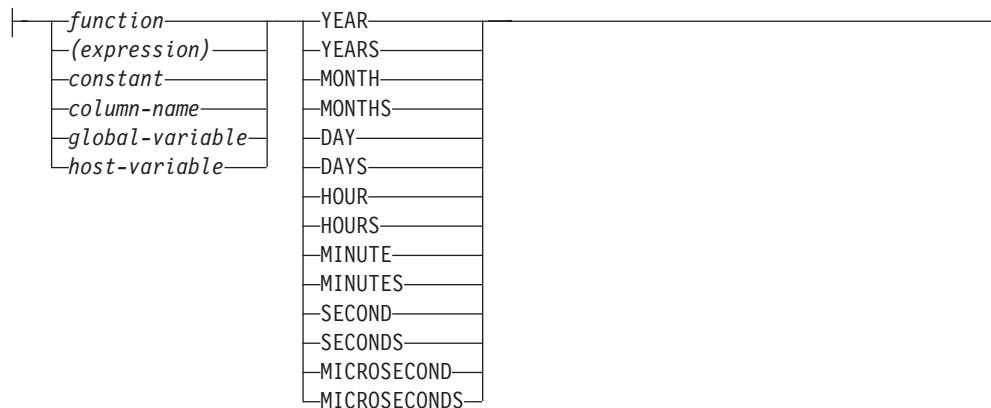
Datetime values can be incremented, decremented, and subtracted. These operations can involve decimal numbers called durations.

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:

```
├──┬─function────────┬──┬─YEAR─────────┬───────────────────────────────┤
   ├─(expression)────┤  ├─YEARS────────┤
   ├─constant────────┤  ├─MONTH────────┤
   ├─column-name─────┤  ├─MONTHS───────┤
   ├─global-variable─┤  ├─DAY──────────┤
   └─host-variable───┘  ├─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, except for SECONDS which uses DECIMAL(27,12) to allow 0 to 12 digits of fractional seconds to be included. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

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 fractional seconds, expressed as a DECIMAL(14+*s*,*s*) number, where *s* is the number of digits of fractional seconds ranging from 0 to 12. To be properly interpreted, the number must have the format *yyyymmddhhmmss.nnnnnnnnnnnn*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *nnnnnnnnnnnn* represent, respectively, the number of years, months, days, hours, minutes, seconds, and fractional seconds. The result of subtracting one timestamp value from another is a timestamp duration, with scale that matches the maximum timestamp precision of the timestamp operands.

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of YEARS, MONTHS, or DAYS.
- If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a timestamp, the second operand must be a date, a timestamp, a string representation of a date, a string representation of a timestamp, or a duration. If the second operand is a string representation of a timestamp, it is implicitly converted to a timestamp with the same precision of the first operand.
- If the second operand is a timestamp, the first operand must be a date, a timestamp, a string representation of a date, or a string representation of a timestamp. If the first operand is a string representation of a timestamp, it is implicitly converted to a timestamp with the same precision of the second operand.
- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of YEARS, MONTHS, or DAYS.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of HOURS, MINUTES, or SECONDS.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- Neither operand of the subtraction operator can be a parameter marker.

Date arithmetic

Dates can be subtracted, incremented, or decremented.

- The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = DATE1 - DATE2.

  ```
  If DAY(DATE2) <= DAY(DATE1)
  then DAY(RESULT) = DAY(DATE1) - DAY(DATE2).

  If DAY(DATE2) > DAY(DATE1)
  then DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)
  where N = the last day of MONTH(DATE2).
  MONTH(DATE2) is then incremented by 1.

  If MONTH(DATE2) <= MONTH(DATE1)
  then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).

  If MONTH(DATE2) > MONTH(DATE1)
  then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
  YEAR(DATE2) is then incremented by 1.

  YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).
  ```

  For example, the result of DATE('3/15/2000') - '12/31/1999' is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

- The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

  If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

  Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

  Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

  Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

  When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

  ```
  DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.
  ```

  When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 - X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
     DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.
```

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time arithmetic

Times can be subtracted, incremented, or decremented.

- The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

  If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

  If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TIME1 - TIME2.

  ```
  If SECOND(TIME2) <= SECOND(TIME1)
  then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).

  If SECOND(TIME2) > SECOND(TIME1)
  then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
  MINUTE(TIME2) is then incremented by 1.

  If MINUTE(TIME2) <= MINUTE(TIME1)
  then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).

  If MINUTE(TIME1) > MINUTE(TIME1)
  then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
  HOUR(TIME2) is then incremented by 1.

  HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
  ```

  For example, the result of TIME('11:02:26') - '00:32:56' is 102930. (a duration of 10 hours, 29 minutes, and 30 seconds).

- The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

  Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

  Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

  Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. TIME1 + X, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

  ```
  TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
  ```

  When subtracting a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the time value has up to 12 fractional second digits but the result is returned with the fractional seconds truncated.

> **Note:** Although the time '24:00:00' is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (for example, time('24:00:00')±0 seconds = '00:00:00').

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

- The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and fractional seconds between the two timestamps. The data type of the result is DECIMAL(14+s,s), where s is the maximum timestamp precision of TS1 and TS2.

  If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TS1 - TS2:

  ```
  If SECOND(TS2,s) <= SECOND(TS1,s)
  then SECOND(RESULT,s) = SECOND(TS1,s) -
  SECOND(TS2,s).

  If SECOND(TS2,s) > SECOND(TS1,s)
  then SECOND(RESULT,s) = 60 +
  SECOND(TS1,s) - SECOND(TS2,s).
  MINUTE(TS2) is then incremented by 1.
  ```

  The minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

  ```
  If HOUR(TS2) <= HOUR(TS1)
  then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).

  If HOUR(TS2) > HOUR(TS1)
  then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
  and DAY(TS2) is incremented by 1.
  ```

  The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

- The result of subtracting a date (D1) from a timestamp (TS1) is the same as subtracting TIMESTAMP(D1) from TS1. Similarly, the result of subtracting one timestamp (TS1) from a date (D2) is the same as subtracting TS1 from TIMESTAMP(D2).

- The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. The precision of the result timestamp matches the precision of the timestamp operand. The date arithmetic portion is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. The time arithmetic portion is similar to time arithmetic except that it also considers the fractional seconds included in the duration. Thus, subtracting a duration, X, from a timestamp, TIMESTAMP1, where X is a DECIMAL(14+s,s) number, is equivalent to the expression:
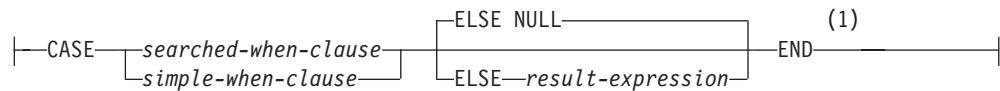
  ```
  TIMESTAMP1 - YEAR(X)  YEARS - MONTH(X)  MONTHS - DAY(X)  DAYS
              - HOUR(X)  HOURS - MINUTE(X)  MINUTES - SECOND(X, s)  SECONDS
  ```

  When subtracting a duration with non-zero scale or a labeled duration of SECOND or SECONDS with a value that includes fractions of a second, the subtraction is performed as if the timestamp value has up to 12 fractional second digits. The resulting value is assigned to a timestamp value with the timestamp precision of the timestamp operand which could result in truncation of fractional second digits.
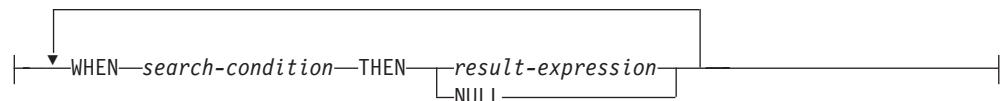
CASE expression

CASE expressions allow an expression to be selected based on the evaluation of one or more conditions.

case-expression:
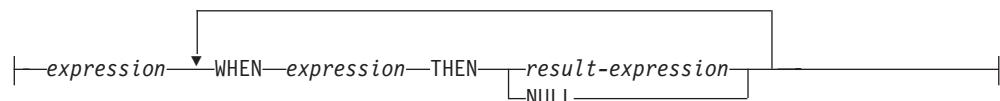
```
        ┌─ELSE NULL──────────────────┐        (1)
├─CASE──┬─searched-when-clause─┬──────┼──────────────────┼──END──────┤
        └─simple-when-clause───┘      └─ELSE─result-expression─┘
```

searched-when-clause:

```
   ┌─────────────────────────────────────────────────────┐
   ▼                                                      │
├────WHEN─search-condition─THEN──┬─result-expression─┬────────────────────┤
                                 └─NULL──────────────┘
```

simple-when-clause:

```
                ┌──────────────────────────────────────────┐
                ▼                                           │
├─expression──────WHEN─expression─THEN──┬─result-expression─┬───────────────┤
                                        └─NULL──────────────┘
```

Notes:

1   If the result type of *result-expression* is a row type, then the syntax represents a *row-case-expression* and can only be used where a *row-expression* is allowed.

In general, the value of the *case-expression* is the value of the *result-expression* following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is present then the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a case evaluates to unknown (because of NULLs), the case is not true and hence is treated the same way as a case that evaluates to false.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the *search-condition* in a *searched-when-clause* cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the *simple-when-clause*, the value of the *expression* before the first *WHEN* keyword is tested for equality with the value of the *expression* following the WHEN keyword. The data type of the *expression* before the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the WHEN keyword(s). The *expression* before the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is not deterministic or has an external action (SQLSTATE 42845).

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All result expressions must have compatible data types (SQLSTATE 42804).

Examples

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

```
SELECT EMPNO, LASTNAME,
  CASE SUBSTR(WORKDEPT,1,1)
    WHEN 'A' THEN 'Administration'
    WHEN 'B' THEN 'Human Resources'
    WHEN 'C' THEN 'Accounting'
    WHEN 'D' THEN 'Design'
    WHEN 'E' THEN 'Operations'
  END
  FROM EMPLOYEE;
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  CASE
    WHEN EDLEVEL < 15 THEN 'SECONDARY'
    WHEN EDLEVEL < 19 THEN 'COLLEGE'
    ELSE 'POST GRADUATE'
  END
  FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN NULL
  ELSE COMM/SALARY
  END) > 0.25;
```

- The following CASE expressions are the same:

```
SELECT LASTNAME,
  CASE
  WHEN LASTNAME = 'Haas' THEN 'President'
  ...

SELECT LASTNAME,
  CASE LASTNAME
  WHEN 'Haas' THEN 'President'
  ...
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 27 shows the equivalent expressions using CASE or these functions.

Table 27. Equivalent CASE Expressions

Expression	Equivalent Expression

CASE
  WHEN e1=e2 THEN NULL
  ELSE e1
END
	NULLIF(e1,e2)

CASE
  WHEN e1 IS NOT NULL THEN e1
  ELSE e2
END
	COALESCE(e1,e2)

Table 27. Equivalent CASE Expressions (continued)
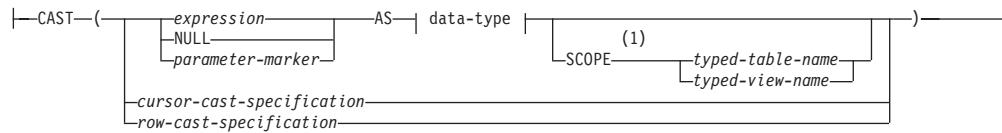
Expression	Equivalent Expression
``CASE
WHEN e1 IS NOT NULL THEN e1
ELSE COALESCE(e2,...,eN)
END```	COALESCE(e1,e2,...,eN)
``CASE
WHEN c1=var1 OR (c1 IS NULL AND var1 IS NULL)
THEN 'a'
WHEN c1=var2 OR (c1 IS NULL AND var2 IS NULL)
THEN 'b'
ELSE NULL
END```	DECODE(c1,var1, 'a', var2, 'b')
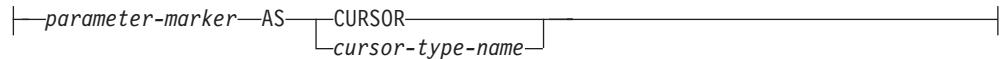
CAST specification

The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data-type*. If the cast is not supported, an error is returned (SQLSTATE 42846).

cast-specification:
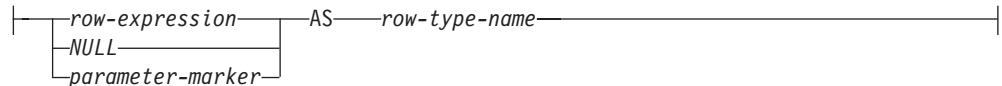
```
├──CAST──(──┬─expression────────┬──AS─┤ data-type ├──┬───────────────────────────────────┬──)──┤
            ├─NULL──────────────┤                    │               (1)                 │
            └─parameter-marker──┘                    └─SCOPE──┬─typed-table-name─┬────────┘
                                                             └─typed-view-name──┘
            ├─cursor-cast-specification─────────────────────────────────────────────────────┤
            └─row-cast-specification────────────────────────────────────────────────────────┘
```

cursor-cast-specification:

```
├──parameter-marker──AS──┬─CURSOR─────────────┬──┤
                         └─cursor-type-name───┘
```

row-cast-specification:

```
├──┬─row-expression─────┬──AS────row-type-name──┤
   ├─NULL───────────────┤
   └─parameter-marker───┘
```

data-type:

```
├──┬─┤ built-in-type ├──────┬──┤
   ├─array-type-name────────┤
   ├─distinct-type-name─────┤
   ├─structured-type-name───┤
   └─REF──(type-name2)──────┘
```

built-in-type:

```
├─┬─SMALLINT──────────────────────────────────────────────────────────────────┤
  ├─INTEGER─┐
  │  └─INT─┤
  ├─BIGINT─┘
  │                              ┌─(5,0)────────────────┐
  ├─┬─DECIMAL─┐──────────────────┼──────────────────────┤
  │  ├─DEC───┤                   └─(integer─┬──────────┬─)─┘
  │  ├─NUMERIC┤                             └─,0──────┤
  │  └─NUM───┘                                └─,integer─┘
  │           ┌─(53)──────┐
  ├─┬─FLOAT───┼───────────┤
  │  │         └─(integer)─┘
  │  ├─REAL───────────────┤
  │  │         ┌─PRECISION─┐
  │  └─DOUBLE──┴───────────┘
  │            ┌─(34)─┐
  ├─DECFLOAT───┼──────┤
  │            └─(16)─┘
  │                             ┌─(1)──────────────────────┐
  ├─┬─CHARACTER─┐───────────────┼──────────────────────────┤        ┌─FOR BIT DATA─┐ (2)
  │  └─CHAR────┘                └─(integer─┬────────────┬─)─┘        └──────────────┘
  │                                        ├─OCTETS─────┤
  │                                        └─CODEUNITS32─┘
  │    ┌─VARCHAR──────────────────┐──(integer─┬────────────┬─)─┐
  │    ├─CHARACTER──VARYING───────┤           ├─OCTETS─────┤
  │    └─CHAR─────────────────────┘           └─CODEUNITS32─┘
  │                                               ┌─(1M)─────────────────────┐
  │    ┌─CLOB──────────────────────────┐──────────┼──────────────────────────┤
  │    ├─CHARACTER──LARGE OBJECT───────┤          └─(integer─┬───┬─┬──────────┬─)─┘
  │    └─CHAR──────────────────────────┘                     ├─K─┤ ├─OCTETS─────┤
  │                                                          ├─M─┤ └─CODEUNITS32─┘
  │                                                          └─G─┘
  │                ┌─(1)────────────────────┐
  ├─┬─GRAPHIC──────┼────────────────────────┤
  │  │             └─(integer─┬───────────┬─)─┘
  │  │                        ├─CODEUNITS16─┤
  │  │                        └─CODEUNITS32─┘
  │  ├─VARGRAPHIC──(integer─┬───────────┬─)─┤
  │  │                      ├─CODEUNITS16─┤
  │  │                      └─CODEUNITS32─┘
  │  │            ┌─(1M)──────────────────────┐
  │  └─DBCLOB─────┼───────────────────────────┤
  │               └─(integer─┬───┬─┬───────────┬─)─┘
  │                          ├─K─┤ ├─CODEUNITS16─┤
  │                          ├─M─┤ └─CODEUNITS32─┘
  │                          └─G─┘
  │                             ┌─(1M)─────────┐
  ├─┬─BLOB───────────────────┐──┼──────────────┤
  │  └─BINARY LARGE OBJECT──┘  └─(integer─┬───┬─)─┘
  │                                       ├─K─┤
  │                                       ├─M─┤
  │                                       └─G─┘
  ├─DATE────────────────────────────────┤
  ├─TIME────────────────────────────────┤
  │                ┌─(6)──────┐
  ├─TIMESTAMP──────┼──────────┤
  │                └─(integer)─┘
  ├─BOOLEAN─────────────────────────────┤
  ├─XML─────────────────────────────────┤
  │  ┌─SYSPROC.─┐
  └─┴──────────┴──DB2SECURITYLABEL──────┘
```

Notes:

1   The SCOPE clause only applies to the REF data type.

2   The FOR BIT DATA clause can be specified in any order with the other column constraints that follow. The FOR BIT DATA clause cannot be specified with string units CODEUNITS32 (SQLSTATE 42613).

expression

> If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data-type*.

> When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than

DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

When casting an array, the target data type must be a user-defined array data type (SQLSTATE 42821). The data type of the elements of the array must be the same as the data type of the elements of the target array data type (SQLSTATE 42846). The cardinality of the array must be less than or equal to the maximum cardinality of the target array data type (SQLSTATE 2202F).

NULL
> If the cast operand is the keyword NULL, the result is a null value that has the specified *data-type*.

parameter-marker
> A parameter marker is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the specified *data-type* is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

cursor-cast-specification
> A cast specification used to indicate that a parameter marker is expected to be a cursor type. It can be used wherever an expression is supported in contexts that allow cursor types.

> *parameter-marker*
>> The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified cursor type.

> *CURSOR*
>> Specifies the built-in data type CURSOR.

> *cursor-type-name*
>> Specifies the name of a user-defined cursor type.

row-cast-specification
> A cast specification where the input is a row value and the result is a user-defined row type. A *row-cast-specification* is only valid where a *row-expression* is allowed.

> *row-expression*
>> The data type of *row-expression* must be a variable of row type that is anchored to the definition of a table or view. The data type of *row-expression* must not be a user-defined row type (SQLSTATE 42846).

> **NULL**
>> Specifies that the cast operand is the null value. The result is a row with the null value for every field of the specified data type.

> *parameter-marker*
>> The cast operand is a parameter marker and is considered a promise that the replacement will be assignable to the specified *row-type-name*.

> *row-type-name*
>> Specifies the name of a user-defined row type. The *row-expression* must be castable to *row-type-name* (SQLSTATE 42846).

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.
>
> If the data type is a distinct type defined with data type constraints, the data type constraints are applied and the constraints must evaluate to true or unknown otherwise an error is returned (SQLSTATE 23528).

built-in-type
> See "CREATE TABLE" for the description of built-in data types.

SCOPE
> When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.
>
> *typed-table-name*
> > The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).
>
> *typed-view-name*
> > The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM).

When numeric data is cast to character data, the result data type is a fixed-length character string. When character data is cast to numeric data, the result data type depends on the type of number specified. For example, if cast to integer, it becomes a large integer.

Examples

- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.

  ```
  SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
  ```
- Assume the existence of a distinct type called T_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume

the existence of a distinct type called R_YEAR that is defined on INTEGER and used to create column RETIRE_YEAR in PERSONNEL table. The following update statement could be prepared.

```
UPDATE PERSONNEL SET RETIRE_YEAR =?
  WHERE AGE = CAST( ? AS T_AGE)
```

The first parameter is an untyped parameter marker that would have a data type of R_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

The second parameter marker is a typed parameter marker that is cast as a distinct type T_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

Successful processing of this statement assumes that the SQL path includes the schema name of the schema (or schemas) where the two distinct types are defined.

- An application supplies a value that is a series of bits, for example an audio stream, and it should not undergo code page conversion before being used in an SQL statement. The application could use the following CAST:

```
CAST( ? AS VARCHAR(10000) FOR BIT DATA)
```

- Assume that an array type and a table have been created as follows:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0) ARRAY[5]

CREATE TABLE EMP_PHONES
  (ID          INTEGER,
   PHONENUMBER DECIMAL(10,0) )
```

The following procedure returns an array with the phone numbers for the employee with ID 1775. If there are more than five phone numbers for this employee, an error is returned (SQLSTATE 2202F).

```
CREATE PROCEDURE GET_PHONES(OUT EPHONES PHONELIST)
BEGIN
  SELECT CAST(ARRAY_AGG(PHONENUMBER) AS PHONELIST)
  INTO EPHONES
  FROM EMP_PHONES
  WHERE ID = 1775;
END
```

Field reference

A field of a row type is referenced using the field name qualified by a variable that returns a row type which includes a field with that field name, or an array element specification that returns a row type which includes a field with that field name.

field-reference:

```
├──┬─row-variable-name──────────────────┬─.field-name─────────────────────────┤
   └─row-array-element-specification─────┘
```

row-variable-name
> The name of a variable with a data type that is a row type.

row-array-element-specification
> An *array-element-specification* where the data type of the array element is a row type.

field-name
> The name of a field within the row type.

Examples

1. To reference a field of a simple row type:
```
CREATE OR REPLACE  TYPE REC_TYPE AS ROW
(
   ID INTEGER,
   NAME VARCHAR(10)
)@

BEGIN
    DECLARE NAME VARCHAR(10);
    DECLARE TMPREC REC_TYPE;
    SET NAME = TMPREC.NAME;
END @
```

2. To reference a field of a nested row type, you must assign the inner row to a temporary variable before a field of this inner row can be referenced:
```
CREATE OR REPLACE TYPE INNER_REC_TYPE AS ROW
(
  ID INTEGER,
  NAME VARCHAR(10)
)@
CREATE OR REPLACE  TYPE REC_TYPE AS ROW
(
  INNER_REC INNER_REC_TYPE
)@
CREATE OR REPLACE TYPE ARRAY_TYPE AS REC_TYPE ARRAY[INTEGER]@

BEGIN
  DECLARE TMPRECORD INNER_REC_TYPE;
  DECLARE TMPARRAY ARRAY_TYPE;
  DECLARE NAME VARCHAR(10);

  SET TMPRECORD = TMPARRAY[1].INNER_REC;
  SET NAME = TMPRECORD.NAME;
  END @
```
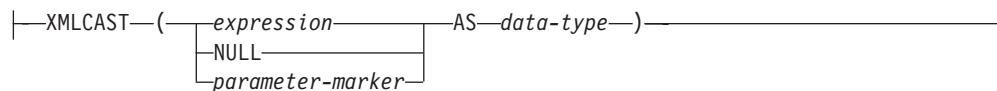
XMLCAST specification

The XMLCAST specification returns the cast operand (the first operand) cast to the type specified by the data type. XMLCAST supports casts involving XML values, including conversions between non-XML data types and the XML data type. If the cast is not supported, an error is returned (SQLSTATE 22003).

xmlcast-specification:

```
├──XMLCAST──(──┬──expression──────┬──AS──data-type──)──────────────┤
              ├──NULL────────────┤
              └──parameter-marker─┘
```

expression
> If the cast operand is an expression (other than a parameter marker or NULL), the result is the argument value converted to the specified target data type. The expression or the target data type must be the XML data type (SQLSTATE 42846).

NULL
> If the cast operand is the keyword NULL, the target data type must be the XML data type (SQLSTATE 42846). The result is a null XML value.

parameter-marker
> If the cast operand is a parameter marker, the target data type must be XML (SQLSTATE 42846). A parameter marker is normally considered to be an expression, but is documented separately in this case because it has special meaning. If the cast operand is a parameter marker, the specified data type is considered to be a promise that the replacement will be assignable to the specified (XML) data type (using store assignment). Such a parameter marker is considered to be a typed parameter marker, which is treated like any other typed value for the purpose of function resolution, a describe operation on a select list, or column assignment.

data-type
> The name of an existing SQL data type. If the name is not qualified, the SQL path is used to perform data type resolution. If a data type has associated attributes, such as length or precision and scale, these attributes should be included when specifying a value for *data-type*. CHAR defaults to a length of 1, and DECIMAL defaults to a precision of 5 and a scale of 0 if not specified. Restrictions on the supported data types are based on the specified cast operand.
>
> - For a cast operand that is an expression, the supported target data types depend on the data type of the cast operand (source data type).
> - For a cast operand that is the keyword NULL, the target data type must be XML.
> - For a cast operand that is a parameter marker, the target data type must be XML.

Note: Support in non-Unicode databases: When XMLCAST is used to convert an XML value to an SQL data type, code page conversion is performed. The encoding of the cast expression is converted from UTF-8 to the database code page. Characters in the original expression that are not present in the database code page are replaced by substitution characters as a result of this conversion.

Examples

- Create a null XML value.

  ```
  XMLCAST(NULL AS XML)
  ```

- Convert a value extracted from an XMLQUERY expression into an INTEGER:

  ```
  XMLCAST(XMLQUERY('$m/PRODUCT/QUANTITY'
    PASSING xmlcol AS "m") AS INTEGER)
  ```

- Convert a value extracted from an XMLQUERY expression into a varying-length character string:

  ```
  XMLCAST(XMLQUERY('$m/PRODUCT/ADD-TIMESTAMP'
    PASSING xmlcol AS "m") AS VARCHAR(30))
  ```

- Convert a value extracted from an SQL scalar subquery into an XML value.

  ```
  XMLCAST((SELECT quantity FROM product AS p
    WHERE p.id = 1077) AS XML)
  ```

ARRAY element specification

The ARRAY element specification returns the element from an array specified by *expression*. If any argument to *expression* is null, the null value is returned.

array-element-specification:

```
                          (1)
├──┬─array-variable───────────────────────────────────┬──[─expression─]──────┤
   └─CAST──(─parameter-marker──AS──array-type-name──)──┘
```

Notes:

1   If the data type of the elements in the array is a row type, then the syntax represents an array-element-specification with a row data type and can only be used where a *row-expression* is allowed.

array-variable
An SQL variable, SQL parameter, or global variable of an array type.

CAST (*parameter-marker* **AS** *array-type-name***)**
A parameter marker is normally considered to be an expression, but in this case it must explicitly be cast to a user-defined array data type.
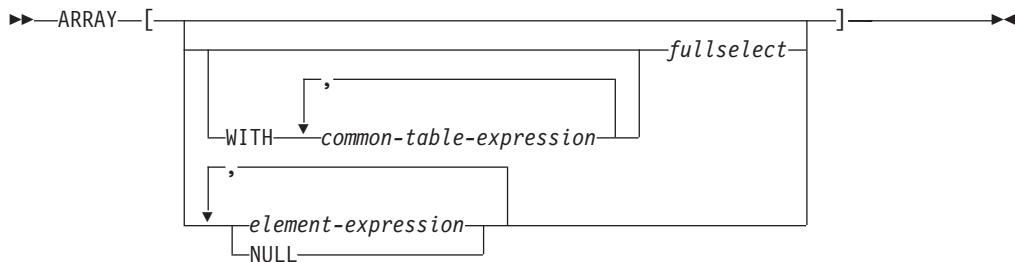
[*expression***]**
Specifies the array index of the element that is to be extracted from the array. The array index of an ordinary array must be assignable to INTEGER (SQLSTATE 428H1); its value must be between 1 and the cardinality of the array (SQLSTATE 2202E). The array index of an associative array must be assignable to the index data type (SQLSTATE 428H1).

Array constructor

An array constructor is a language element that can be used to define and construct an array data type value within a valid context.

Syntax

```
►►──ARRAY──[─┬──────────────────────────────────────┬──┬──fullselect──┬──]──►◄
             │              ┌─,────────────────────┐ │  │              │
             └─WITH──────▼──common-table-expression─┴──┘  │              │
             ┌─,──────────────────┐                       │              │
             ▼──element-expression─┴───────────────────────┘
              └─NULL──┘
```

Authorization

No specific authorizations are required to reference an array constructor within an SQL statement, however for the statement execution to be successful all other authorization requirements for the statement must be satisfied.

Description

WITH *common-table-expression*
   Defines a common table expression for use with the following fullselect.

fullselect
   A fullselect that returns a single column. The values that are returned by the fullselect for each row are the elements of the array. The cardinality of the array is equal to the number of rows that are returned by the fullselect. If the fullselect includes an *order-by-clause*, the order determines the order in which row values are assigned to elements of the array. If no *order-by-clause* is specified, the order in which row values are assigned to elements of the array is not deterministic.

element-expression
   An expression that defines the value of an element in the array. The cardinality of the array is equal to the number of element expressions. The first *element-expression* is assigned to the array element with array index 1. The second *element-expression* is assigned to the array element with array index 2 and so on. Every *element-expression* must have a compatible data type with every other *element-expression* and the base type of the array is determined using the "Rules for result data types" topic.

NULL
   Specifies the null value.

If no value is specified within the brackets, the result is an empty array.

Rules

- The base type of the *array-constructor*, as derived from the *element-expression*s or the *fullselect*, must be assignable to the base type of the target array (SQLSTATE 42821).
- The number of elements in the *array-constructor* must not exceed the maximum cardinality of the target array variable (SQLSTATE 2202F).

Array constructor

Notes

- An array constructor can be used to define only an ordinary array with elements that are not a row type. An array constructor cannot be used to define an associative array or an ordinary array with elements that are a row type. Such arrays can only be constructed by assigning the individual elements.

Examples

Example 1: Set the array variable RECENT_CALLS of array type PHONENUMBERS to an array of fixed numbers.

```
SET RECENT_CALLS = ARRAY[9055553907, 4165554213, 4085553678]
```

Example 2: Set the array variable DEPT_PHONES of array type PHONENUMBERS to an array of phone numbers retrieved from the DEPARTMENT_INFO table.
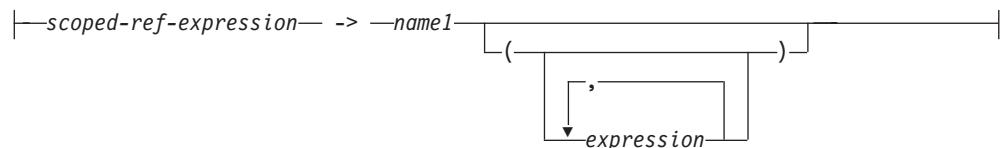
```
SET DEPT_PHONES = ARRAY[SELECT DECIMAL(AREA_CODE CONCAT '555' CONCAT EXTENSION,16)
                          FROM DEPARTMENT_INFO
                          WHERE DEPTID = 624]
```

Dereference operation

The scope of the scoped reference expression is a table or view called the *target* table or view.

The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

dereference-operation:

```
├──scoped-ref-expression── -> ──name1──────────────────────────────────────┤
                                        └─(─┬───────────────┬─)─┘
                                            │  ┌─,─────────┐ │
                                            └──▼─expression─┴─┘
```

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.

Dereference operation

Examples

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

      SELECT EMPNO, DEPTREF->DEPTNAME
        FROM EMPLOYEE
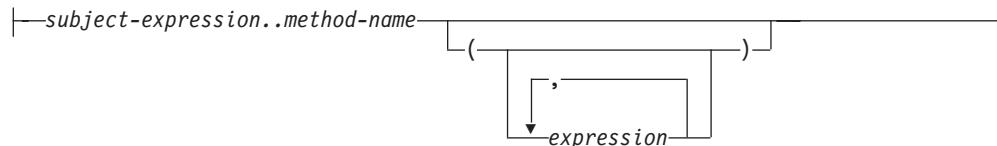
- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

      SELECT EMPNO, DEPTREF->BUDGET('1997') **AS** DEPTBUDGET97
        FROM EMPLOYEE

Method invocation

Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

method-invocation:

```
├──subject-expression..method-name──────────────────────────────────┤
                                    └─(─┬───────────────┬─)─┘
                                        │   ┌─,──────┐   │
                                        └──▼─expression─┴─┘
```

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).

Method invocation

Example

- Use the double-dot operator to invoke a method called AREA. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that the method AREA has been defined previously for the CIRCLE type as AREA() RETURNS DOUBLE.

  ```
  SELECT CIRCLE_COL..AREA() FROM RINGS
  ```
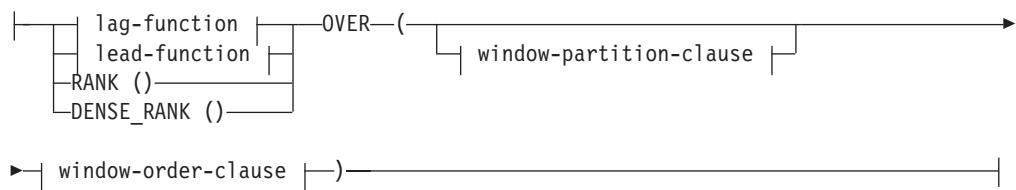
OLAP specification

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing aggregate function information as a scalar value in a query result.

OLAP-specification:

```
├──┬─ ordered-OLAP-specification ─┬──────────────────────────────────┤
   ├─ numbering-specification ────┤
   └─ aggregation-specification ──┘
```
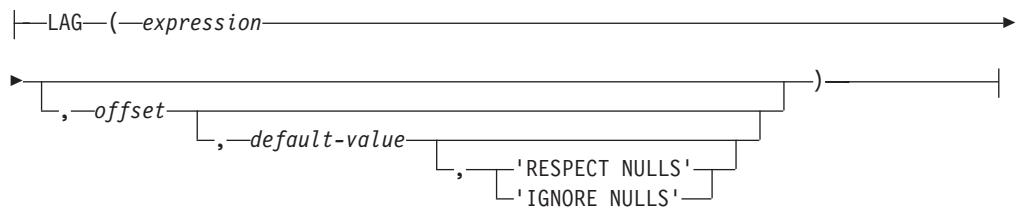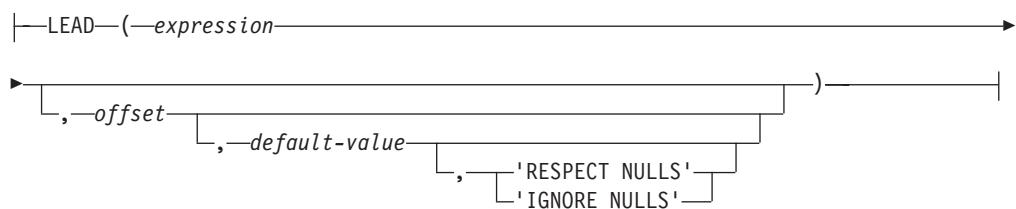
ordered-OLAP-specification:

```
├──┬─┤ lag-function ├──────┬──OVER──(──┬──────────────────────────┬──►
   │ ├─┤ lead-function ├────┤           └─┤ window-partition-clause ├┘
   ├───RANK ()─────────────┤
   └───DENSE_RANK ()───────┘

►──┤ window-order-clause ├──)──────────────────────────────────────┤
```

lag-function:

```
├───LAG──(──expression──────────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────┬──)──────┤
   └─,──offset──┬─────────────────────────────────────────┘
               └─,──default-value──┬──────────────────────┘
                                   └─,──┬─'RESPECT NULLS'─┬┘
                                        └─'IGNORE NULLS'──┘
```

lead-function:

```
├───LEAD──(──expression─────────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────┬──)──────┤
   └─,──offset──┬─────────────────────────────────────────┘
               └─,──default-value──┬──────────────────────┘
                                   └─,──┬─'RESPECT NULLS'─┬┘
                                        └─'IGNORE NULLS'──┘
```

window-partition-clause:

```
                    ┌──────,──────────────┐
                    ▼                      │
├───PARTITION BY─────partitioning-expression──┴─────────────────────┤
```

window-order-clause:

```
                ┌──────,──────────────────────────────┐
                ▼                                      │
├───ORDER BY──┬───sort-key-expression──┬─ asc-option ─┬┴───────────┤
             │                         └─ desc-option ┘
             └───ORDER OF──table-designator───────────┘
```
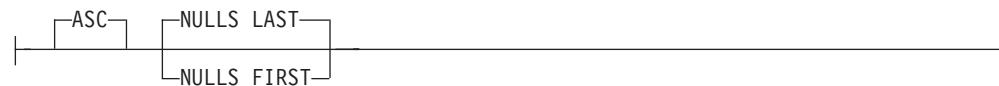
OLAP specification

asc-option:

```
          ┌─ASC─┐  ┌─NULLS LAST──┐
├─────────┴─────┴──┴─────────────┴──────────────────────────────────┤
                   └─NULLS FIRST─┘
```

desc-option:

```
                  ┌─NULLS FIRST─┐
├──DESC───────────┴─────────────┴───────────────────────────────────┤
                  └─NULLS LAST──┘
```

numbering-specification:

```
                                      ┌──────────────────────────┐
├──ROW_NUMBER ()──OVER──(─────────────┴──window-partition-clause──┴────►
```
```
  ┌──────────────────────────┐
►─┴──window-order-clause──────┴──)──────────────────────────────────┤
```

aggregation-specification:

```
     ┌─column-function─────────┐   (1)
├────┤                         ├──OVER──(──────────────────────────────►
     └─OLAP-column-function────┘          └──window-partition-clause──┘
```
```
   ┌─RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING─────────┐
►──┤                                                                   ├──)──┤
   └──window-order-clause──┬─RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW─┐
                           └──window-aggregation-group-clause────────────────┘
```

OLAP-column-function:
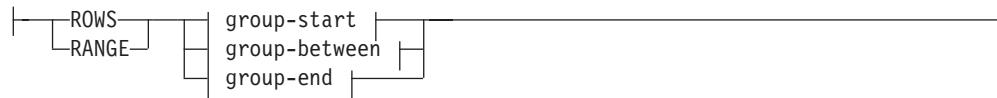
```
      ┌──first-value-function─────┐
├─────┼──last-value-function──────┼──────────────────────────────────┤
      └──ratio-to-report-function─┘
```

first-value-function:

```
├──FIRST_VALUE──(──expression────────────────────────)──────────────┤
                               └─,─┬─'RESPECT NULLS'─┬─┘
                                   └─'IGNORE NULLS'──┘
```

last-value-function:

```
├──LAST_VALUE──(──expression────────────────────────)───────────────┤
                              └─,─┬─'RESPECT NULLS'─┬─┘
                                  └─'IGNORE NULLS'──┘
```

ratio-to-report-function:

```
├──RATIO_TO_REPORT──(──expression──)──────────────────────────────────────────────┤
```

window-aggregation-group-clause:

```
├──┬─ROWS──┬──┬─ group-start ──┬──────────────────────────────────────────────────┤
   └─RANGE─┘  ├─ 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 an aggregate function in
     aggregation-specification (SQLSTATE 42887).

An OLAP function can be included in expressions in a select-list or the ORDER BY
clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used
within an argument to an XMLQUERY or XMLEXISTS expression (SQLSTATE
42903). An OLAP function cannot be used as an argument of an aggregate function
(SQLSTATE 42607). The query result to which the OLAP function is applied is the
result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows
over which the function is applied, and in what order. When used with an
aggregate function, the applicable rows can be further refined, relative to the

current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three month period.

The 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 constant (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows before the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The LEAD function returns the expression value for the row at *offset* rows after the current row. The *offset* must be a positive integer constant (SQLSTATE 42815). An *offset* value of 0 means the current row. If a window-partition-clause is specified, *offset* means *offset* rows after the current row and within the current partition. If *offset* is not specified, the value 1 is used. If *default-value* (which can be an expression) is specified, it will be returned if the offset goes beyond the scope of the current partition. Otherwise, the null value is returned. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all rows are null, *default-value* (or the null value if *default-value* was not specified) is returned.

The FIRST_VALUE function returns the expression value for the first row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, FIRST_VALUE returns the null value.

The LAST_VALUE function returns the expression value for the last row in an OLAP window. If 'IGNORE NULLS' is specified, all rows where the expression value for the row is the null value are not considered in the calculation. If 'IGNORE NULLS' is specified and all values in the OLAP window are null, LAST_VALUE returns the null value.

The data type of the result of FIRST_VALUE, LAG, LAST_VALUE, and LEAD is the data type of the expression. The result can be null.

The RATIO_TO_REPORT function returns the ratio of an argument to the sum of the arguments in an OLAP partition. For example, the following functions are equivalent:

```
RATIO_TO_REPORT(expression) OVER (...)
CAST(expression AS DECFLOAT(34)) / SUM(expression) OVER(...)
```

The division is always performed using DECFLOAT(34). The result data type is DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

PARTITION BY (_partitioning-expression_**,...)**
> Defines the partition within which the function is applied. A _partitioning-expression_ is an expression that is used in defining the partitioning of the result set. Each _column-name_ that is referenced in a _partitioning-expression_ must unambiguously reference a column of the result table of the subselect that contains the OLAP specification (SQLSTATE 42702 or 42703). A _partitioning-expression_ cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845).

window-order-clause

> **ORDER BY (**_sort-key-expression_**,...)**
>> Defines the ordering of rows within a partition that determines the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (it does not define the ordering of the query result set).

> _sort-key-expression_
>> An expression used in defining the ordering of the rows within a window partition. Each column name referenced in a _sort-key-expression_ must unambiguously reference a column of the result set of the subselect, including the OLAP function (SQLSTATE 42702 or 42703). A _sort-key-expression_ cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any function or query that is not deterministic or that has an external action (SQLSTATE 42845). This clause is required for the RANK and DENSE_RANK functions (SQLSTATE 42601).

> **ASC**
>> Uses the values of the sort-key-expression in ascending order.

> **DESC**
>> Uses the values of the sort-key-expression in descending order.

NULLS FIRST
The window ordering considers null values *before* all non-null values in the sort order.

NULLS LAST
The window ordering considers null values *after* all non-null values in the sort order.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

window-aggregation-group-clause
The aggregation group of a row R is a set of rows defined in relation to R (in the ordering of the rows of R's partition). This clause specifies the aggregation group. If this clause is not specified and a window-order-clause is also not specified, the aggregation group consists of all the rows of the window partition. This default can be specified explicitly using RANGE (as shown) or ROWS.

If window-order-clause is specified, the default behavior is different when window-aggregation-group-clause is not specified. The window aggregation group consists of all rows of the partition of R that precede R 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.

unsigned-constant **PRECEDING**

Specifies either the range or number of rows preceding the current row. If ROWS is specified, then *unsigned-constant* must be zero or a positive integer indicating a number of rows. If RANGE is specified, then the data type of *unsigned-constant* must be comparable to the type of the *sort-key-expression* of the *window-order-clause*. There can only be one *sort-key-expression*, and the data type of the *sort-key-expression* must allow subtraction. This clause cannot be specified in *group-bound2* if *group-bound1* is CURRENT ROW or *unsigned-constant* FOLLOWING.

unsigned-constant **FOLLOWING**

Specifies either the range or number of rows following the current row. If ROWS is specified, then *unsigned-constant* must be zero or a positive integer indicating a number of rows. If RANGE is specified, then the data type of *unsigned-constant* must be comparable to the type of the *sort-key-expression* of the *window-order-clause*. There can only be one *sort-key-expression*, and the data type of the *sort-key-expression* must allow addition.

Examples

- 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
```

OLAP specification

- 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, FIRSTNME, TOTAL_SALARY, RANK_SALARY
  FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
    RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
    FROM EMPLOYEE) AS RANKED_EMPLOYEE
  WHERE RANK_SALARY < 6
  ORDER BY RANK_SALARY
```

Note that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

- For each department, list employee salaries and show how much less each person makes compared to the employee in that department with the next highest salary.

```
SELECT EMPNO, WORKDEPT, LASTNAME, FIRSTNME, JOB, SALARY,
    LEAD(SALARY, 1) OVER (PARTITION BY WORKDEPT
      ORDER BY SALARY) - SALARY AS DELTA_SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, SALARY
```

- Calculate an employee's salary relative to the salary of the employee who was first hired for the same type of job.

```
SELECT JOB, HIREDATE, EMPNO, LASTNAME, FIRSTNME, SALARY,
    FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
      ORDER BY HIREDATE) AS FIRST_SALARY,
    SALARY - FIRST_VALUE(SALARY) OVER (PARTITION BY JOB
      ORDER BY HIREDATE) AS DELTA_SALARY
  FROM EMPLOYEE
  ORDER BY JOB, HIREDATE
```

- Calculate the average close price for stock XYZ during the month of January, 2006. If a stock doesn't trade on a given day, its close price in the DAILYSTOCKDATA table is the null value. Instead of returning the null value for days that a stock doesn't trade, use the COALESCE function and LAG function to return the close price for the most recent day the stock was traded. Limit the search for a previous not-null close value to one month before January 1st, 2006.

```
WITH V1(SYMBOL, TRADINGDATE, CLOSEPRICE) AS
(
SELECT SYMBOL, TRADINGDATE,
  COALESCE(CLOSEPRICE,
    LAG(CLOSEPRICE,
      1,
      CAST(NULL AS DECIMAL(8,2)),
      'IGNORE NULLS')
    OVER (PARTITION BY SYMBOL
    ORDER BY TRADINGDATE)
)
FROM DAILYSTOCKDATA
WHERE SYMBOL = 'XYZ' AND
  TRADINGDATE BETWEEN '2005-12-01' AND '2006-01-31'
)
SELECT SYMBOL, AVG(CLOSEPRICE) AS AVG
  FROM V1
  WHERE TRADINGDATE BETWEEN '2006-01-01' AND '2006-01-31'
  GROUP BY SYMBOL
```

- Calculate the 30-day moving average for stocks ABC and XYZ during the year 2005.

```
WITH V1(SYMBOL, TRADINGDATE, MOVINGAVG30DAY) AS
(
SELECT SYMBOL, TRADINGDATE,
  AVG(CLOSEPRICE) OVER (PARTITION BY SYMBOL
    ORDER BY TRADINGDATE
  ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
FROM DAILYSTOCKDATA
WHERE SYMBOL IN ('ABC', 'XYZ')
  AND TRADINGDATE BETWEEN DATE('2005-01-01') - 2 MONTHS
    AND '2005-12-31'
)
SELECT SYMBOL, TRADINGDATE, MOVINGAVG30DAY
  FROM V1
  WHERE TRADINGDATE BETWEEN '2005-01-01' AND '2005-12-31'
  ORDER BY SYMBOL, TRADINGDATE
```

- 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
```

- For each employee, calculate the average salary for the set of employees that includes those employees in the same department who have an education level 1 lower and 1 higher than the employee.
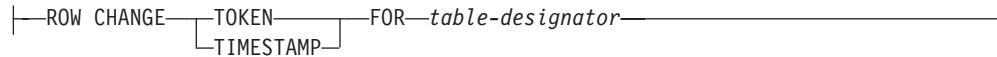
```
SELECT WORKDEPT, EDLEVEL, SALARY, AVG(SALARY)
      OVER (PARTITION BY WORKDEPT
      ORDER BY EDLEVEL
      RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    FROM EMPLOYEE
    ORDER BY WORKDEPT, EDLEVEL
```

ROW CHANGE expression

A ROW CHANGE expression returns a token or a timestamp that represents the last change to a row.

row-change-expression:

```
├──ROW CHANGE──┬─TOKEN─────┬──FOR──table-designator─────────────────────────────┤
               └─TIMESTAMP─┘
```

TOKEN
Specifies that a BIGINT value representing a relative point in the modification sequence of a row is to be returned. If the row has not been changed, the result is a token that represents when the initial value was inserted. The result can be null. ROW CHANGE TOKEN is not deterministic.

TIMESTAMP
Specifies that a TIMESTAMP value representing the last time that a row was changed is to be returned. If the row has not been changed, the result is the time that the initial value was inserted. The result can be null. ROW CHANGE TIMESTAMP is not deterministic.

FOR *table-designator*
Identifies the table in which the expression is referenced. The *table-designator* must uniquely identify a base table, view, or nested table expression (SQLSTATE 42867). If *table-designator* identifies a view or a nested table expression, the ROW CHANGE expression returns the TOKEN or TIMESTAMP of the base table of the view or nested table expression. The view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42867). If the *table-designator* is a view or nested table expression, it must be deletable (SQLSTATE 42703). For information about deletable views, see the "Notes" section of "CREATE VIEW". The table designator of a ROW CHANGE TIMESTAMP expression must resolve to a base table that contains a row change timestamp column (SQLSTATE 55068).

Notes

- ROW CHANGE TOKEN and ROW CHANGE TIMESTAMP are not valid expressions for a column-organized table (SQLSTATE 42703).

Examples

- Return a timestamp value that corresponds to the most recent change to each row from the EMPLOYEE table for employees in department 20. Assume that the EMPLOYEE table has been altered to contain a column defined with the ROW CHANGE TIMESTAMP clause.

```
SELECT ROW CHANGE TIMESTAMP FOR EMPLOYEE
   FROM EMPLOYEE WHERE DEPTNO = 20
```

- Return a BIGINT value that represents a relative point in the modification sequence of the row corresponding to employee number 3500. Also return the RID_BIT scalar function value that is to be used in an optimistic locking DELETE scenario. Specify the WITH UR option to get the latest ROW CHANGE TOKEN value.

```
SELECT ROW CHANGE TOKEN FOR EMPLOYEE, RID_BIT (EMPLOYEE)
   FROM EMPLOYEE WHERE EMPNO = '3500' WITH UR
```

The preceding statement succeeds whether or not there is a row change timestamp column in the EMPLOYEE table. The following searched DELETE

statement deletes the row specified by the ROW CHANGE TOKEN and RID_BIT values from the preceding SELECT statement, assuming the two parameter marker values are set to the values obtained from the preceding statement.

```
DELETE FROM EMPLOYEE E
  WHERE RID_BIT (E) = ? AND ROW CHANGE TOKEN FOR E = ?
```
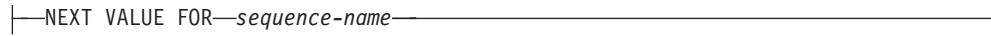
Sequence reference

A sequence reference is an expression which references a sequence defined at the application server.

sequence-reference:

```
├──┬── nextval-expression ──┬───────────────────────────────────────────────┤
   └── prevval-expression ──┘
```

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

- *Authorization:* If a sequence-reference is used in a statement, the privileges held by the authorization ID of the statement must include at least one of the following privileges:
  - The USAGE privilege on the sequence
  - DATAACCESS authority
- A new value is generated for a sequence when a NEXT VALUE expression specifies the name of that sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result.
- The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown in the following example:

```
INSERT INTO order(orderno, cutno)
  VALUES (NEXT VALUE FOR order_seq, 123456);

INSERT INTO line_item (orderno, partno, quantity)
  VALUES (PREVIOUS VALUE FOR order_seq, 987654, 1);
```

- NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:
  - select-statement or SELECT INTO statement (within the select-clause, provided that the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword)
  - INSERT statement (within a VALUES clause)
  - INSERT statement (within the select-clause of the fullselect)
  - UPDATE statement (within the SET clause (either a searched or a positioned UPDATE statement), except that NEXT VALUE cannot be specified in the select-clause of the fullselect of an expression in the SET clause)
  - SET Variable statement (except within the select-clause of the fullselect of an expression; a NEXT VALUE expression can be specified in a trigger, but a PREVIOUS VALUE expression cannot)
  - VALUES INTO statement (within the select-clause of the fullselect of an expression)
  - CREATE PROCEDURE statement (within the routine-body of an SQL procedure)
  - CREATE TRIGGER statement within the triggered-action (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
- NEXT VALUE and PREVIOUS VALUE expressions cannot be specified (SQLSTATE 428F9) in the following places:
  - Join condition of a full outer join
  - DEFAULT value for a column in a CREATE 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, as described previously
  - SELECT statement for which the outer SELECT contains a DISTINCT operator
  - Join condition of a join
  - SELECT statement for which the outer SELECT contains a GROUP BY clause
  - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT set operator

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.

- *Scope of PREVIOUS VALUE:* The value of PREVIOUS VALUE persists until the next value is generated for the sequence in the current session, the sequence is dropped or altered, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The value of PREVIOUS VALUE cannot be directly set and is a result of executing the NEXT VALUE expression for the sequence.

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the PREVIOUS VALUE for a sequence should be relied on only until the end of the transaction. Examples of where this type of situation can occur include applications that use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.

- If in generating a value for a sequence, the maximum value for the sequence is exceeded (or the minimum value for a descending sequence) and cycles are not permitted, then an error occurs (SQLSTATE 23522). In this case, the user could ALTER the sequence to extend the range of acceptable values, or enable cycles for the sequence, or DROP and CREATE a new sequence with a different data type that has a larger range of values.

  For example, a sequence may have been defined with a data type of SMALLINT, and eventually the sequence runs out of assignable values. DROP and re-create the sequence with the new definition to redefine the sequence as INTEGER.

- A reference to a NEXT VALUE expression in the select statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression for each row that is fetched from the database. If blocking is done at the client, the values may have been generated at the server before the processing of the FETCH statement. This can occur when there is blocking of the rows of the result table. If the client application does not explicitly FETCH all the rows that the database has materialized, then the

application will not see the results of all the generated sequence values (for the materialized rows that were not returned).

- A reference to a PREVIOUS VALUE expression in the select statement of a cursor refers to a value that was generated for the specified sequence before the opening of the cursor. However, closing the cursor can affect the values returned by PREVIOUS VALUE for the specified sequence in subsequent statements, or even for the same statement in the event that the cursor is reopened. This would be the case when the select statement of the cursor included a reference to NEXT VALUE for the same sequence name.

- *Syntax alternatives*: The following are supported for compatibility with previous versions of DB2 and with other database products. These alternatives are non-standard and should not be used.
   - NEXTVAL and PREVVAL can be specified in place of NEXT VALUE and PREVIOUS VALUE
   - *sequence-name*.NEXTVAL can be specified in place of NEXT VALUE FOR *sequence-name*
   - *sequence-name*.CURRVAL can be specified in place of PREVIOUS VALUE FOR *sequence-name*

Examples

Assume that there is a table called "order", and that a sequence called "order_seq" is created as follows:

```
CREATE SEQUENCE order_seq
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  CACHE 24
```

Following are some examples of how to generate an "order_seq" sequence number with a NEXT VALUE expression:

```
INSERT INTO order(orderno, custno)
  VALUES (NEXT VALUE FOR order_seq, 123456);
```

or

```
UPDATE order
  SET orderno = NEXT VALUE FOR order_seq
  WHERE custno = 123456;
```

or

```
VALUES NEXT VALUE FOR order_seq INTO :hv_seq;
```

Subtype treatment

The *subtype-treatment* is used to cast a structured type expression into one of its subtypes.

subtype-treatment:

```
├──TREAT──(──expression──AS──data-type──)──────────────────────────────────────┤
```

The static type of *expression* must be a user-defined structured type, and that type must be the same type as, or a supertype of, *data-type*. If the type name in *data-type* is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is *data-type*, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not *data-type* or a subtype of *data-type*, an error is returned (SQLSTATE 0D000).

Example

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as RGB() RETURNS DOUBLE.

  ```
  SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
    FROM RINGS
  ```

At run time, if there are instances of dynamic type CIRCLE, an error is raised (SQLSTATE 0D000). This error can be avoided by using the TYPE predicate in a CASE expression, as follows:

```
SELECT (CASE
  WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
    THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
    ELSE NULL
  END)
  FROM RINGS
```

Determining data types of untyped expressions

An untyped expression refers to the usage of a parameter marker which is specified without a target data type associated with it, a null value which is specified without a target data type associated with it, or a DEFAULT keyword.

Untyped expressions can be used in SQL statements as long as one of the following conditions is true:

- A PREPARE statement is being executed by a CLI or JDBC application to compile the SQL statement; the client interface is using deferred prepare; and the registry variable, DB2_DEFERRED_PREPARE_SEMANTICS is set to YES. In this case, any untyped parameter marker derives its data type based on the input descriptor associated with the subsequent OPEN or EXECUTE statement. The length attribute is set to the maximum of the length according to the UNTYPED row, as described in the Table 24 on page 223 in "Functions" and the length as determined from the following tables. For data types not listed as a target type in Table 24 on page 223 in "Functions", the length from the input descriptor associated with the subsequent OPEN or EXECUTE statement will be used. The data types and lengths may be modified depending on the usage of the untyped parameter marker in the SQL statement.

- The data type can be determined based on the context in the SQL statement. These locations and the resulting data types are shown in the following table. The locations are grouped into expressions, predicates, built-in functions, and user-defined routines to assist in determining the applicability of an untyped expression. If the data type cannot be determined based on the context, an error is issued.

For some cases not listed, untyped expressions in a select list will be resolved to a data type determined based on the usage in the SQL statement.

The code page of the untyped expression is determined by the context. Where there is no context, the code page is the same as if the untyped expression was cast to a VARCHAR data type.

The tables that follow show character string and graphic string data types in string units associated with a database environment where the string units default is SYSTEM. If the Unicode database environment has the string units set to CODEUNITS32, then any character string or graphic string length attributes that represent the data type maximum length should be considered to represent the data type maximum in CODEUNITS32. All character string or graphic string data types have the default string units of the database environment.

Determining data types of untyped expressions

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES)

Untyped Expression Location	Data Type
Alone in a select list	If the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement, then an error is returned, unless the untyped expression is the null value. In such cases, the data type is VARCHAR(1).
	If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Both operands of a single arithmetic operator, after considering operator precedence and order of operation rules Includes cases such as: (? + ?) + 10	DECFLOAT(34)
One operand of a single operator in an arithmetic expression (not a datetime expression) Includes cases such as: ? + (? * 10)	The data type of the other operand
Labelled duration within a datetime expression (note that the portion of a labelled duration that indicates the type of units cannot be a parameter marker)	DECIMAL(15,0)
Any other operand of a datetime expression (for example, 'timecol + ?' or '? - datecol')	Error
Both operands of a CONCAT operator	VARCHAR(254)
One operand of a CONCAT operator when the other operand is a non-CLOB character data type	If one operand is either CHAR(*n*) or VARCHAR(*n*), where *n* is less than 128, the other is VARCHAR(254 - *n*); in all other cases, the data type is VARCHAR(254)
One operand of a CONCAT operator, when the other operand is a non-DBCLOB graphic data type	If one operand is either GRAPHIC(*n*) or VARGRAPHIC(*n*), where *n* is less than 64, the other is VARGRAPHIC(127 - *n*); in all other cases, the data type is VARGRAPHIC(127)
One operand of a CONCAT operator, when the other operand is a large object string	Same as that of the other operand
The expression following the CASE keyword in a simple CASE expression	Result of applying the "Rules for the result data types" to the expressions following the WHEN keyword that are other than untyped expressions
At least one of the result-expressions in a CASE expression (both simple and searched), with the rest of the result-expressions being untyped expressions	Error

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Expression Location	Data Type
Any or all expressions following the WHEN keyword in a simple CASE expression	Result of applying the "Rules for result data types" to the expression following CASE and the expressions following WHEN keyword that are other than an untyped expression
A result-expression in a CASE expression (both simple and searched), when at least one result-expression is not an untyped expression	Result of applying the "Rules for result data types" to all result-expressions that are other than an untyped expression
Alone as a column-expression in a single-row VALUES clause that is not within an INSERT statement and not within the VALUES clause of in insert operation of a MERGE statement	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the column-expressions in the same position in all other row-expressions are untyped expressions	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in a multi-row VALUES clause that is not within an INSERT statement, and for which the expression in the same position of at least one other row-expression is not an untyped expression	Result of applying the "Rules for result data types" on all operands that are other than untyped expressions
Alone as a column-expression in a single-row VALUES clause within an INSERT statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression in a multi-row VALUES clause within an INSERT statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.

Determining data types of untyped expressions

Table 28. Untyped Expression Usage in Expressions (Including Select List, CASE, and VALUES) (continued)

Untyped Expression Location	Data Type
Alone as a column-expression in a values-clause of the source table for a MERGE statement	Error if the untyped expression is unnamed or is named but not subsequently referenced in the SQL statement. If the untyped expression is named and subsequently referenced in the SQL statement, then the data type may be determined from the subsequent usage. For more information, refer to the "Determining data type from usage" note that follows this table.
Alone as a column-expression in the VALUES clause of an insert operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression on the right side of assignment-clause for an update operation of a MERGE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
Alone as a column-expression on the right side of a SET clause in an UPDATE statement	The data type of the column. If the column is defined as a user-defined distinct type, it is the source data type of the user-defined distinct type. If the column is defined as a user-defined structured type, it is the structured type, also indicating the return type of the transform function.
As a value on the right side of a SET special register statement	The data type of the special register
Argument of the TABLESAMPLE clause of the tablesample-clause of a table-reference	DOUBLE
Argument of the REPEATABLE subclause of the tablesample-clause of a table-reference	INTEGER
As a value in a FREE LOCATOR statement	Locator
As a value for the password in a SET ENCRYPTION PASSWORD statement	VARCHAR(128)

Note:

Determining data type from usage

> The following is an example of how the data type for an untyped expression can be determined from subsequent usage:

> If the named untyped expression is subsequently referenced in a comparison operator, it will then have the data type of the other operand. If there are multiple references of the named untyped expression in the SQL statement, the data type, length, precision, scale, and code page that is independently determined for each of those references must be identical or an error is returned.

Table 29. Untyped Expression Usage in Predicates

Untyped Expression Location	Data Type
Both operands of a comparison operator	VARCHAR(254)
One operand of a comparison operator, when the other operand is other than an untyped expression	The data type of the other operand
All operands of a BETWEEN predicate	VARCHAR(254)
Two operands of a BETWEEEN predicate	Same as that of the only typed expression
Only one operand of a BETWEEN predicate	Result of applying the "Rules for result data types" on all operands that are other than untyped expressions
All operands of an IN predicate, for example, ? IN (?,?,?)	VARCHAR(254)
The first operand of an IN predicate, when the right side is a fullselect, for example, IN (fullselect)	Data type of the selected column
The first operand of an IN predicate, when the right side is not a subselect;, for example, ? IN (?,A,B), or ? IN (A,?,B,?)	Result of applying the "Rules for result data types" on all operands of the IN list (operands to the right of the IN keyword) that are other than untyped expressions
Any or all operands of the IN list of the IN predicate, for example, A IN (?,B, ?)	Result of applying the "Rules for result data types" on all operands of the IN predicate (operands to the left and right of the IN keyword) that are other than untyped expressions
Both the operand in a row-value-expression of an IN predicate, and the corresponding result column of the fullselect, for example, (c1, ?) IN (SELECT c1, ? FROM ...)	VARCHAR(254)
Any operands in a row-value-expression of an IN predicate, for example, (c1,?) IN fullselect	Data type of the corresponding result column of the fullselect
Any select list items in a subquery if a row-value-expression is specified in an IN predicate, for example, (c1,c2) IN (SELECT?, c1, FROM ...)	Data type of the corresponding operand in the row-value-expression
All three operands of the LIKE predicate	Match expression (operand 1) and pattern expression (operand 2) are VARCHAR(32672); escape expression (operand 3) is VARCHAR(2)
The match expression of the LIKE predicate when either the pattern expression or the escape expression is other than an untyped expression	Either VARCHAR(32672) or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression
The pattern expression of the LIKE predicate when either the match expression or the escape expression is other than an untyped expression	Either VARCHAR(32672) or VARGRAPHIC(16336), depending on the data type of the first operand that is not an untyped expression; if the data type of the match expression is BLOB, the data type of the pattern expression is assumed to be BLOB(32672)

Determining data types of untyped expressions

Table 29. Untyped Expression Usage in Predicates (continued)

Untyped Expression Location	Data Type
The escape expression of the LIKE predicate when either the match expression or the pattern expression is other than an untyped expression	Either VARCHAR(2) or VARGRAPHIC(1), depending on the data type of the first operand that is not an untyped expression; if the data type of the match expression or pattern expression is BLOB, the data type of the escape expression is assumed to be BLOB(1)
Operand of the NULL predicate	VARCHAR(254)

Table 30. Untyped Expression Usage in Built-in Functions

Untyped Parameter Marker Location	Data Type
All arguments of COALESCE, MIN, MAX, NULLIF, or VALUE	Error
Any argument of COALESCE, MIN, MAX, NULLIF, or VALUE, when at least one argument is other than an untyped parameter marker	Result of applying the "Rules for result data types" on all arguments that are other than untyped parameter markers
First argument of DAYNAME	TIMESTAMP(12)
The argument of DIGITS	DECIMAL(31,6)
First argument of MONTHNAME	TIMESTAMP(12)
POSSTR (both arguments)	Both arguments are VARCHAR(32672)
POSSTR (one argument, when the other argument is a character data type)	VARCHAR(32672)
POSSTR (one argument, when the other argument is a graphic data type)	VARGRAPHIC(16336)
POSSTR (the search-string argument, when the other argument is a BLOB)	BLOB(32672)
First argument of SUBSTR	VARCHAR(32672)
Second and third argument of SUBSTR	INTEGER
First argument of SUBSTRB	VARCHAR(32672)
First argument of SUBSTR2	VARGRAPHIC(16336) if database supports graphic types; otherwise VARCHAR(32672)
First argument of SUBSTR4	VARCHAR(32672)
Second and third arguments of TRANSLATE	VARCHAR(32672) if the first argument is a character type; VARGRAPHIC(16336) if the first argument is a graphic type
Fourth argument of TRANSLATE	VARCHAR(1) if the first argument is a character type; VARGRAPHIC(1) if the first argument is a graphic type
The Second argument of TIMESTAMP	TIME
First argument of VARCHAR_FORMAT	TIMESTAMP(12)
First argument of TIMESTAMP_FORMAT	VARCHAR(254)
First argument of XMLVALIDATE	XML
First argument of XMLCOMMENT	VARCHAR(32672)
First argument of XMLTEXT	VARCHAR(32672)
Second argument of XMLPI	VARCHAR(32672)

Table 30. Untyped Expression Usage in Built-in Functions (continued)

Untyped Parameter Marker Location	Data Type
First argument of XMLSERIALIZE	XML
First argument of XMLDOCUMENT	XML
First argument of XMLXSROBJECTID	XML
All arguments of XMLCONCAT	XML
Second argument of TRIM_ARRAY	BIGINT
Array index of an ARRAY	BIGINT
Unary minus	DECFLOAT(34)
Unary plus	DECFLOAT(34)
Arguments of a aggregate function	Error
Arguments of XMLEXISTS, XMLELEMENT, XMLTABLE or XMLQUERY	Error
All other arguments of all other scalar functions	The data type of the parameter of the function definition as determined by function resolution. The length of the argument is derived based on Table 24 on page 223 in Function Resolution section.

Table 31. Untyped Expression Usage in User-defined Routines

Untyped Parameter Marker Location	Data Type
Argument of a function	The data type and length of the parameter, as defined when the function was created.
Argument of a method	Error
Argument of a procedure	The data type of the parameter, as defined when the procedure was created

Row expression

A row expression specifies a row of data that could have a specific user-defined row type or the built-in data type ROW.
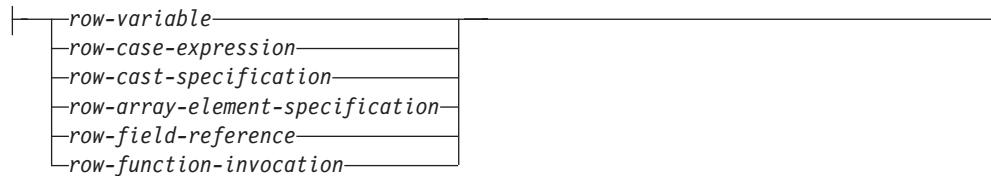
Authorization

The use of some of the row expressions may require having the appropriate authorization. For these row expressions, the privileges held by the authorization ID of the statement must include the following authorization:

- *row-variable.* For information about authorization considerations when *row-variable* is a global variable, see "Global variables".
- *row-function-invocation.* The authorization to execute the function. For information about authorization considerations, see "Function invocation" in the "Functions" topic
- *expression.* Authorizations might be required for the use of certain expressions referenced in a *row-expression.* For information about authorization considerations, see "Expressions".

Syntax

row-expression:

```
|-----+--row-variable------------------+-----------------------|
      |--row-case-expression----------|
      |--row-cast-specification-------|
      |--row-array-element-specification--|
      |--row-field-reference----------|
      |--row-function-invocation------|
```

Description

row-variable
    A variable that is defined with row type.

row-case-expression
    A case-expression that returns a row type.

row-cast-specification
    A CAST that returns a row type.

row-array-element-specification
    An array-element-specification of an array with row type elements.

row-field-reference
    A field-reference of a row where the field is also a row type

row-function-invocation
    A *function-invocation* of a user-defined function that has a return type that is a row type. The function could return a user-defined row type or the data type ROW with defined field names and field types.

Notes

- Row expressions can be used to generate a row within SQL PL contexts.

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given value, row, or group.

The following rules apply to all types of predicates:
- All values specified in a predicate must be compatible.
- An expression used in a basic, quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4000, a graphic string with a length attribute greater than 2000, or a LOB string of any size.
- The value of a host variable can be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, is done according to the rules for string conversions.
- Use of a structured type value is limited to the NULL predicate and the TYPE predicate.
- In a Unicode database, all predicates that accept a character or graphic string will accept any string type for which conversion is supported.

A fullselect is a form of the SELECT statement that, when used in a predicate, is also called a *subquery*.

Row-value-expression

The operand of several predicates (basic, quantified, and IN) can be expressed as a row value expression:

row-value-expression:
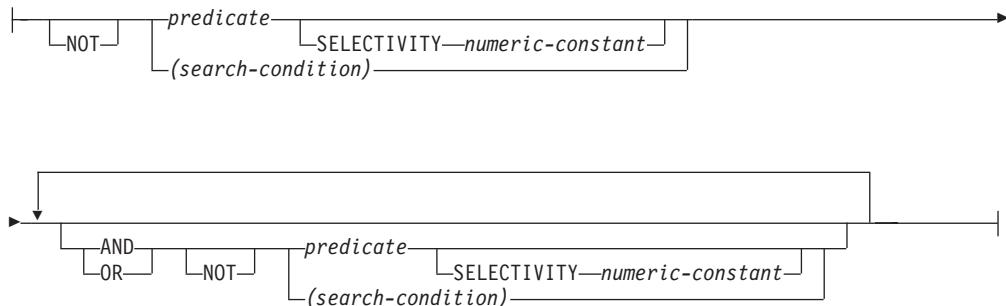
A row value expression returns a single row that consists of one or more fields. The field values can be specified as a list of expressions. The number of fields that are returned by the row value expression is equal to the number of expressions that are specified in the list.

Search conditions

A *search condition* specifies a condition that is "true," "false," or "unknown" about a given value, row, or group.

search-condition:

The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in Table 32, in which P and Q are any predicates:

Table 32. Truth Tables for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

MAJPROJ = 'MA2100' **AND** DEPTNO = 'D11' **OR** DEPTNO = 'B03' **OR** DEPTNO = 'E11'

1   2 or 3   2 or 3

MAJPROJ = 'MA2100' **AND** (DEPTNO = 'D11' **OR** DEPTNO = 'B03') **OR** DEPTNO = 'E11'

2   1   3
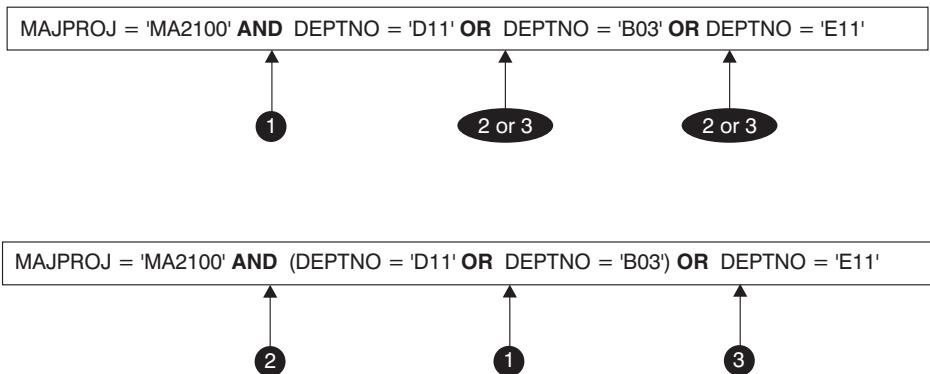
Figure 15. Search Conditions Evaluation Order

SELECTIVITY *numeric-constant*
> The SELECTIVITY clause is used to indicate what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified for the following predicates:
>
> - A user-defined predicate, regardless of the *DB2_SELECTIVITY* query compiler registry variable setting
> - A basic predicate in which at least one expression contains host variables or parameter markers. Specifying SELECTIVITY for this type of predicate applies only when the *DB2_SELECTIVITY* query compiler registry variable is set to YES.
>
> A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function myfunction is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:
>
> ```
>     SELECT *
>       FROM STORES
>       WHERE myfunction(parm,parm) = 1 SELECTIVITY 0.004
> ```
>
> The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.ROUTINES view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).
>
> A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:
>
> - the predicate specification is present in the CREATE FUNCTION statement
> - the UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
> - there is no negation (NOT operator)

Examples

In the following query, the within UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate.

```
    SELECT *
      FROM customers
      WHERE within(location, :sanJose) = 1 SELECTIVITY  0.2
```

However, the presence of `within` in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```
SELECT *
  FROM customers
  WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY  0.3
```

In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance from one store to another is computed by the radius of the city in which the customers live.
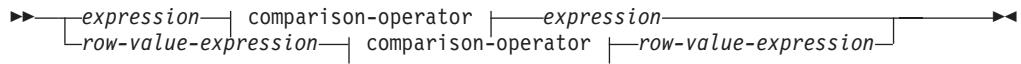
```
SELECT *
  FROM customers, stores
  WHERE distance(customers.loc, stores.loc) <
    CityRadius(stores.loc) SELECTIVITY 0.02
```

In the preceding query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

However, since the result produced by CityRadius is used as a range producer function, the user-defined predicate shown previously will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.

Basic predicate

A *basic predicate* compares two values or compares a set of values with another set of values.

```
►►──┬─expression───┬─ comparison-operator ─┬──expression──────┬──────────►◄
    └─row-value-expression─┤ comparison-operator ├─row-value-expression─┘
```

comparison-operator:

```
├──┬─ = ──────────────┬──────────────────────────────────┤
   │      (1)         │
   ├─ <> ─────────────┤
   ├─ < ──────────────┤
   ├─ > ──────────────┤
   │      (1)         │
   ├─ <= ─────────────┤
   │      (1)         │
   └─ >= ─────────────┘
```

Notes:

1   The following forms of the comparison operators are also supported in basic and quantified predicates: ^=, ^<, ^>, !=, !<, and !>. In code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

The six comparison operators can effectively be expressed based on just two of the comparison operators. If the predicate operands are x and y, then the other four comparison operators can be expressed using the following alternative predicates.

Table 33. Predicates and alternative predicates

Predicate	Alternative predicate
$x <> y$	NOT $(x = y)$
$x > y$	$y < x$
$x <= y$	$x < y$ OR $x = y$
$x >= y$	$y < x$ OR $x = y$

When the predicate operands are specified as an *expression*, the data types of the expressions must be comparable. If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

Table 34. Predicate evaluation with scalar operands

Predicate (with operand values x and y)	Boolean value	If and Only If...
$x = y$	is True	x is equal to y
$x < y$	is True	x is less than y
$x = y$	is False	x is not equal to y
$x < y$	is False	$x = y$ is True or $y < x$ is True

Basic predicate

When the predicate operands are specified as a *row-value-expression*, they must have the same number of fields and the data types of the corresponding fields of the operands must be comparable. The result of the comparison is based on comparisons of the corresponding fields in the *row-value-expression* operands.

Table 35. Predicate evaluation with row operands

Predicate (with operand values Rx and Ry that have fields Rxi and Ryi where $0 < i <$ *number of fields*)	Boolean value	If and Only If...
$Rx = Ry$	is True	All pairs of corresponding value expressions are equal ($Rxi = Ryi$ is True for all values of i).
$Rx < Ry$	is True	The first N pairs of corresponding value expressions are equal and the next pair has the left value expression less than the right value expression for some value of N ($Rxi = Ryi$ is True for all values of $i < n$ and $Rxn < Ryn$ is True for some value of n).
$Rx = Ry$	is False	At least one pair of corresponding value expressions are not equal (NOT ($Rxi = Ryi$) is True for some value of i).
$Rx < Ry$	is False	All pairs of corresponding value expressions are equal ($Rx = Ry$ is True) or the first N pairs of corresponding value expressions are equal and the next pair has the right value expression less than the left value expression for some value of N ($Rxi = Ryi$ is True for all values of $i < n$ and $Ryn < Rxn$ is True for some value of n).
Rx comparison operator Ry	is Unknown	The comparison is neither True nor False.
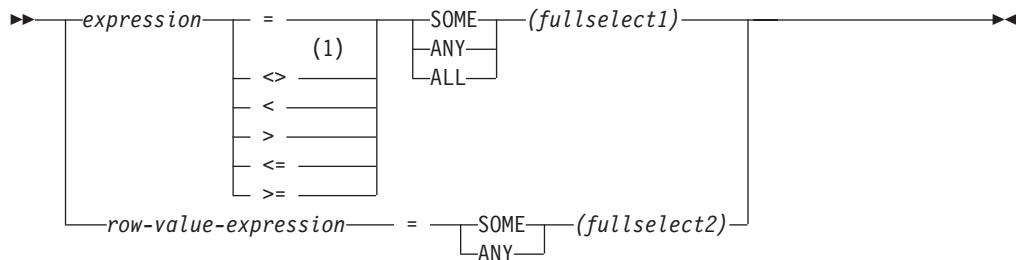
Examples:
```
EMPNO='528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
(YEARVAL, MONTHVAL) >= (2009, 10)
```

Quantified predicate

A *quantified predicate* compares a value or values with a collection of values.

Notes:

1   The following forms of the comparison operators are also supported in basic and quantified predicates: ^=, ^<, ^>, !=, !<, and !>. In code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported. All of these product-specific forms of the comparison operators are intended only to support existing SQL statements that use these operators, and are not recommended for use when writing new SQL statements.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

When ALL is specified:
- The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
- The result is false if the specified relationship is false for at least one value returned by the fullselect.
- The result is unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:
- The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
- The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
- The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.

Quantified predicate

TBL**AB**:

COL**A**	COL**B**
1	12
2	12
3	13
4	14
-	-

TBL**XY**:

COL**X**	COL**Y**
2	22
3	23

Figure 16. Tables for quantified predicate examples

Example 1

```
SELECT COLA FROM TBLAB
    WHERE COLA = ANY(SELECT COLX FROM TBLXY)
```

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

Example 2

```
SELECT COLA FROM TBLAB
    WHERE COLA > ANY(SELECT COLX FROM TBLXY)
```

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

```
SELECT COLA FROM TBLAB
    WHERE COLA > ALL(SELECT COLX FROM TBLXY)
```

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

```
SELECT COLA FROM TBLAB
    WHERE COLA > ALL(SELECT COLX FROM TBLXY
                          WHERE COLX<0)
```

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

Example 5

```
SELECT * FROM TBLAB
    WHERE (COLA,COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)
```

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

Example 6

```
SELECT * FROM TBLAB
    WHERE (COLA,COLB) = ANY (SELECT COLX,COLY-10 FROM TBLXY)
```

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

ARRAY_EXISTS

The ARRAY_EXISTS predicate tests for the existence of an array index in an array.

►►──ARRAY_EXISTS──(──*array-variable*──,──*array-index*──)────────────────────────►◄

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
> The data type of *array-index* must be assignable to the data type of the array index of the array. If *array-variable* is an ordinary array, then *array-index* must be assignable to INTEGER (SQLSTATE 428H1).

The result is true if *array-variable* includes an array index that is equal to *array-index*cast to the data type of the array index of *array-variable*; otherwise the result is false.

The result cannot be unknown; if either argument is null, the result is false.
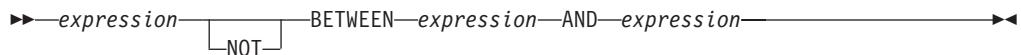
Example

- Assume that array variableRECENT_CALLS is defined as an ordinary array of array type PHONENUMBERS. The following IF statement tests if the recent calls list has reached the 40th saved call yet. If it has, the local Boolean variable EIGHTY_PERCENT is set to true:

```
IF (ARRAY_EXISTS(RECENT_CALLS, 40)) THEN
  SET EIGHTY_PERCENT = TRUE;
END IF
```

BETWEEN predicate

The BETWEEN predicate compares a value with a range of values.

```
►►─expression─┬──────┬─BETWEEN─expression─AND─expression────────────────►◄
              └─NOT──┘
```

If the data types of the operands are not the same, all values are converted to the data type that would result by applying the "Rules for result data types", except if the data types of all the operands are numeric, in which case no values are converted.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,
value1 < value2 OR value1 > value3.
```

The first operand (expression) cannot include a function that is not deterministic or has an external action (SQLSTATE 42845).

Examples

Example 1

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Results in all salaries between $20,000.00 and $40,000.00.

Example 2

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Assuming :HV1 is 5000, results in all salaries below $25,000.00 and above $40,000.00.

Cursor predicates

Cursor predicates are SQL keywords that can be used to determine the state of a cursor defined within the current scope. They provide a means for easily referencing whether a cursor is open, closed or if there are rows associated with the cursor.

►►──*cursor-variable-name*──IS─┬──────┬─┬─FOUND─┬──────────────────────────►◄
                              └─NOT──┘ └─OPEN──┘

cursor-variable-name
> The name of a SQL variable or SQL parameter of a cursor type.

IS Specifies that a cursor predicate property is to be tested.

NOT
> Specifies that the opposite value of testing the cursor predicate property is to be returned.

FOUND
> Specifies to check if the cursor contains rows after the execution of a FETCH statement. If the last FETCH statement executed was successful, and if the IS FOUND predicate syntax is used, the returned value is TRUE. If the last FETCH statement executed resulted in a condition where rows were not found, the result is false. The result is unknown when:
> - the value of cursor-variable-name is null
> - the underlying cursor of cursor-variable-name is not open
> - the predicate is evaluated before the first FETCH action was performed on the underlying cursor
> - the last FETCH action returned an error
>
> The IS FOUND predicate can be useful within a portion of SQL PL logic that loops and performs a fetch with each iteration. The predicate can be used to determine if rows remain to be fetched. It provides an efficient alternative to using a condition handler that checks for the error condition that is raised when no more rows remain to be fetched.
>
> When the NOT keyword is specified, so that the syntax is IS NOT FOUND, the result value is the opposite.

OPEN
> Specifies to check if the cursor is in an open state. If the cursor is in open and if the IS OPEN predicate syntax is used, the returned value is TRUE. This can be a useful predicate in cases where cursors are passed as parameters to functions and procedures. Before attempting to open the cursor, this predicate can be used to determine if the cursor is already open.
>
> When the NOT keyword is specified, so that the syntax is IS NOT OPEN, the result value is the opposite.

Notes
- A cursor predicate can only be used in statements within a compound SQL (compiled) statement (SQLSTATE 42818).

Example

The following script defines an SQL procedure that contains references to these
predicates as well as the prerequisite objects required to successfully compile and
call the procedure:

```
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)@

INSERT INTO T1 VALUES (1,1,1),(2,2,2),(3,3,3) @

CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)@

CREATE TYPE myCursorType AS myRowType CURSOR@


CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
  DECLARE C1 CURSOR;
  DECLARE lvarInt INT;

  SET count = -1;
  SET c1 = CURSOR FOR SELECT c1 FROM t1;

  IF (c1 IS NOT OPEN) THEN
     OPEN c1;
  ELSE
     set count = -2;
  END IF;

  SET count = 0;
  IF (c1 IS OPEN) THEN

    FETCH c1 INTO lvarInt;

    WHILE (c1 IS FOUND) DO
      SET count = count + 1;
      FETCH c1 INTO lvarInt;
    END WHILE;
  ELSE
     SET COUNT = 0;
  END IF;

END@

CALL p()@
```

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows.

►►──EXISTS──*(fullselect)*──►◄

The fullselect may specify any number of columns, and
- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.
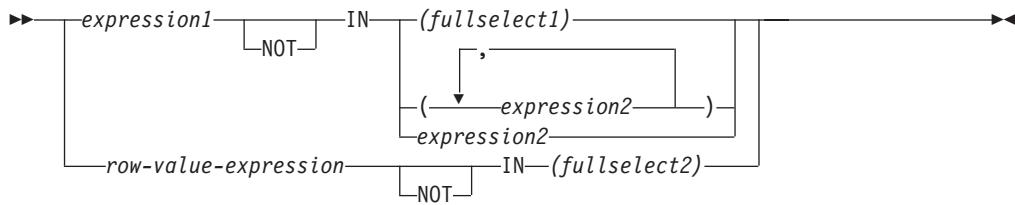
Example

```
EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```

IN predicate

The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:
  ```
  expression IN expression
  ```

  is equivalent to a basic predicate of the form:
  ```
  expression = expression
  ```
- An IN predicate of the form:
  ```
  expression IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:
  ```
  expression = ANY (fullselect)
  ```
- An IN predicate of the form:
  ```
  expression NOT IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:
  ```
  expression <> ALL (fullselect)
  ```
- An IN predicate of the form:
  ```
  expression IN (expressiona, expressionb, ..., expressionk)
  ```

  is equivalent to:
  ```
  expression = ANY (fullselect)
  ```

  where fullselect in the values-clause form is:
  ```
  VALUES (expressiona), (expressionb), ..., (expressionk)
  ```
- An IN predicate of the form:
  ```
  (expressiona, expressionb,..., expressionk) IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:
  ```
  (expressiona, expressionb,..., expressionk) = ANY (fullselect)
  ```

  Note that the operand on the left side of this form of these predicates is referred to as a *row-value-expression*.

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each field of the *row-value-expression* and its corresponding column of *fullselect2* in the IN predicate must be compatible. The rules for result data types can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary, the code page is determined by applying rules for string conversions to the IN list first, and then to the predicate, using the derived code page for the IN list as the second operand.

Examples

Example 1: The following condition evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

Example 2: The following condition evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

Example 3: Given the following information, this example evaluates to true if the specific value in the row of the COL_1 column matches any of the values in the list:

Table 36. IN Predicate example

Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

the two host variables will be converted to code page 850, based on the rules for string conversions.

Example 4: The following condition evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):
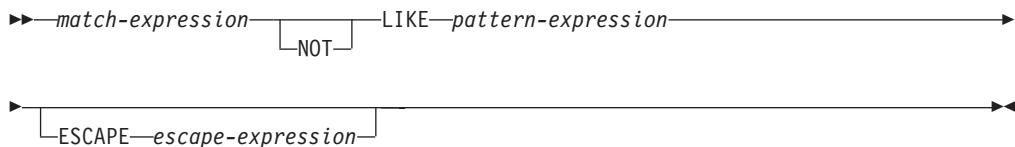
```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

Example 5: The following condition evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```

LIKE predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and the percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

```
►►─match-expression──┬─────┬──LIKE──pattern-expression─────────────────────────►
                     └─NOT─┘

►──┬──────────────────────────────────┬──────────────────────────────────────►◄
   └─ESCAPE──escape-expression─────────┘
```

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 variable
- A scalar function
- A large object locator
- A column name
- An expression concatenating any of the previously listed items

LIKE *pattern-expression*
An expression that specifies the string that is to be matched.

The expression can be specified in the same way as *match-expression* with the following restrictions:
- No element in the expression can be of type CLOB or DBCLOB. In addition it cannot be a BLOB file reference variable.
- The actual length of *pattern-expression* cannot be more than 32 672 bytes.

A **simple description** of the use of the LIKE predicate is that the pattern is used to specify the conformance criteria for values in the *match-expression*, where:
- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or the percent character in the pattern.

A **rigorous description** of the use of the LIKE predicate follows. Note that this description ignores the use of the *escape-expression*; its use is covered later.

- Let *m* denote the value of *match-expression* and let *p* denote the value of *pattern-expression*. The string *p* is interpreted as a sequence of the minimum number of substring specifiers so each character of *p* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

  The result of the predicate is unknown if *m* or *p* is the null value. Otherwise, the result is either true or false. The result is true if *m* and *p* are both empty strings or there exists a partitioning of *m* into substrings such that:

  - A substring of *m* is a sequence of zero or more contiguous characters and each character of *m* is part of exactly one substring.
  - If the *n*th substring specifier is an underscore, the *n*th substring of *m* is any single character.
  - If the *n*th substring specifier is a percent sign, the *n*th substring of *m* is any sequence of zero or more characters.
  - If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *m* is equal to that substring specifier and has the same length as that substring specifier.
  - The number of substrings of *m* is the same as the number of substring specifiers.

  Thus, if *p* is an empty string and *m* is not an empty string, the result is false. Similarly, it follows that if *m* is an empty string and *p* is not an empty string (except for a string containing only percent signs), the result is false.

  The predicate *m* NOT LIKE *p* is equivalent to the search condition NOT (*m* LIKE *p*).

When the *escape-expression* is specified, the *pattern-expression* must not contain the escape character identified by the *escape-expression*, except when immediately followed by the escape character, the underscore character, or the percent sign character (SQLSTATE 22025).

If the *match-expression* is a character string in an MBCS database, it can contain mixed data. In this case, the pattern can include both SBCS and non-SBCS characters. For non-Unicode databases, the special characters in the pattern are interpreted as follows:

- An SBCS halfwidth underscore refers to one SBCS character.
- A non-SBCS fullwidth underscore refers to one non-SBCS character.
- An SBCS halfwidth or non-SBCS fullwidth percent sign refers to zero or more SBCS or non-SBCS characters.

In a Unicode database, there is really no distinction between "single-byte" and "non-single-byte" characters. Although the UTF-8 format is a "mixed-byte" encoding of Unicode characters, there is no real distinction between SBCS and non-SBCS characters in UTF-8. Every character is a Unicode character, regardless of the number of bytes in UTF-8 format.

In a Unicode graphic column, every non-supplementary character, including the halfwidth underscore character (U&'\005F') and the halfwidth percent sign character (U&'\0025'), is two bytes in width. In a Unicode database, special characters in a pattern are interpreted as follows:

- For character strings, a halfwidth underscore character (X'5F') or a fullwidth underscore character (X'EFBCBF') refers to one Unicode character, and a

halfwidth percent sign character (X'25') or a fullwidth percent sign character (X'EFBC85') refers to zero or more Unicode characters.

- For graphic strings, a halfwidth underscore character (U&'\005F') or a fullwidth underscore character (U&'\FF3F') refers to one Unicode character, and a halfwidth percent sign character (U&'\0025') or a fullwidth percent sign character (U&'\FF05') refers to zero or more Unicode characters.
- To be recognized as special characters when a locale-sensitive UCA-based collation is in effect, the underscore character and the percent sign character must not be followed by non-spacing combining marks (diacritics). For example, the pattern U&'%\0300' (percent sign character followed by non-spacing combining grave accent) will be interpreted as a search for %̀ and not as a search for zero or more Unicode characters followed by a letter with a grave accent.

A Unicode supplementary character is stored as two graphic code points in a Unicode graphic column. To match a Unicode supplementary character in a Unicode graphic column, use one underscore if the database uses locale-sensitive UCA-based collation, and two underscores otherwise. To match a Unicode supplementary character in a Unicode character column, use one underscore for all collations. To match a base character with one or more trailing non-spacing combining characters, use one underscore if the database uses locale-sensitive UCA-based collation. Otherwise, use as many underscore characters as the number of non-spacing combining characters plus the base character.

escape-expression
This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:
- A constant
- A special register
- A global variable
- A host variable
- A scalar function with any of the previously mentioned operands
- An expression concatenating any of the previously listed items

with the restrictions that:
- No element in the expression can be of type CLOB or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- For character columns, the result of the expression must be one character, or a binary string containing exactly one byte (SQLSTATE 22019).
- For graphic columns, the result of the expression must be one character (SQLSTATE 22019).
- The result of the expression must not be a non-spacing combining character sequence (for example, U&'\0301', Combining Acute Accent).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:

- Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:
  - If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.
  - If n is even, S represents n/2 literal occurrences of the escape character. Unlike the case where n is odd, S could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that S is not part of a larger sequence of successive escape characters). If S is followed by an underscore or percent sign, that character has its special meaning.

Following is an illustration of the effect of successive occurrences of the escape character which, in this case, is the back slash (\).

Pattern string
    Actual Pattern

\%   A percent sign

\\%   A back slash followed by zero or more arbitrary characters

\\\% A back slash followed by a percent sign

The code page used in the comparison is based on the code page of the *match-expression* value.

- The *match-expression* value is never converted.
- If the code page of *pattern-expression* is different from the code page of *match-expression*, the value of *pattern-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (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.
- If the pattern specified in a LIKE predicate is a parameter marker, and a fixed-length character host variable is used to replace the parameter marker, the value specified for the host variable must have the correct length. If the correct length is not specified, the select operation will not return the intended results.

For example, if the host variable is defined as CHAR(10), and the value WYSE% is assigned to that host variable, the host variable is padded with blanks on assignment. The pattern used is:

```
'WYSE%     '
```

The database manager searches for all values that start with WYSE and that end with five blank spaces. If you want to search only for values that start with 'WYSE', assign a value of 'WSYE%%%%%' to the host variable.

- The pattern is matched using the collation of the database, unless either operand is defined as FOR BIT DATA, in which case the pattern is matched using a binary comparison.

Examples

- Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

    ```
    SELECT PROJNAME FROM PROJECT
      WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
    ```

- Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

    ```
    SELECT FIRSTNME FROM EMPLOYEE
      WHERE EMPLOYEE.FIRSTNME LIKE 'J_'
    ```

- Search for a string of any length, with a first character of 'J', in the FIRSTNME column of the EMPLOYEE table.

    ```
    SELECT FIRSTNME FROM EMPLOYEE
      WHERE EMPLOYEE.FIRSTNME LIKE 'J%'
    ```

- In the CORP_SERVERS table, search for a string in the LA_SERVERS column that matches the value in the CURRENT SERVER special register.

    ```
    SELECT LA_SERVERS FROM CORP_SERVERS
      WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER
    ```

- Retrieve all strings that begin with the character sequence '_\' in column A of table T.

    ```
    SELECT A FROM T
      WHERE T.A LIKE '\_\\%' ESCAPE '\'
    ```

- Use 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'OE')
    ```

- In a Unicode database defined with the case insensitive collation CLDR181_LEN_S1, find all names that start with 'Bill'.

    ```
    SELECT NAME FROM CUSTDATA WHERE NAME LIKE 'Bill%'
    ```

    This query returns the names 'Bill Smith', 'billy simon', and 'BILL JONES'.

NULL predicate

The NULL predicate tests for null values.

►►──*expression*──IS─┬──────┬──NULL──►◄
                  └─NOT─┘

The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

A row type value cannot be used as the operand in a NULL predicate, except as the qualifier of a field name. If *expression* is a row type, an error is returned (SQLSTATE 428H2).

Examples

```
PHONENO IS NULL
```

```
SALARY IS NOT NULL
```

Trigger event predicates

A trigger event predicate is used in a triggered action to test the event that activated the trigger. It is only valid in the triggered action of a compiled trigger definition (SQLSTATE 42601).

```
►►──┬─DELETING──┬──────────────────────────────────────────────────►◄
    ├─INSERTING─┤
    └─UPDATING──┘
```

DELETING
  True if the trigger was activated by a delete operation. False otherwise.

INSERTING
  True if the trigger was activated by an insert operation. False otherwise.

UPDATING
  True if the trigger was activated by an update operation. False otherwise.

Notes

- Trigger event predicates can be used anywhere in the triggered action of a CREATE TRIGGER statement that uses a compound SQL (compiled) statement as the *SQL-procedure-statement*. In other contexts the keywords will not be recognized and will attempt to resolve as column or variable names.

Example

The following trigger increments the number of employees each time a new person is hired (that is, each time a new row is inserted into the EMPLOYEE table); decrements the number of employees each time an employee leaves the company; and raises an error when an update occurs that would result in a salary increase greater than ten percent of the current salary, by using trigger event predicates in its conditions:

```
CREATE TRIGGER HIRED
  AFTER INSERT OR DELETE OR UPDATE OF SALARY ON EMPLOYEE
  REFERENCING NEW AS N OLD AS O FOR EACH ROW
    BEGIN
      IF INSERTING
      THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1;
      END IF;

      IF DELETING
      THEN UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1;
      END IF;

      IF (UPDATING AND (N.SALARY > 1.1 * O.SALARY))
      THEN SIGNAL SQLSTATE '75000' SET MESSAGE_TEXT = 'Salary increase>10%'
      END IF;
    END;
```
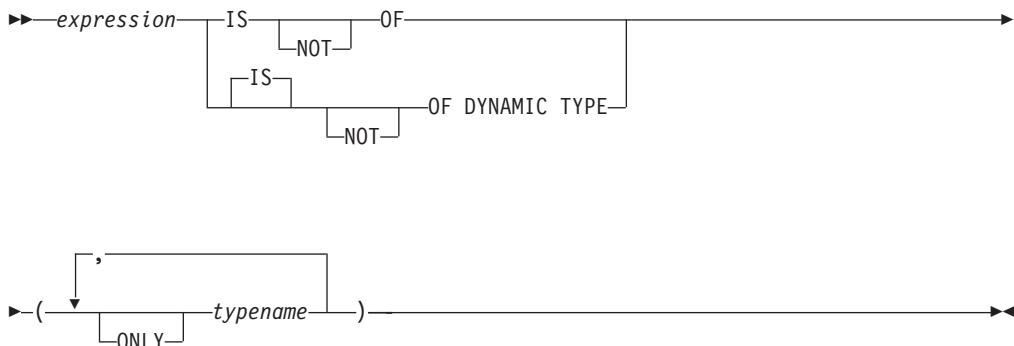
TYPE predicate

A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.

```
►►──expression──┬─IS─┬───────OF───────────────────────────────►
                │    └─NOT─┘                                    
                └─IS─┐                                          
                     │       ──OF DYNAMIC TYPE──               
                     └─NOT─┘                                    
```

```
►─(─┬────────────typename──┬─)──────────────────────────────►◄
    └─ONLY─┘                                                  
```

The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This might differ from the target type of an expression involving the reference, which is called the static type of the expression.

If the value of *expression* is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the *expression* is a subtype of one of the structured types specified by *typename*, otherwise the result is false. If ONLY precedes any *typename* the proper subtypes of that type are not considered.

If *typename* is not qualified, it is resolved using the SQL path. Each *typename* must identify a user-defined type that is in the type hierarchy of the static type of *expression* (SQLSTATE 428DU).

The DEREF function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of *expression* is the target type of the reference.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

Examples

A table hierarchy exists with root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table, ACTIVITIES, includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. The following example shows a type predicate that evaluates to true when a row corresponding to WHO_RESPONSIBLE is a manager:

```
DEREF (WHO_RESPONSIBLE) IS OF (MGR)
```

If a table contains a column EMPLOYEE of type EMP, EMPLOYEE may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```
EMPL IS OF (MGR)
```

returns true when EMPL is not null and is actually a manager.

VALIDATED predicate
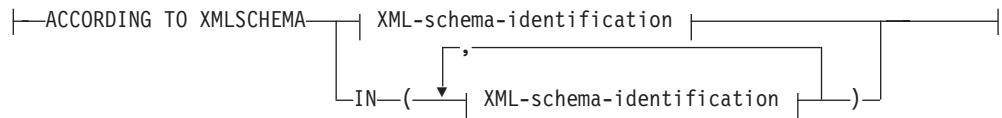
The VALIDATED predicate tests whether or not the value specified by *XML-expression* has been validated using the XMLVALIDATE function.

If the value specified is null, the result of the validation constraint is unknown; otherwise, the result of the validation constraint is either true or false. The value you specify must be of type XML.
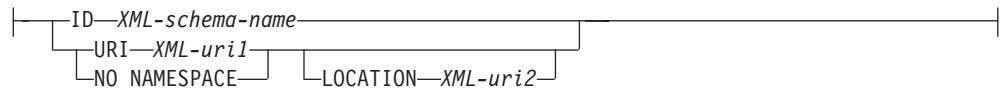
If the ACCORDING TO XMLSCHEMA clause is not specified, then XML schemas used for validation do not impact the result of the validation constraint.

```
►►─XML-expression─IS─┬─────┬─VALIDATED─┬────────────────────────┬─►◄
                     └─NOT─┘           └─ 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─┘
```

Description

XML-expression
: Specifies the XML value tested, where *XML-expression* can consist of an XML document, XML content, a sequence of XML nodes, an XML *column-name*, or an XML *correlation-name*.

  If an XML *column-name* is specified, the predicate evaluates whether or not XML documents associated with the specified column name have been validated.

  See "CREATE TRIGGER" for information about specifying correlation names of type XML as part of triggers.

IS VALIDATED or IS NOT VALIDATED
: Specifies the required validation state for the *XML-expression* operand.

  For a constraint that specifies IS VALIDATED to evaluate as true, the operand must have been validated. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

  For a constraint that specifies IS NOT VALIDATED to evaluate as false, the operand must be in an validated state. If an optional ACCORDING TO XMLSCHEMA clause includes one or several XML schemas, the operand must have been validated using one of the identified XML schemas.

according-to-clause
: Specifies one or several XML schemas against which the operand must or must

not have been validated. Only XML schemas previously registered with the XML schema repository may be specified.

ACCORDING TO XMLSCHEMA

> **ID** *XML-schema-name*
> Specifies an SQL identifier for the XML schema. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

> **URI** *XML-uri1*
> Specifies the target namespace URI of the XML schema. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

> **NO NAMESPACE**
> Specifies that the XML schema has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

> **LOCATION** *XML-uri2*
> Specifies the XML schema location URI of the XML schema. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

Examples

Example 1: Assume that column XMLCOL is defined in table T1. Retrieve only the XML values that have been validated by any XML schema.

```
SELECT XMLCOL FROM T1
  WHERE XMLCOL IS VALIDATED
```

Example 2: Assume that column XMLCOL is defined in table T1. Enforce the rule that values cannot be inserted or updated unless they have been validated.
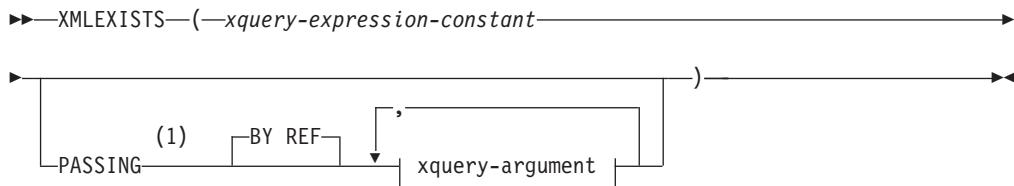
```
ALTER TABLE T1 ADD CONSTRAINT CK_VALIDATED
  CHECK (XMLCOL IS VALIDATED)
```

Example 3: Assume that you want to select only those rows from table T1 with XML column XMLCOL that have been validated with the XML schema URI http://www.posample.org.
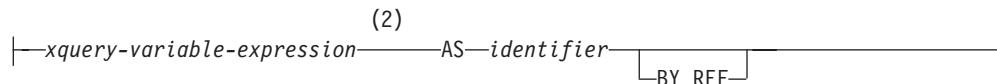
```
SELECT XMLCOL FROM T1
  WHERE XMLCOL IS VALIDATED
    ACCORDING TO XMLSCHEMA URI
    'http://www.posample.org'
```

XMLEXISTS predicate

The XMLEXISTS predicate tests whether an XQuery expression returns a sequence of one or more items.

```
►►─XMLEXISTS─(─xquery-expression-constant──────────────────────────────►

►──┬──────────────────────────────────────────────┬─)──────────────►◄
   │            (1)    ┌─BY REF─┐  ┌──────,──────┐  │
   └─PASSING───────────┴────────┴──▼─xquery-argument─┴──┘
```

xquery-argument:

```
                                        (2)
├──xquery-variable-expression──────AS─identifier─┬──────────┬──┤
                                                 └─BY REF─┘
```

Notes:

1   The data type cannot be DECFLOAT.

2   The data type of the expression cannot be DECFLOAT.

xquery-expression-constant
    Specifies an SQL character string constant that is interpreted as an XQuery expression. The constant string is converted directly to UTF-8 without conversion to the database or section code page. The XQuery expression executes using an optional set of input XML values, and returns an output sequence that is tested to determine the result of the XMLEXISTS predicate. The value for *xquery-expression-constant* must not be an empty string or a string of blank characters (SQLSTATE 10505).

PASSING
    Specifies input values and the manner in which these values are passed to the XQuery expression specified by *xquery-expression-constant*. By default, every unique column name that is in the scope where the function is invoked is implicitly passed to the XQuery expression using the name of the column as the variable name. If an *identifier* in a specified xquery-argument matches an in-scope column name, then the explicit xquery-argument is passed to the XQuery expression overriding that implicit column.

    BY REF
        Specifies that the default passing mechanism is by reference for any *xquery-variable-expression* of data type XML. When XML values are passed by reference, the XQuery evaluation uses the input node trees, if any, directly from the specified input expressions, preserving all properties, including the original node identities and document order. If two arguments pass the same XML value, node identity comparisons and document ordering comparisons involving some nodes contained between the two input arguments might refer to nodes within the same XML node tree.

        This clause has no impact on how non-XML values are passed. The non-XML values create a new copy of the value during the cast to XML.

    xquery-argument
        Specifies an argument that is to be passed to the XQuery expression

specified by *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:
- 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.

Example

```
SELECT c.cid FROM customer c
  WHERE XMLEXISTS('$d/*:customerinfo/*:addr[ *:city = "Aurora" ]'
    PASSING info AS "d")
```

XMLEXISTS predicate

Chapter 3. Built-in global variables

Built-in global variables are provided with the database manager and are used in SQL statements to retrieve scalar values associated with the variables.

As an example, the ROUTINE_TYPE global variable is referenced in an SQL statement to retrieve the current routine type.

For most built-in global variables, the authorization ID of any statement that retrieves the value of the global variable is required to have the READ privilege on the global variable or DATAACCESS authority. However, there are exceptions where an authorization ID with other database authorities also has access to the global variable. Exceptions to the authorization required to retrieve the value of a global variable are specified within the descriptions of the built-in global variables.

In a non-restrictive database, the READ privilege is granted to PUBLIC for most built-in global variables at creation time. Exceptions to granting this privilege are specified within the descriptions of the built-in global variables.

Examples
- To access the global variable CLIENT_HOST, run the following query:

  ```
  VALUES SYSIBM.CLIENT_HOST
  ```

  This query returns the host name of the current client:

  ```
  1
  ----------
  hotellnx93
  ```

  An alternative way to call the global variable is to use it in a SELECT statement:

  ```
  SELECT C1, C2
    FROM T1
    WHERE C3 = CLIENT_HOST
  ```
- For read/write global variables, set the value with the "SET variable" statement:

  ```
  SET NLS_STRING_UNITS = 'CODEUNITS32'
  ```

CLIENT_HOST global variable

This built-in global variable contains the host name of the current client, as returned by the operating system.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(255).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the client connection originated from an application running on the local system, the value of the variable is NULL. The database manager obtains the client IP address from the network when the connection is accepted. The client host name is obtained from the client IP address by invoking the TCP/IP GetAddrInfo function. If the processes did not originate from a remote system using TCP/IP, the value of the variable is NULL.

CLIENT_IPADDR global variable

This built-in global variable contains the IP address of the current client, as returned by the operating system.

This global variable has the following characteristics:

- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

The value of the CLIENT_IPADDR global variable is NULL if the client did not connect by using the TCP/IP or SSL protocol.

CLIENT_ORIGUSERID global variable

This built-in global variable contains the original user identifier, as supplied by an application, usually from a multiple-tier server environment.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(1024).
- The schema is SYSIBM.
- The scope of this global variable is session.

If an application does not supply a value, the value of the CLIENT_ORIGUSERID global variable is NULL.

The value that is used for CLIENT_ORIGUSERID can be set with the following APIs:
- the originalUser variable of the getDB2Connection (trusted reuse) and reuseDB2Connection (trusted connection reuse) APIs of the IBM® Data Server Driver for JDBC and SQLJ.

Notes

No privileges are granted to PUBLIC when the CLIENT_ORIGUSERID global variable is created.

CLIENT_USRSECTOKEN global variable

This built-in global variable contains a security token, as supplied by an application, usually from a multiple-tier server environment.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is BLOB(4K).
- The schema is SYSIBM.
- The scope of this global variable is session.

If an application does not supply a value, the value of the CLIENT_USRSECTOKEN global variable is NULL.

The value that is used for CLIENT_USERSECTOKEN can be set with the following APIs:
- the userSecToken variable of the getDB2Connection (trusted reuse) and reuseDB2Connection (trusted connection reuse) APIs of the IBM® Data Server Driver for JDBC and SQLJ.

Notes

No privileges are granted to PUBLIC when the CLIENT_USRSECTOKEN global variable is created.

MON_INTERVAL_ID global variable

This built-in global variable contains the identifier for the current monitoring interval.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is BIGINT.
- The schema is SYSIBM.
- The scope of this global variable is database.

The MON_INTERVAL_ID global variable facilitates the collection and aggregation of monitoring data by external monitoring applications. The value of this global variable is intended to be set by monitoring tools. However, you can increment the value by using the MON_INCREMENT_INTERVAL_ID procedure.

The value of the MON_INTERVAL_ID global variable is 0 if there is no current monitoring interval.

The **mon_interval_id** monitor element contains the value of the monitor interval ID at the time that the monitoring data was captured. You can use the value of the **mon_interval_id** monitor element to correlate data that you gather over a specific monitoring interval.

Notes

In addition to the typical authorization groups or IDs that can read built-in global variables, an authorization ID with either DBADM or SQLADM authority has read access to the MON_INTERVAL_ID global variable.

NLS_STRING_UNITS global variable

This built-in global variable specifies the default string units that are used when defining character and graphic data types in a Unicode database.

This global variable has the following characteristics:

- It is a read/write variable, with values maintained by the user.
- The type is VARCHAR(11 OCTETS).
- The schema is SYSIBM.
- The scope of this global variable is session.
- It has a default value of NULL.

This global variable is only applicable to a Unicode database and must have a value of NULL, 'SYSTEM', or 'CODEUNITS32' (SQLSTATE 42815).

NULL The STRING_UNITS database configuration parameter is used to determine the default string units

SYSTEM

CHAR, VARCHAR, and CLOB data types defined without specifying the CODEUNITS32 keyword will default to OCTETS.

GRAPHIC, VARGRAPHIC, and DBCLOB data types defined without specifying the CODEUNITS32 keyword will default to CODEUNITS16.

CODEUNITS32

CHAR, VARCHAR, and CLOB data types defined without specifying the OCTETS keyword will default to CODEUNITS32.

GRAPHIC, VARGRAPHIC, and DBCLOB data types defined without specifying the CODEUNITS16 keyword will default to CODEUNITS32.

PACKAGE_NAME global variable

This built-in global variable contains the name of the currently executing package.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

In a nested execution scenario, where one package invokes another, the PACKAGE_NAME global variable contains the name of the immediate package context. For example, if package A checks the value of the PACKAGE_NAME variable, the value is A. If package A invokes package B and package B checks the value of the PACKAGE_NAME variable, the value is B.

PACKAGE_SCHEMA global variable

This built-in global variable contains the schema name of the currently executing package.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

In a nested execution scenario, where one package invokes another, the PACKAGE_SCHEMA global variable contains the schema name of the immediate package context. For example, if package X.A checks the value of the PACKAGE_SCHEMA variable, the schema value is X. If package X.A invokes package Y.B and package B checks the value of the PACKAGE_SCHEMA variable, the schema value is Y.

PACKAGE_VERSION global variable

This built-in global variable contains the version identifier of the currently executing package.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(64).
- The schema is SYSIBM.
- The scope is session.

If the current executing package does not have a version identifier, the value is NULL.

In a nested execution scenario, where one package invokes another, the PACKAGE_VERSION global variable contains the version identifier of the immediate package context. For example, if package A checks the value of the PACKAGE_VERSION variable, then the value is 1.0. If package A invokes package B and package B checks the value of the PACKAGE_VERSION variable, then the value is 1.8.

ROUTINE_MODULE global variable

This built-in global variable contains the module name of the currently executing routine.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the currently executing routine does not belong to a module or if the variable is referenced outside a routine execution context, the value of the ROUTINE_MODULE global variable is NULL.

Notes

The value of the ROUTINE_MODULE global variable is set only for procedures and compiled functions: the value always reflects the name of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_SCHEMA global variable

This built-in global variable contains the schema name of the currently executing routine.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_SCHEMA global variable is referenced outside a routine execution context, the value of the variable is NULL.

Notes

The value of the ROUTINE_SCHEMA global variable is set only for procedures and compiled functions: the value always reflects the schema name of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_SPECIFIC_NAME global variable

This built-in global variable contains the specific name of the currently executing routine.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_SPECIFIC_NAME global variable is referenced outside a routine execution context, the value of the variable is NULL.

Notes

The value of the ROUTINE_SPECIFIC_NAME global variable is set only for procedures and compiled functions: the value always reflects the specific name of the currently executing routine.

The value is not changed for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

ROUTINE_TYPE global variable

This built-in global variable contains the type of the currently executing routine.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is CHAR(1).
- The schema is SYSIBM.
- The scope of this global variable is session.

If the ROUTINE_TYPE global variable is referenced outside a routine execution context, the value of the variable is NULL.

The value of the ROUTINE_TYPE global variable is P for a stored procedure and F for a function.

Notes

The value of the ROUTINE_TYPE global variable is set only for procedures and compiled functions: the value always reflects the type of the currently executing routine.

The value does not change for inline functions or methods: the value remains the same as it was when the inline function or method was invoked.

TRUSTED_CONTEXT global variable

This built-in global variable contains the name of the trusted context that was matched to establish the current trusted connection.

This global variable has the following characteristics:
- It is read-only, with values maintained by system.
- The type is VARCHAR(128).
- The schema is SYSIBM.
- The scope is session.

If no trusted connection is established, the value of the TRUSTED_CONTEXT global variable is NULL.

Chapter 4. Built-in functions

Built-in functions are functions provided with the database manager and are classified as aggregate functions, scalar functions, or table functions.

This topic lists the supported built-in functions classified by type:
- Aggregate functions (Table 37 on page 346)
- Array functions (Table 38 on page 347)
- Cast scalar functions (Table 39 on page 347)
- Datetime scalar functions (Table 40 on page 348)
- Miscellaneous scalar functions (Table 41 on page 350)
- Numeric scalar functions (Table 42 on page 351)
- Partitioning scalar functions (Table 43 on page 353)
- Security scalar functions (Table 44 on page 353)
- String scalar functions (Table 45 on page 353)
- Table functions (Table 46 on page 355)
- XML functions (Table 47 on page 356)

The "OLAP specification" topic documents the following OLAP functions which are sometimes referred to as built-in functions:
- FIRST_VALUE and LAST_VALUE
- LAG and LEAD
- RANK and DENSE_RANK
- RATIO_TO_REPORT
- ROW_NUMBER

There are additional built-in functions documented under the following headings:
- ADMIN_CMD procedure and associated SQL routines
- Audit routines and procedures
- Configuration SQL routines and views
- DB2 pureScale instance views
- Environment views
- Explain routines
- Monitor routines
- MQSeries® SQL routines
- Security SQL routines and views
- Snapshot SQL routines and views
- SQL procedures SQL routines
- Workload management routines
- Miscellaneous SQL routines and views

For details about these additional built-in functions, see "Supported built-in SQL routines and views" in *Administrative Routines and Views*.

Built-in functions

Table 37. Aggregate functions

Function	Description
"ARRAY_AGG" on page 359	Aggregates a set of elements into an array.
"AVG" on page 363	Returns the average of a set of numbers.
"CORRELATION" on page 365	Returns the coefficient of correlation of a set of number pairs.
"COUNT" on page 366	Returns the number of rows or values in a set of rows or values.
"COUNT_BIG" on page 367	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 369	Returns the covariance of a set of number pairs.
"GROUPING" on page 370	Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.
"LISTAGG" on page 372	Aggregates a set string elements into one string by concatenating the strings.
"MAX" on page 374	Returns the maximum value in a set of values.
"MIN" on page 376	Returns the minimum value in a set of values.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_R2 aggregate function returns the coefficient of determination for the regression.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_SLOPE aggregate function returns the slope of the line.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.

Table 37. Aggregate functions (continued)

Function	Description
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)" on page 377	The REGR_SYY aggregate function returns quantities used to compute diagnostic statistics.
"STDDEV" on page 380	Returns the standard deviation of a set of numbers.
"SUM" on page 381	Returns the sum of a set of numbers.
"VARIANCE" on page 382	Returns the variance of a set of numbers.
"XMLAGG" on page 383	Returns an XML sequence containing an item for each non-null value in a set of XML values.
"XMLGROUP" on page 385	Returns an XML value with a single XQuery document node containing one top-level element node.

Table 38. Array functions

Function	Description
"ARRAY_AGG" on page 359	Aggregates a set of elements into an array.
"ARRAY_DELETE" on page 393	Deletes an element or range of elements from an associative array.
"ARRAY_FIRST" on page 394	Returns the smallest array index value of the array.
"ARRAY_LAST" on page 395	Returns the largest array index value of the array.
"ARRAY_NEXT" on page 396	Returns the next larger array index value for an array relative to the specified array index argument.
"ARRAY_PRIOR" on page 397	Returns the next smaller array index value for an array relative to the specified array index argument.
"CARDINALITY" on page 408	Returns a value of type BIGINT representing the number of elements of an array
"MAX_CARDINALITY" on page 539	Returns a value of type BIGINT representing the maximum number of elements that an array can contain.
"TRIM_ARRAY" on page 677	Returns a value with the same array type as *array-variable* but with the cardinality reduced by the value of *numeric-expression*.
"UNNEST" on page 764	Returns a result table that includes a row for each element of the specified array.

Table 39. Cast scalar functions

Function	Description
"BIGINT" on page 403	Returns a 64-bit integer representation of a value in the form of an integer constant.
"BLOB" on page 407	Returns a BLOB representation of a string of any type.
"CHAR" on page 410	Returns a CHARACTER representation of a value.
"CLOB" on page 420	Returns a CLOB representation of a value.
"DATE" on page 432	Returns a DATE from a value.

Built-in functions

Table 39. Cast scalar functions (continued)

Function	Description
"DBCLOB" on page 440	Returns a DBCLOB representation of a string.
"DECFLOAT" on page 443	Returns the decimal floating-point representation of a value.
"DECIMAL or DEC" on page 447	Returns a DECIMAL representation of a value.
"DOUBLE_PRECISION or DOUBLE" on page 459	Returns the floating-point representation of a value.
EMPTY_BLOB, EMPTY_CLOB, and EMPTY_DBCLOB scalar functions	Return a zero-length value of the associated data type.
"FLOAT" on page 469	Returns a FLOAT representation of a value.
"GRAPHIC" on page 474	Returns a GRAPHIC representation of a string.
"INTEGER or INT" on page 502	Returns an INTEGER representation of a value.
"NCHAR" on page 554	Returns a fixed-length national character string representation of a value.
"NCLOB" on page 556	Returns an NCLOB representation of a national character string.
"NVARCHAR" on page 557	Returns a varying-length national character string representation of a value.
"REAL" on page 583	Returns the single-precision floating-point representation of a value.
"SMALLINT" on page 623	Returns a SMALLINT representation of a value.
"TIME" on page 649	Returns a TIME from a value.
"TIMESTAMP" on page 650	Returns a TIMESTAMP from a value or a pair of values.
"TO_CLOB" on page 663	Returns a CLOB representation of a character string type.
"TO_NCLOB" on page 666	Returns an NCLOB representation of a character string.
"VARCHAR" on page 692	Returns a VARCHAR representation of a value.
"VARGRAPHIC" on page 709	Returns a VARGRAPHIC representation of a value.

Table 40. Datetime scalar functions

Function	Description
"ADD_MONTHS" on page 391	Returns a datetime value that represents *expression* plus a specified number of months.
"DAY" on page 433	Returns the day part of a value.
"DAYNAME" on page 434	Returns a character string containing the name of the day (for example, Friday) for the day portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.
"DAYOFWEEK" on page 436	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday.
"DAYOFWEEK_ISO" on page 437	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday.
"DAYOFYEAR" on page 438	Returns the day of the year from a value.

Table 40. Datetime scalar functions (continued)

Function	Description
"DAYS" on page 439	Returns an integer representation of a date.
"EXTRACT" on page 467	Returns a portion of a date or timestamp based on the arguments.
"HOUR" on page 487	Returns the hour part of a value.
"JULIAN_DAY" on page 504	Returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.
"LAST_DAY" on page 505	Returns a datetime value that represents the last day of the month of the argument.
"MICROSECOND" on page 540	Returns the microsecond part of a value.
"MIDNIGHT_SECONDS" on page 541	Returns an integer value representing the number of seconds between midnight and a specified time value.
"MINUTE" on page 543	Returns the minute part of a value.
"MONTH" on page 547	Returns the month part of a value.
"MONTHNAME" on page 548	Returns a character string containing the name of the month (for example, January) for the month portion of expression, based on locale-name or the value of the special register CURRENT LOCALE LC_TIME.
"MONTHS_BETWEEN" on page 550	Returns an estimate of the number of months between *expression1* and *expression2*.
"NEXT_DAY" on page 559	Returns a datetime value that represents the first weekday, named by *string-expression*, that is later than the date in *expression*.
"QUARTER" on page 579	Returns an integer that represents the quarter of the year in which a date resides.
"ROUND" on page 600	Returns a datetime value, rounded to the unit specified by *format-string*.
"ROUND_TIMESTAMP" on page 606	Returns a timestamp that is the *expression* rounded to the unit specified by the *format-string*.
"SECOND" on page 618	Returns the seconds part of a value.
"TIMESTAMP_FORMAT" on page 652	Returns a timestamp from a character string (*argument1*) that has been interpreted using a format template (*argument2*).
"TIMESTAMP_ISO" on page 659	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 660	Returns an estimated number of intervals of type *argument1*, based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR.
"TO_CHAR" on page 662	Returns a CHARACTER representation of a timestamp.
"TO_DATE" on page 664	Returns a timestamp from a character string.

Built-in functions

Table 40. Datetime scalar functions (continued)

Function	Description
"TO_NCHAR" on page 665	Returns a national character representation of an input expression that has been formatted using a character template.
"TO_TIMESTAMP" on page 669	Returns a timestamp that is based on the interpretation of the input string using the specified format.
"TRUNCATE or TRUNC" on page 680	Returns a datetime value, truncated to the unit specified by *format-string*.
"TRUNC_TIMESTAMP" on page 678	Returns a timestamp that is the *expression* truncated to the unit specified by the *format-string*.
"VARCHAR_FORMAT" on page 700	Returns a CHARACTER representation of a timestamp (*argument1*), formatted according to a template (*argument2*).
"WEEK" on page 718	Returns the week of the year from a value, where the week starts with Sunday.
"WEEK_ISO" on page 719	Returns the week of the year from a value, where the week starts with Monday.
"YEAR" on page 761	Returns the year part of a value.

Table 41. Miscellaneous scalar functions

Function	Description
"BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT" on page 405	These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.
"COALESCE" on page 421	Returns the first argument that is not null.
"CURSOR_ROWCOUNT" on page 430	Returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.
"DECODE" on page 451	Compares each specified *expression2* to *expression1*. If *expression1* is equal to *expression2*, or both *expression1* and *expression2* are null, the value of the following *result-expresssion* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned; otherwise a null value is returned.
"DEREF" on page 456	Returns an instance of the target type of the reference type argument.
"EVENT_MON_STATE" on page 465	Returns the operational state of particular event monitor.
"GREATEST" on page 481	Returns the maximum value in a set of values.
"HEX" on page 484	Returns a hexadecimal representation of a value.
"IDENTITY_VAL_LOCAL" on page 488	Returns the most recently assigned value for an identity column.
"LEAST" on page 509	Returns the minimum value in a set of values.
"LENGTH" on page 513	Returns the length of a value.
"MAX" on page 538	Returns the maximum value in a set of values.
"MIN" on page 542	Returns the minimum value in a set of values.
"NULLIF" on page 562	Returns a null value if the arguments are equal; otherwise, it returns the value of the first argument.

Table 41. Miscellaneous scalar functions (continued)

Function	Description
"NVL" on page 563	Returns the first argument that is not null.
"RAISE_ERROR" on page 581	Raises an error in the SQLCA. The sqlstate that is to be returned is indicated by *argument1*. The second argument contains any text that is to be returned.
"REC2XML" on page 585	Returns a string formatted with XML tags, containing column names and column data.
"RID_BIT and RID" on page 595	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 644	Returns an unqualified name of a table or view based on the object name specified in *argument1*, and the optional schema name specified in *argument2*. The returned value is used to resolve aliases.
"TABLE_SCHEMA" on page 645	Returns the schema name portion of a two-part table or view name (given by the object name in *argument1* and the optional schema name in *argument2*). The returned value is used to resolve aliases.
"TYPE_ID" on page 683	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 684	Returns the unqualified name of the dynamic data type of the argument.
"TYPE_SCHEMA" on page 685	Returns the schema name of the dynamic data type of the argument.
"VALUE" on page 691	Returns the first argument that is not null.

Table 42. Numeric scalar functions

Function	Description
"ABS or ABSVAL" on page 389	Returns the absolute value of a number.
"ACOS" on page 390	Returns the arc cosine of a number, in radians.
"ASIN" on page 399	Returns the arc sine of a number, in radians.
"ATAN" on page 400	Returns the arc tangent of a number, in radians.
"ATANH" on page 402	Returns the hyperbolic arc tangent of a number, in radians.
"ATAN2" on page 401	Returns the arc tangent of x and y coordinates as an angle expressed in radians.
"CEILING or CEIL" on page 409	Returns the smallest integer value that is greater than or equal to a number.
"COMPARE_DECFLOAT" on page 424	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 427	Returns the cosine of a number.
"COSH" on page 428	Returns the hyperbolic cosine of a number.

Built-in functions

Table 42. Numeric scalar functions (continued)

Function	Description
"COT" on page 429	Returns the cotangent of the argument, where the argument is an angle expressed in radians.
"DECFLOAT_FORMAT" on page 445	Returns a DECFLOAT(34) from a character string.
"DEGREES" on page 455	Returns the number of degrees of an angle.
"DIGITS" on page 458	Returns a character-string representation of the absolute value of a number.
"EXP" on page 466	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument.
"FLOOR" on page 470	Returns the largest integer value that is less than or equal to a number.
"LN" on page 518	Returns the natural logarithm of a number.
"LOG10" on page 526	Returns the common logarithm (base 10) of a number.
"MOD" on page 544	Returns the remainder of the first argument divided by the second argument.
"MOD (SYSFUN schema)" on page 546	Returns the remainder of the first argument divided by the second argument.
"MULTIPLY_ALT" on page 552	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 561	Returns a decimal floating-point value that is the result of the argument set to its simplest form.
"POWER" on page 576	Returns the result of raising the first argument to the power of the second argument.
"QUANTIZE" on page 577	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 580	Returns the number of radians for an argument that is expressed in degrees.
"RAND" on page 582	Returns a random number.
"ROUND" on page 600	Returns a numeric value that has been rounded to the specified number of decimal places.
"SIGN" on page 620	Returns the sign of a number.
"SIN" on page 621	Returns the sine of a number.
"SINH" on page 622	Returns the hyperbolic sine of a number.
"SQRT" on page 626	Returns the square root of a number.
"TAN" on page 647	Returns the tangent of a number.
"TANH" on page 648	Returns the hyperbolic tangent of a number.
"TO_NUMBER" on page 667	Returns a DECFLOAT(34) from a character string.
"TOTALORDER" on page 670	Returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.
"TRUNCATE or TRUNC" on page 680	Returns a number value that has been truncated at a specified number of decimal places.

Table 43. Partitioning scalar functions

Function	Description
"DATAPARTITIONNUM" on page 431	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 441	Returns the database partition number of the row. The argument is any column name within the table.
"HASHEDVALUE" on page 482	Returns the distribution map index (0 to 32767) of the row. The argument is a column name within a table.

Table 44. Security scalar functions

Function	Description
"SECLABEL" on page 614	Returns an unnamed security label.
"SECLABEL_BY_NAME" on page 615	Returns a specific security label.
"SECLABEL_TO_CHAR" on page 616	Accepts a security label and returns a string that contains all elements in the security label.
VERIFY_GROUP_ FOR_USER	Returns a value that indicates whether the groups associated with **authorization-id-expression** is in the group names specified by the list of **group-name-expression** arguments.
"VERIFY_ROLE_FOR_USER" on page 716	Returns a value that indicates whether any of the roles associated with **authorization-id-expression** are in, or contain any of, the role names specified by the list of **role-name-expression** arguments.
VERIFY_TRUSTED_ CONTEXT_ROLE_ FOR_USER	Returns a value that indicates whether **authorization-id-expression** has acquired a role under a trusted connection associated with some trusted context and that role is in, or contained in any of, the role names specified by the list of **role-name-expression** arguments.

Table 45. String scalar functions

Function	Description
"ASCII" on page 398	Returns the ASCII code value of the leftmost character of the argument as an integer.
"CHARACTER_LENGTH" on page 417	Returns the length of an expression in the specified *string-unit*.
"CHR" on page 419	Returns the character that has the ASCII code value specified by the argument.
"COLLATION_KEY_BIT" on page 422	Returns a VARCHAR FOR BIT DATA string representing the collation key of the specified *string-expression* in the specified *collation-name*.
"CONCAT" on page 426	Returns a string that is the concatenation of two strings.
"DECRYPT_BIN and DECRYPT_CHAR" on page 453	Returns a value that is the result of decrypting encrypted data using a password string.
"DIFFERENCE" on page 457	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.

Built-in functions

Table 45. String scalar functions (continued)

Function	Description
"ENCRYPT" on page 462	Returns a value that is the result of encrypting a data string expression.
"GENERATE_UNIQUE" on page 471	Returns a bit data character string that is unique compared to any other execution of the same function.
"GETHINT" on page 473	Returns the password hint if one is found.
"INITCAP" on page 492	Returns a string with the first character of each word converted to uppercase and the rest to lowercase.
"INSERT" on page 494	Returns a string, where *argument3* bytes have been deleted from *argument1* (beginning at *argument2*), and *argument4* has been inserted into *argument1* (beginning at *argument2*).
"INSTR" on page 498	Returns the starting position of a string within another string.
"INSTRB" on page 501	Returns the starting position, in bytes, of a string within another string.
"LCASE" on page 506	Returns a string in which all the SBCS characters have been converted to lowercase characters.
"LCASE (locale sensitive)" on page 507	Returns a string in which all characters have been converted to lowercase characters using the rules from the Unicode standard associated with the specified locale.
"LCASE (SYSFUN schema)" on page 508	Returns a string in which all the SBCS characters have been converted to lowercase characters.
"LOWER (locale sensitive)" on page 530	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 510	Returns the leftmost characters from a string.
"LOCATE" on page 519	Returns the starting position of one string within another string.
"LOCATE_IN_STRING" on page 523	Returns the starting position of the first occurrence of one string within another string.
"LOWER" on page 529	Returns a string in which all the characters have been converted to lowercase characters.
"LPAD" on page 532	Returns a string that is padded on the left with the specified character, or with blanks.
"LTRIM" on page 535	Removes blanks from the beginning of a string expression.
"LTRIM (SYSFUN schema)" on page 537	Removes blanks from the beginning of a string expression.
"OCTET_LENGTH" on page 565	Returns the length of an expression in octets (bytes).
"OVERLAY" on page 566	Returns a string in which, beginning at *start* in the specified *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.
"POSITION" on page 571	Returns the starting position of *argument2* within *argument1*.
"POSSTR" on page 574	Returns the starting position of one string within another string.
"REPEAT" on page 589	Returns a character string composed of *argument1* repeated *argument2* times.

Table 45. String scalar functions (continued)

Function	Description
"REPLACE" on page 590	Replaces all occurrences of *argument2* in *argument1* with *argument3*.
"REPLACE (SYSFUN schema)" on page 594	Replaces all occurrences of *expres sion2* in *expression1* with *e xpression3*.
"RIGHT" on page 597	Returns the rightmost characters from a string.
"RPAD" on page 608	Returns a string that is padded on the right with the specified character, string, or with blanks.
"RTRIM" on page 611	Removes blanks from the end of a string expression.
"RTRIM (SYSFUN schema)" on page 613	Removes blanks from the end of a string expression.
"SOUNDEX" on page 624	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 625	Returns a character string that consists of a specified number of blanks.
"STRIP" on page 627	Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.
"SUBSTR" on page 628	Returns a substring of a string.
"SUBSTRB" on page 639	Returns a substring of a string.
"SUBSTRING" on page 642	Returns a substring of a string.
"TO_SINGLE_BYTE" on page 668	Returns a string in which multi-byte characters are converted to the equivalent single-byte character where an equivalent character exists.
"TRANSLATE" on page 672	Returns a string in which one or more characters in a string are converted to other characters.
"TRIM" on page 675	Removes leading or trailing blanks or other specified leading or trailing characters from a string expression.
"UCASE" on page 686	The UCASE function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.
"UCASE (locale sensitive)" on page 687	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 688	Returns a string in which all the characters have been converted to uppercase characters.
"UPPER (locale sensitive)" on page 689	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 46. Table functions

Function	Description
"BASE_TABLE" on page 762	Returns both the object name and schema name of the object found after any alias chains have been resolved.
"UNNEST" on page 764	Returns a result table that includes a row for each element of the specified array.

Built-in functions

Table 46. Table functions (continued)

Function	Description
"XMLTABLE" on page 766	Returns a table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.

Table 47. XML functions

Function	Description
"PARAMETER" on page 570	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 383	Returns an XML sequence containing an item for each non-null value in a set of XML values.
"XMLATTRIBUTES" on page 720	Constructs XML attributes from the arguments.
"XMLCOMMENT" on page 722	Returns an XML value with a single XQuery comment node with the input argument as the content.
"XMLCONCAT" on page 723	Returns a sequence containing the concatenation of a variable number of XML input arguments.
"XMLDOCUMENT" on page 724	Returns an XML value with a single XQuery document node with zero or more children nodes.
"XMLELEMENT" on page 726	Returns an XML value that is an XML element node.
"XMLFOREST" on page 733	Returns an XML value that is a sequence of XML element nodes.
"XMLGROUP" on page 385	Returns an XML value with a single XQuery document node containing one top-level element node.
"XMLNAMESPACES" on page 736	Constructs namespace declarations from the arguments.
"XMLPARSE" on page 738	Parses the argument as an XML document and returns an XML value.
"XMLPI" on page 741	Returns an XML value with a single XQuery processing instruction node.
"XMLQUERY" on page 742	Returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.
"XMLROW" on page 745	Returns an XML value with a single XQuery document node containing one top-level element node.
"XMLSERIALIZE" on page 747	Returns a serialized XML value of the specified data type generated from the argument.
"XMLTABLE" on page 766	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 749	Returns an XML value with a single XQuery text node having the input argument as the content.
"XMLVALIDATE" on page 751	Returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.

Table 47. XML functions (continued)

Function	Description
"XMLXSROBJECTID" on page 756	Returns the XSR object identifier of the XML schema used to validate the XML document that is specified in the argument
"XSLTRANSFORM" on page 757	Converts XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

Aggregate functions

An aggregate function accepts arguments and returns a single scalar value that is the result of an evaluation of a set of like values, such as those in a column within a set of one or more rows.

The argument of an aggregate function is a set of values derived from an expression. The expression can include columns, but cannot include a *scalar-fullselect*, another aggregate function, or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42607). The scope of the set is a group or an intermediate result table.

If a GROUP BY clause is specified in a query, and the intermediate result of the FROM, WHERE, GROUP BY, and HAVING clauses is the empty set, the aggregate functions are not applied; the result of the query is the empty set; the SQLCODE is set to +100; and the SQLSTATE is set to '02000'.

If a GROUP BY clause is *not* specified in a query, and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, the aggregate functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

The keyword DISTINCT is not considered to be an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT in aggregate functions.

Expressions can be used in aggregate functions. For example:

Aggregate functions

```
SELECT MAX(BONUS + 1000)
  INTO :TOP_SALESREP_BONUS
  FROM EMPLOYEE
  WHERE COMM > 5000
```
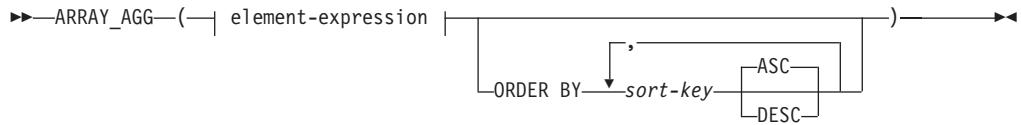
Aggregate functions can be qualified with a schema name (for example, SYSIBM.COUNT(*)).

ARRAY_AGG
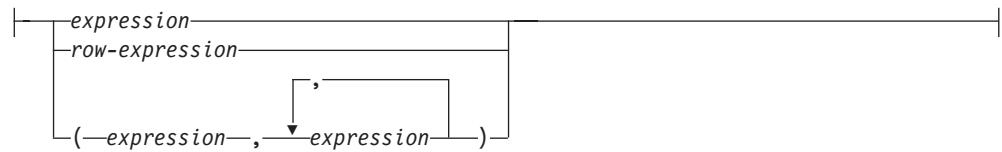
The ARRAY_AGG function aggregates a set of elements into an array.

Invocation of the ARRAY_AGG aggregate function is based on the result array type.

Ordinary array aggregation

>>──ARRAY_AGG──(──┤ element-expression ├─────────────────────────)──><
                                    └─ORDER BY──▼─sort-key─┬─ASC──┬─┘
                                                          └─DESC─┘

element-expression:
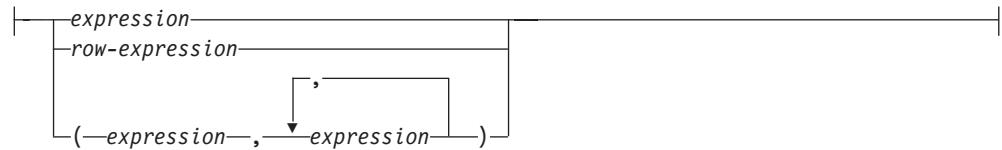
├─┬─expression────────────────────────┬──────────────────────────┤
  ├─row-expression────────────────────┤
  │   ┌─,───────────┐   │
  └─(──expression──,──▼─expression──┴──)─┘

Associative array aggregation

>>──ARRAY_AGG──(──index-expression──,──┤ element-expression ├──)──────────><

element-expression:

├─┬─expression────────────────────────┬──────────────────────────┤
  ├─row-expression────────────────────┤
  │   ┌─,───────────┐   │
  └─(──expression──,──▼─expression──┴──)─┘

The schema is SYSIBM.

Ordinary array aggregation

element-expression
Specifies the source for the elements of the array.

expression
An expression that specifies the element value for the array. The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement (SQLSTATE 429C2).

row-expression
A row expression that specifies the value that has a row data type as the element of the array.

(*expression,expression...* **)**
A list of two or more expressions that specify the fields for a value that has a row data type as the element of the array. The data type of each expression must be a valid data type for a row field as described in CREATE TYPE (row) statement (SQLSTATE 429C5).

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation of an ordinary array. If the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

If ORDER BY is specified, it determines the order of the aggregated elements in the ordinary array. If ORDER BY is not specified and no other ARRAY_AGG, LISTAGG or XMLAGG is included in the same SELECT clause with ordering specified, the ordering of elements within the ordinary array is not deterministic. If ORDER BY is not specified and the same SELECT clause has multiple occurrences of ARRAY_AGG, LISTAGG, or XMLAGG that specify ordering the same ordering of elements within the ordinary array is used for each result of ARRAY_AGG.

sort-key

The sort key can be a column name or a *sort-key-expression*. If the sort key is a constant, it does not refer to the position of the output column (as in the ORDER BY clause of a query), but it is simply a constant that implies no sort key.

ASC

Processes the *sort-key* in ascending order. This is the default option.

DESC

Processes the *sort-key* in descending order.

The result data type is an ordinary array. If the element values are specified using a single *expression* or *row-expression*, then the data type of the array element is the same as the type of the *expression* or *row-expression*. If the element values are specified with a list of expressions, then the array element is a row type with field types that correspond to the expressions.

If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Associative array aggregation

index-expression

Specifies the index of an associative array. When used in a context where there is a target user-defined array data type in the same statement or the result of the ARRAY_AGG is explicitly cast to a user-defined array data type, the data type of *index-expression* must be castable to the index data type of the target associative array data type. Otherwise, the data type of the *index-expression* must be a data type that can be specified for the index of an associative array in a CREATE TYPE (array) statement (SQLSTATE 429C2).

There cannot be any duplicate *index-expression* values in the grouping set that is processed to aggregate the associative array (SQLSTATE 22545).

element-expression

Specifies the source for the elements of the array.

expression

An expression that specifies the element value for the array.

The data type of the expression must be a data type that can be specified in a CREATE TYPE (array) statement (SQLSTATE 429C2).

row-expression
A row expression that specifies the value that has a row data type as the element of the array.

(*expression,expression...* **)**
A list of two or more expressions that specify the fields for a value that has a row data type as the element of the array. The data type of each expression must be a valid data type for a row field as described in CREATE TYPE (row) statement (SQLSTATE 429C5).

The result data type is an associative array. If the ARRAY_AGG is used in a context where there is a target user-defined array data type in the same statement or the result of the ARRAY_AGG is explicitly cast to a user-defined array data type, the data type of the index matches the data type of the target associative array. If the element values are specified using a single *expression* or *row-expression*, then the data type of the array element is the same as the type of the expression or row-expression. If the element values are specified with a list of expressions, then the array element is a row type with field types that correspond to the expressions.

Notes

- The ARRAY_AGG function can only be specified within an SQL procedure, compiled SQL function, or compound SQL (compiled) statement the following specific contexts (SQLSTATE 42887):
  - The select-list of a SELECT INTO statement
  - The select-list of a fullselect in the definition of a cursor that is not scrollable
  - The select-list of a scalar subquery on the right side of a SET statement
- ARRAY_AGG cannot be used as part of an OLAP function (SQLSTATE 42887).
- The SELECT statement that uses ARRAY_AGG cannot contain an ORDER BY clause or a DISTINCT clause, and the SELECT clause or HAVING clause cannot contain a subquery or invoke an inlined SQL function that returns a subquery (SQLSTATE 42887).

Examples

- *Example 1:* Given the following DDL:

```
CREATE TYPE PHONELIST AS DECIMAL(10, 0)ARRAY[10]

CREATE TABLE EMPLOYEE (
   ID          INTEGER NOT NULL,
   PRIORITY    INTEGER NOT NULL,
   PHONENUMBER DECIMAL(10, 0),
   PRIMARY KEY(ID, PRIORITY))
```

Create a procedure that uses a SELECT INTO statement to return the prioritized list of contact numbers under which an employee can be reached.

```
CREATE PROCEDURE GETPHONENUMBERS
  (IN  EMPID   INTEGER,
   OUT NUMBERS PHONELIST)
BEGIN
  SELECT ARRAY_AGG(PHONENUMBER ORDER BY PRIORITY)
```

```
        INTO NUMBERS
      FROM EMPLOYEE
      WHERE ID = EMPID;
   END
```

Create a procedure that uses a SET statement to return the list of contact numbers for an employee, in an arbitrary order.

```
CREATE PROCEDURE GETPHONENUMBERS
    (IN   EMPID    INTEGER,
     OUT NUMBERS PHONELIST)
  BEGIN
    SET NUMBERS =
      (SELECT ARRAY_AGG(PHONENUMBER)
       FROM EMPLOYEE
       WHERE ID = EMPID);
   END
```

- *Example 2:* Create a procedure that uses a SELECT INTO statement to aggregate priority 1 phone numbers into an associative array indexed by IDs from the EMPLOYEE table.

```
CREATE TYPE EMPPHONES AS DECIMAL(10,0) ARRAY[INTEGER]

CREATE PROCEDURE GETPHONES
(OUT EMPLOYEES EMPPHONES)
 BEGIN
  SELECT ARRAY_AGG(ID, PHONENUMBER)
  INTO EMPLOYEES
  FROM EMPLOYEE WHERE PRIORITY=1;
 END
```

- *Example 3:* Create a procedure that uses a SELECT INTO statement to aggregate the EMPLOYEE table into an array of row variable.
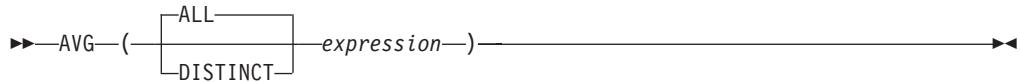
```
CREATE TYPE EMPROW AS ROW ANCHOR ROW EMPLOYEE

CREATE TYPE EMPARRAY AS EMPROW ARRAY[]

CREATE PROCEDURE GETEMPLOYEES
    (OUT EMPLOYEES EMPARRAY)
 BEGIN
  SELECT ARRAY_AGG((ID, PRIORITY, PHONENUMBER) ORDER BY ID)
  INTO EMPLOYEES
  FROM EMPLOYEE;
 END
```

AVG

The AVG function returns the average of a set of numbers.

```
►►─AVG─(─┬─ALL──────┬─expression─)────────────────────────►◄
         └─DISTINCT─┘
```

The schema is SYSIBM.

expression
    An expression that returns a value of any built-in numeric data type.

The data type of the result is the same as the data type of the 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).
- The result is decimal with precision 31 and scale 31-p+s if the argument values are decimal numbers with precision -p and scale -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).

The result can be null. If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of the set.

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

- *Example 1*: Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(PRSTAFF)
    INTO :AVERAGE
    FROM PROJECT
    WHERE DEPTNO = 'D11'
```

  Results in AVERAGE being set to 4.25 (that is 17/4) when using the sample table.

- *Example 2*: Using the PROJECT table, set the host variable ANY_CALC (decimal(5,2)) to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

```
SELECT AVG(DISTINCT PRSTAFF)
  INTO :ANY_CALC
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

Results in ANY_CALC being set to 4.66 (that is 14/3) when using the sample table.

CORRELATION

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

>>—CORRELATION—(—*expression1*—,—*expression2*—)—————————————><

The schema is SYSIBM.

expression1
   An expression that returns a value of any built-in numeric data type.

expression2
   An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null. When not null, the result is between -1 and 1.

The function is applied to the set of (*expression1*, *expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, or if either STDDEV(*expression1*) or STDDEV(*expression2*) is equal to zero, the result is a null value. Otherwise, the result is the correlation coefficient for the value pairs in the set. The result is equivalent to the following expression:

```
COVARIANCE(expression1,expression2)/
(STDDEV(expression1)*
STDDEV(expression2))
```

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

CORR can be specified in place of CORRELATION.

Example

Using the EMPLOYEE table, set the host variable CORRLN (double-precision floating point) to the correlation between salary and bonus for those employees in department (WORKDEPT) 'A00'.
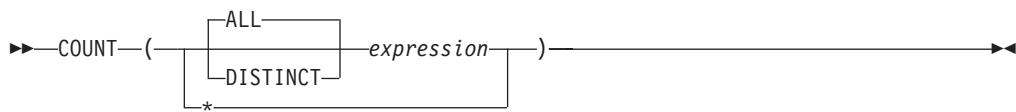
```
SELECT CORRELATION(SALARY, BONUS)
  INTO :CORRLN
  FROM EMPLOYEE
  WHERE WORKDEPT = 'A00'
```

CORRLN is set to approximately 9.99853953399538E-001 when using the sample table.

COUNT

The COUNT function returns the number of rows or values in a set of rows or values.

```
>>-COUNT--(--+-----ALL-----+--expression--)------------------><
             |             |
             +-DISTINCT----+
             |             |
             +-*-----------+
```

The schema is SYSIBM.

expression

> If ALL is implied or specified, an expression that returns a value of any built-in data type. If DISTINCT is specified, an expression that returns a value of any built-in data type except BLOB, CLOB, DBCLOB, or XML.

The result of the function is a large integer. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples

- *Example 1*: Using the EMPLOYEE table, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is 'F'.

  ```
  SELECT COUNT(*)
    INTO :FEMALE
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```

  Results in FEMALE being set to 13 when using the sample table.

- *Example 2*: Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.

  ```
  SELECT COUNT(DISTINCT WORKDEPT)
    INTO :FEMALE_IN_DEPT
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```

  Results in FEMALE_IN_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

COUNT_BIG

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.

```
►►──COUNT_BIG──(─┬─────────────┬─expression─┬──)──────────────────►◄
                 │   ┌─ALL─┐   │            │
                 ├───┴──────────┤            │
                 │  └─DISTINCT─┘ │            │
                 └─*────────────────────────┘
```

The schema is SYSIBM.

expression

> If ALL is implied or specified, an expression that returns a value of any built-in data type. If DISTINCT is specified, an expression that returns a value of any built-in data type except BLOB, CLOB, DBCLOB, or XML.

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only null values is included in the count.

The argument of COUNT_BIG(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT_BIG(*expression*) or COUNT_BIG(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples

- *Example 1*: Refer to COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- *Example 2*: Some applications may require the use of COUNT but need to support values larger than the largest integer. This can be achieved by use of sourced user-defined functions and setting the SQL path. The following series of statements shows how to create a sourced function to support COUNT(*) based on COUNT_BIG and returning a decimal value with a precision of 15. The SQL path is set such that the sourced function based on COUNT_BIG is used in subsequent statements such as the query shown.

```
CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
  SOURCE SYSIBM.COUNT_BIG();
SET CURRENT PATH RICK, SYSTEM PATH;
SELECT COUNT(*) FROM EMPLOYEE;
```

Note how the sourced function is defined with no parameters to support COUNT(*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:

```
SELECT MYCOUNT() FROM EMPLOYEE;
```

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements created a sourced function that will take any CHAR column as a argument and use COUNT_BIG to perform the counting.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
  SOURCE SYSIBM.COUNT_BIG(CHAR());
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

COVARIANCE

The COVARIANCE function returns the (population) covariance of a set of number pairs.

►►──COVARIANCE──(──*expression1*──,──*expression2*──)────────────────►◄

The schema is SYSIBM.

expression1
    An expression that returns a value of any built-in numeric data type.

expression2
    An expression that returns a value of any built-in numeric data type.

If either argument is decimal floating-point, the result is DECFLOAT(34); otherwise, the result is a double-precision floating-point number. The result can be null.

The function is applied to the set of (*expression1*,*expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The 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 1.68888888888889E+006 when using the sample table.

GROUPING

Used in conjunction with grouping-sets and super-groups, the GROUPING function returns a value that indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.

►►──GROUPING──(──*expression*──)──►◄

The schema is SYSIBM.

expression

An expression that matches a *grouping-expression* from the GROUP BY clause of the same subselect.

The result of the function is a small integer. It is set to one of the following values:

1 The value of *expression* in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide sub-total values for the GROUP BY expression.

0 The value is other than the previously listed value.

Example

The following query:

```
SELECT SALES_DATE, SALES_PERSON,
    SUM(SALES) AS UNITS_SOLD,
    GROUPING(SALES_DATE) AS DATE_GROUP,
    GROUPING(SALES_PERSON) AS SALES_GROUP
  FROM SALES
  GROUP BY CUBE (SALES_DATE, SALES_PERSON)
  ORDER BY SALES_DATE, SALES_PERSON
```
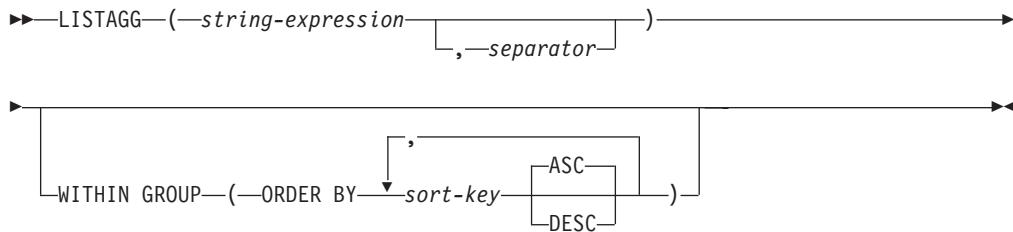
results in:

SALES_DATE	SALES_PERSON	UNITS_SOLD	DATE_GROUP	SALES_GROUP
12/31/1995	GOUNOT	1	0	0
12/31/1995	LEE	6	0	0
12/31/1995	LUCCHESSI	1	0	0
12/31/1995	-	8	0	1
03/29/1996	GOUNOT	11	0	0
03/29/1996	LEE	12	0	0
03/29/1996	LUCCHESSI	4	0	0
03/29/1996	-	27	0	1
03/30/1996	GOUNOT	21	0	0
03/30/1996	LEE	21	0	0
03/30/1996	LUCCHESSI	4	0	0
03/30/1996	-	46	0	1
03/31/1996	GOUNOT	3	0	0
03/31/1996	LEE	27	0	0
03/31/1996	LUCCHESSI	1	0	0
03/31/1996	-	31	0	1
04/01/1996	GOUNOT	14	0	0
04/01/1996	LEE	25	0	0
04/01/1996	LUCCHESSI	4	0	0
04/01/1996	-	43	0	1
-	GOUNOT	50	1	0
-	LEE	91	1	0
-	LUCCHESSI	14	1	0
-	-	155	1	1

An application can recognize a SALES_DATE sub-total row by the fact that the value of DATE_GROUP is 0 and the value of SALES_GROUP is 1. A SALES_PERSON sub-total row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE_GROUP and SALES_GROUP.

LISTAGG

The LISTAGG function aggregates a set of string elements into one string by concatenating the strings. Optionally, a separator string can be provided which is inserted between contiguous input strings.

```
►►──LISTAGG──(──string-expression─────────────────────)───────────────────►
                                    └─,──separator─┘

►────────────────────────────────────────────────────────────────────────►◄
   └─WITHIN GROUP──(──ORDER BY──┬──sort-key──┬─ASC──┬──)─┘
                                └───,────     └─DESC─┘
```

The schema is SYSIBM.

The LISTAGG function aggregates a set of string values for the group into one string by appending the *string-expression* values based on the order specified in the WITHIN GROUP clause.

The function is applied to the set of values derived from the first argument by the elimination of null values. If a *separator* argument is specified that is not the null value, the value is inserted between each pair of non-null *string-expression* values.

string-expression
An expression that returns a string that has data type of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC.

separator
A constant expression that returns a string that has a data type of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC for use as a separator between non-null *string-expression* values.

The separator can be a literal, special register, variable, or an expression based on literals, special registers, or variables, provided that the expression does not include a non-deterministic function or a function that takes external action.

WITHIN GROUP
Indicates that the aggregation will follow the specified ordering within the grouping set.

If WITHIN GROUP is not specified and no other LISTAGG, ARRAY_AGG, or XMLAGG is included in the same SELECT clause with ordering specified, the ordering of strings within the result is not deterministic. If WITHIN GROUP is not specified, and the same SELECT clause has multiple occurrences of XMLAGG, ARRAY_AGG, or LISTAGG that specify ordering, the same ordering is used for the result of the LISTAGG function invocation.

ORDER BY
Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key
The sort key can be a column name or a *sort-key-expression*. If the sort key

is a constant, it does not refer to the position of the output column (as in the ORDER BY clause of a query); it is a constant, which implies no sort key.

ASC

Processes the *sort-key* in ascending order. This is the default option.

DESC

Processes the *sort-key* in descending order.

The result data type of LISTAGG is based on the data type of *string-expression*:

- If the data type of *string-expression* is CHAR(n) or VARCHAR(n), the data type of result is VARCHAR(MAX(4000, n))
- If the data type of *string-expression* is GRAPHIC(n or VARGRAPHIC(n), the data type of result is VARGRAPHIC(MAX(2000, n))

The string units of the result data type is the same as the string units of the data type of *string-expression*.

The result data type can exceed VARCHAR(4000) or VARGRAPHIC(2000) if a derived size is used to determine the size of the result, to a maximum for the result data type. The following example successfully yields a return data type of VARCHAR(10000):

```
LISTAGG(CAST(NAME AS VARCHAR(10000)), ',')
```

If the actual length of the aggregated result string exceeds the maximum for the result data type, an error is returned (SQLSTATE 22001).

The result can be null. If the function is applied to an empty set or all of the *string-expression* values in the set are null values, the result is a null value.

Rules

- If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).
- LISTAGG cannot be used as part of an OLAP specification (SQLSTATE 42887).
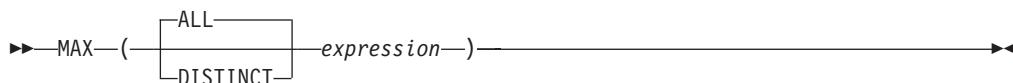
Example

Produce an alphabetical list of comma-separated names, grouped by department.

```
SELECT workdept,
       LISTAGG(lastname, ', ') WITHIN GROUP(ORDER BY lastname)
       AS employees
    FROM emp
    GROUP BY workdept
```

MAX

The MAX function returns the maximum value in a set of values.

```
►►─MAX─(─┬──────────┬─expression─)──────────────────────►◄
          ├─ALL──────┤
          └─DISTINCT─┘
```

The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The data type, length and code page of the result are the same as the data type, length and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Notes
- *Results involving DECFLOAT special values:* If the data type of the argument is decimal floating-point and positive or negative infinity, sNaN, or NaN is found, the maximum value is determined using decimal floating-point ordering rules. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples
- *Example 1*: Using the EMPLOYEE table, set the host variable MAX_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY/12) value.

  ```
  SELECT MAX(SALARY) / 12
    INTO :MAX_SALARY
    FROM EMPLOYEE
  ```

  Results in MAX_SALARY being set to 4395.83 when using the sample table.
- *Example 2*: Using the PROJECT table, set the host variable LAST_PROJ(char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

  ```
  SELECT MAX(PROJNAME)
    INTO :LAST_PROJ
    FROM PROJECT
  ```

  Results in LAST_PROJ being set to 'WELD LINE PLANNING' when using the sample table.
- *Example 3*: Similar to the previous example, set the host variable LAST_PROJ (char(40)) to the project name that comes last in the collating sequence when a project name is concatenated with the host variable PROJSUPP. PROJSUPP is '_Support'; it has a char(8) data type.
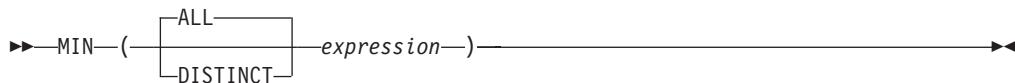
```
SELECT MAX(PROJNAME CONCAT PROJSUPP)
  INTO :LAST_PROJ
  FROM PROJECT
```

Results in LAST_PROJ being set to 'WELD LINE PLANNING_SUPPORT' when using the sample table.

MIN

The MIN function returns the minimum value in a set of values.

```
>>-MIN-(--+-ALL------+--expression-)----------------------------><
          +-DISTINCT-+
```

expression
> An expression that returns a value of any built-in data type other than a BLOB, CLOB, DBCLOB, or XML.

The data type, length, and code page of the result are the same as the data type, length, and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If this function is applied to an empty set, the result of the function is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Notes
- *Results involving DECFLOAT special values:* If the data type of the argument is decimal floating-point and positive or negative infinity, sNaN, or NaN is found, the minimum value is determined using decimal floating-point ordering rules. If multiple representations of the same decimal floating-point value are found (for example, 2.00 and 2.0), it is unpredictable which representation will be returned.

Examples
- *Example 1*: Using the EMPLOYEE table, set the host variable COMM_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

  ```
  SELECT MAX(COMM) - MIN(COMM)
    INTO :COMM_SPREAD
    FROM EMPLOYEE
    WHERE WORKDEPT = 'D11'
  ```

  Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.
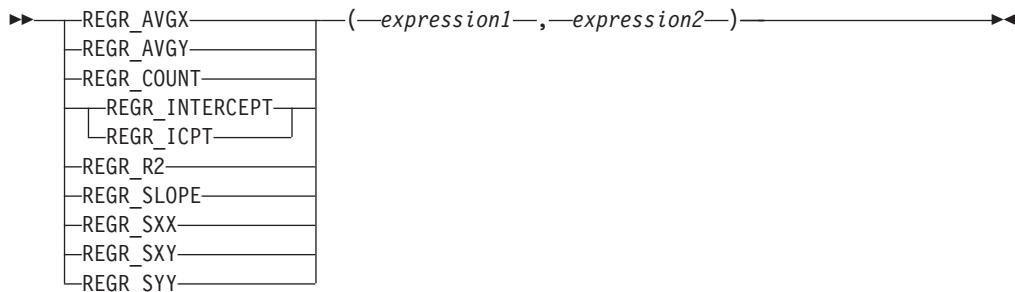
- *Example 2*: Using the PROJECT table, set the host variable (FIRST_FINISHED (char(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

  ```
  SELECT MIN(PRENDATE)
    INTO :FIRST_FINISHED
    FROM PROJECT
  ```

  Results in FIRST_FINISHED being set to '1982-09-15' when using the sample table.

Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)

The regression functions support the fitting of an ordinary-least-squares regression line of the form $y = a * x + b$ to a set of number pairs.

```
►►──┬──REGR_AVGX──────────┬──(─expression1─,─expression2─)──────────►◄
    ├──REGR_AVGY──────────┤
    ├──REGR_COUNT─────────┤
    ├──┬──REGR_INTERCEPT──┬──┤
    │  └──REGR_ICPT───────┘  │
    ├──REGR_R2────────────┤
    ├──REGR_SLOPE─────────┤
    ├──REGR_SXX───────────┤
    ├──REGR_SXY───────────┤
    └──REGR_SYY───────────┘
```

The schema is SYSIBM.

expression1
> An expression that returns a value of any built-in numeric data type. It is interpreted as a value of the dependent variable (that is, a "y value").

expression2
> An expression that returns a value of any built-in numeric data type. It is interpreted as a value of the independent variable (that is, an "x value").

The REGR_COUNT function returns the number of non-null number pairs used to fit the regression line.

The REGR_INTERCEPT (or REGR_ICPT) function returns the y-intercept of the regression line ("b" in the equation mentioned previously).

The REGR_R2 function returns the coefficient of determination ("R-squared" or "goodness-of-fit") for the regression.

The REGR_SLOPE function returns the slope of the line ("a" in the equation mentioned previously).

The REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SXY, and REGR_SYY functions return quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model.

The data type of the result of REGR_COUNT is integer. For the remaining functions, if either argument is DECFLOAT(n), the data type of the result is DECFLOAT(34); otherwise, the data type of the result is double-precision floating-point. If either argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" in "General arithmetic operation rules for decimal floating-point" on page 248.

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.

Regression functions (REGR_AVGX, REGR_AVGY, REGR_COUNT, ...)

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 e*xpression2* is null.

If the set is not empty and VARIANCE(*expression2*) is positive, REGR_COUNT returns the number of non-null pairs in the set, and the remaining functions return results that are defined as follows:

```
REGR_SLOPE(expression1,expression2) =
COVARIANCE(expression1,expression2)/VARIANCE(expression2)
```

```
REGR_INTERCEPT(expression1, expression2) =
AVG(expression1) - REGR_SLOPE(expression1, expression2) * AVG(expression2)
```

```
REGR_R2(expression1, expression2) =
POWER(CORRELATION(expression1, expression2), 2) if VARIANCE(expression1)>0
REGR_R2(expression1, expression2) = 1 if VARIANCE(expression1)=0
```

```
REGR_AVGX(expression1, expression2) = AVG(expression2)
```

```
REGR_AVGY(expression1, expression2) = AVG(expression1)
```

```
REGR_SXX(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression2)
```

```
REGR_SYY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression1)
```

```
REGR_SXY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * COVARIANCE(expression1, expression2)
```

If the set is not empty and VARIANCE(*expression2*) is equal to zero, then the regression line either has infinite slope or is undefined. In this case, the functions REGR_SLOPE, REGR_INTERCEPT, and REGR_R2 each return a null value, and the remaining functions return values as defined previously. If the set is empty, REGR_COUNT returns zero and the remaining functions return a null value.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

The regression functions are all computed simultaneously during a single pass through the data. In general, it is more efficient to use the regression functions to compute the statistics needed for a regression analysis than to perform the equivalent computations using ordinary column functions such as AVERAGE, VARIANCE, COVARIANCE, and so forth.

The usual diagnostic statistics that accompany a linear-regression analysis can be computed in terms of the functions listed previously. For example:

Adjusted R2
> 1 - ((1 - REGR_R2) * ((REGR_COUNT - 1) / (REGR_COUNT - 2)))

Standard error
> SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX))/(REGR_COUNT-2))

Total sum of squares
> REGR_SYY

Regression sum of squares
> POWER(REGR_SXY,2) / REGR_SXX

Residual sum of squares
> (Total sum of squares)-(Regression sum of squares)

t statistic for slope
> REGR_SLOPE * SQRT(REGR_SXX) / (Standard error)

t statistic for y-intercept

REGR_INTERCEPT/((Standard error) * SQRT((1/
REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))

Example

Using the EMPLOYEE table, compute an ordinary-least-squares regression line that expresses the bonus of an employee in department (WORKDEPT) 'A00' as a linear function of the employee's salary. Set the host variables SLOPE, ICPT, RSQR (double-precision floating point) to the slope, intercept, and coefficient of determination of the regression line, respectively. Also set the host variables AVGSAL and AVGBONUS to the average salary and average bonus, respectively, of the employees in department 'A00', and set the host variable CNT (integer) to the number of employees in department 'A00' for whom both salary and bonus data are available. Store the remaining regression statistics in host variables SXX, SYY, and SXY.

```
SELECT REGR_SLOPE(BONUS,SALARY), REGR_INTERCEPT(BONUS,SALARY),
REGR_R2(BONUS,SALARY), REGR_COUNT(BONUS,SALARY),
REGR_AVGX(BONUS,SALARY), REGR_AVGY(BONUS,SALARY),
REGR_SXX(BONUS,SALARY), REGR_SYY(BONUS,SALARY),
REGR_SXY(BONUS,SALARY)
INTO :SLOPE, :ICPT,
:RSQR, :CNT,
:AVGSAL, :AVGBONUS,
:SXX, :SYY,
:SXY
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

When using the sample table, the host variables are set to the following approximate values:

```
SLOPE:    +1.71002671916749E-002
ICPT:    +1.00871888623260E+002
RSQR:    +9.99707928128685E-001
CNT:  3
AVGSAL:   +4.28333333333333E+004
AVGBONUS:   +8.33333333333333E+002
SXX:   +2.96291666666667E+008
SYY:   +8.66666666666667E+004
SXY:   +5.06666666666667E+006
```

STDDEV

The STDDEV function returns the standard deviation ($/n$) of a set of numbers.

```
►►──STDDEV──(─┬─ALL──────┬──expression──)──────────────────────────►◄
              └─DISTINCT─┘
```

The schema is SYSIBM.

expression
   An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is double-precision floating-point. The result can be null.

The formula used to calculate STDDEV is:

```
STDDEV = SQRT(VARIANCE)
```

where SQRT(VARIANCE) is the square root of the variance.

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
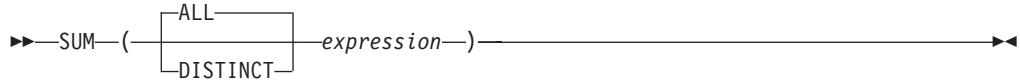
Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries of employees in department (WORKDEPT) 'A00'.

```
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 SUM function returns the sum of a set of numbers.

```
>>--SUM--(--+--ALL------+--expression--)--------------------------------------><
            |           |
            +--DISTINCT-+
```

The schema is SYSIBM.

expression
An expression that returns a value of any built-in numeric data type.

The data type of the result is the same as the data type of the 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).
- The result is decimal with precision 31 and scale s if the argument values are decimal with precision p and scale 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 also eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

The result can be null. If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example

Using the EMPLOYEE table, set the host variable JOB_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').
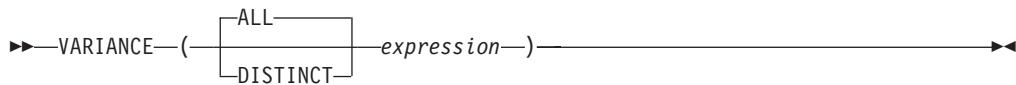
```
SELECT SUM(BONUS)
  INTO :JOB_BONUS
  FROM EMPLOYEE
  WHERE JOB = 'CLERK'
```

Results in JOB_BONUS being set to 2800 when using the sample table.

VARIANCE

The VARIANCE function returns the variance of a set of numbers.

```
►►──VARIANCE──(──┬──ALL──────┬──expression──)──────────────────────►◄
                 └──DISTINCT──┘
```

The schema is SYSIBM.

expression
   An expression that returns a value of any built-in numeric data type.

If the argument is DECFLOAT(*n*), the result is DECFLOAT(*n*); otherwise, the result is double-precision floating-point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated. When interpreting the DISTINCT clause for decimal floating-point values that are numerically equal, the number of significant digits in the value is not considered. For example, the decimal floating-point number 123.00 is not distinct from the decimal floating-point number 123. The representation of the number returned from the query will be any one of the representations encountered (for example, either 123.00 or 123).

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The 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
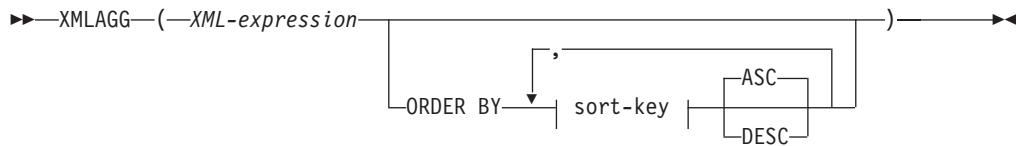
Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department (WORKDEPT) 'A00'.

```
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 XMLAGG function returns an XML sequence containing an item for each non-null value in a set of XML values.

```
►►──XMLAGG──(──XML-expression────────────────────────────────)────►◄
                      │                          ┌─,─┐          │
                      └─ORDER BY──┬─ sort-key ─┬──┬─ASC──┬──┘
                                  └───────────┘  └─DESC─┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

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.

If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Notes

- **Support in OLAP expressions**: XMLAGG cannot be used as a column function of an OLAP aggregation function (SQLSTATE 42601).

Example

Construct a department element for each department, containing a list of employees sorted by last name.

```
SELECT XMLSERIALIZE(
  CONTENT XMLELEMENT(
    NAME "Department", XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.LASTNAME
```

XMLAGG

```
            )
            ORDER BY E.LASTNAME
        )
    )
      AS CLOB(110)
  )
  AS "dept_list"
  FROM EMPLOYEE E
  WHERE E.WORKDEPT IN ('C01','E21')
  GROUP BY WORKDEPT
```
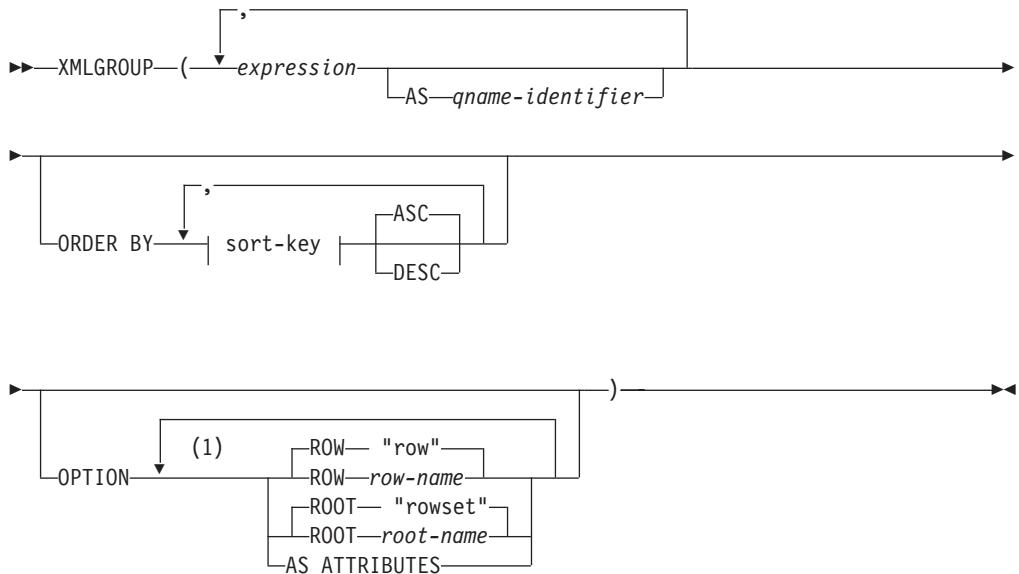
This query produces the following result:

```
dept_list
-----------------------...
<Department name="C01">
  <emp>KWAN</emp>
  <emp>NICHOLLS</emp>
  <emp>QUINTANA</emp>
</Department>
<Department name="E21">
  <emp>GOUNOT</emp>
  <emp>LEE</emp>
  <emp>MEHTA</emp>
  <emp>SPENSER</emp>
</Department>
```

Note: XMLAGG does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

XMLGROUP

The XMLGROUP function returns an XML value with a single XQuery document node containing one top-level element node. This is an aggregate expression that will return a single-rooted XML document from a group of rows where each row is mapped to a row subelement.

Notes:

1   The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

expression
> The content of each generated XML element node (or the value of each generated attribute) is specified by an expression. The data type of *expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, a *qname-identifier* must be specified.

AS *qname-identifier*
> Specifies the XML element name or attribute name as an SQL identifier. The *qname-identifier* must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *qname-identifier* is not specified, *expression* must be a column name (SQLSTATE 42703). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
> Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

ROW *row-name*
> Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

ROOT *root-name*

Specifies the name of the root element node. If this option is not specified, the default root element name is "rowset"

AS ATTRIBUTES

Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

ORDER BY

Specifies the order of the rows from the same grouping set that are processed in the aggregation. If the ORDER BY clause is omitted, or if the ORDER BY clause cannot distinguish the order of the column data, the rows in the same grouping set are arbitrarily ordered.

sort-key

The sort key can be a column name or a *sort-key-expression*. Note that if the sort key is a constant, it does not refer to the position of the output column (as in the ordinary ORDER BY clause), but it is simply a constant, which implies no sort key.

Rules

- If a SELECT clause includes an ARRAY_AGG function, then all invocations of ARRAY_AGG, LISTAGG, XMLAGG, and XMLGROUP functions in the same SELECT clause must specify the same order or not specify an order (SQLSTATE 428GZ).

Notes

The default behavior defines a simple mapping between a result set and an XML value. Some additional notes about function behavior apply:

- By default, each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name serving as the element name.
- The null handling behavior is NULL ON NULL. A null value in a column maps to the absence of the subelement. If all column values are null, no row element will be generated.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- By default, the elements corresponding to the rows in a group are children of a root element named "rowset".
- The order of the row subelements in the root element will be the same as the order in which the rows are returned in the query result set.
- A document node will be added implicitly to the root element to make the XML result a well-formed single-rooted XML document

Examples

The provided examples are based on the following table, T1, with integer columns C1 and C2 that contain numeric data stored in a relational format.

```
C1          C2
----------- -----------
          1           2
          -           2
          1           -
          -           -

  4 record(s) selected.
```

- *Example 1*: The following example shows an XMLGroup query and output fragment with default behavior, using a single top-level element to represent the table:

```
SELECT XMLGROUP(C1, C2)FROM T1

<rowset>
   <row>
      <C1>1</C1>
      <C2>2</C2>
   </row>
   <row>
      <C2>2</C2>
   </row>
   <row>
      <C1>1</C1>
   </row>
</rowset>

  1 record(s) selected.
```

- *Example 2*: The following example shows an XMLGroup query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

```
SELECT XMLGROUP(C1, C2 OPTION AS ATTRIBUTES) FROM T1

<rowset>
   <row C1="1" C2="2"/>
   <row C2="2"/>
   <row C1="1"/>
</rowset>

  1 record(s) selected.
```

- *Example 3*: The following example shows an XMLGroup query and output fragment with the default <rowset> root element replaced by <document> and the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the return set is ordered by column C1:

```
SELECT XMLGROUP(
   C1 AS "column1", C2 AS "column2"
   ORDER BY C1 OPTION ROW "entry" ROOT "document")
FROM T1

<document>
   <entry>
      <column1>1</column1>
      <column2>2</column2>
   </entry>
   <entry>
      <column1>1</column1>
   </entry>
   <entry>
      <column2>2</column2>
    </entry>
</document>
```

Scalar functions

A scalar function optionally accepts arguments and returns a single scalar value each time the function is called.

A scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and aggregate functions also apply when an expression or aggregate function is used within a scalar function. For

Scalar functions

example, the argument of a scalar function can be an aggregate function only if an aggregate function is allowed in the context in which the scalar function is used.

The restrictions on the use of aggregate functions do not apply to scalar functions, because a scalar function is applied to a single value rather than to a set of values.

The result of the following SELECT statement has as many rows as there are employees in department D01:
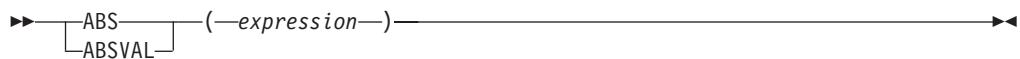
```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

Scalar functions can be qualified with a schema name (for example, SYSIBM.CHAR(123)).

In a Unicode database, all scalar functions that accept a character or graphic string will accept any string types for which conversion is supported.

ABS or ABSVAL

Returns the absolute value of the argument.

```
►►──┬─ABS────┬──(─expression─)──────────────────────────────────►◄
    └─ABSVAL─┘
```

The schema is SYSIBM.

The SYSFUN version of the ABS (or ABSVAL) function continues to be available.

expression
>   An expression that returns a value of any built-in numeric data type.

The result has the same data type and length attribute as the argument. The result can be null; if the argument is null, the result is the null value. If the argument is the maximum negative value for SMALLINT, INTEGER or BIGINT, the result is an overflow error.

Notes

Results involving DECFLOAT special values: For decimal floating-point values, the special values are treated as follows:
- ABS(NaN) and ABS(-NaN) return NaN.
- ABS(Infinity) and ABS(-Infinity) return Infinity.
- ABS(sNaN) and ABS(-sNaN) return sNaN.

Example

```
ABS(-51234)
```

returns an INTEGER with a value of 51234.

ACOS

Returns the arccosine of the argument as an angle expressed in radians.

►►──ACOS──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the ACOS function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Example

Assume that the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
SELECT ACOS(:ACOSINE)
  FROM SYSIBM.SYSDUMMY1
```

This statement returns the approximate value 1.49.

ADD_MONTHS

The ADD_MONTHS function returns a datetime value that represents *expression* plus a specified number of months.

►►──ADD_MONTHS──(──*expression*──,──*numeric-expression*──)──────────────────►◄

The schema is SYSIBM.

expression
> An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

numeric-expression
> An expression that returns a value of any built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The *numeric-expression* specifies the number of months to add to the starting date specified by *expression*. A negative numeric value is allowed.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is DATE. The result can be null; if any argument is null, the result is the null value.

If *expression* is the last day of the month or if the resulting month has fewer days than the day component of *expression*, the result is the last day of the resulting month. Otherwise, the result has the same day component as *expression*. Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function.

Examples

* *Example 1*: Assume today is January 31, 2007. Set the host variable ADD_MONTH with the last day of January plus 1 month.

      SET :ADD_MONTH = ADD_MONTHS(LAST_DAY(CURRENT_DATE), 1);

  The host variable ADD_MONTH is set with the value representing the end of February, 2007-02-28.

* *Example 2*: Assume DATE is a host variable with the value July 27, 1965. Set the host variable ADD_MONTH with the value of that day plus 3 months.

      SET :ADD_MONTH = ADD_MONTHS(:DATE,3);

  The host variable ADD_MONTH is set with the value representing the day plus 3 months, 1965-10-27.

* *Example 3*: It is possible to achieve similar results with the ADD_MONTHS function and date arithmetic. The following examples demonstrate the similarities and contrasts.

      SET :DATEHV = DATE('2008-2-28') + 4 MONTHS;
      SET :DATEHV = ADD_MONTHS('2008-2-28', 4);

  In both cases, the host variable DATEHV is set with the value '2008-06-28'.

  Now consider the same examples but with the date '2008-2-29' as the argument.

      SET :DATEHV = DATE('2008-2-29') + 4 MONTHS;

  The host variable DATEHV is set with the value '2008-06-29'.

      SET :DATEHV = ADD_MONTHS('2008-2-29', 4);

  The host variable DATEHV is set with the value '2008-06-30'.

ADD_MONTHS

In this case, the ADD_MONTHS function returns the last day of the month, which is June 30, 2008, instead of June 29, 2008. The reason is that February 29 is the last day of the month. So, the ADD_MONTHS function returns the last day of June.

ARRAY_DELETE

The ARRAY_DELETE function deletes elements from an array.

```
►►─ARRAY_DELETE─(─array-variable──────────────────────────)─────►◄
                              └─,─array-index1─────┘
                                      └─,─array-index2─┘
```

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index1
> An expression that results in a value that can be assigned to the data type of the array index. If *array-variable* is an ordinary array, *array-index1* must be the null value.

array-index2
> An expression that results in a value that can be assigned to the data type of the array index. If *array-variable* is an ordinary array, *array-index2* must be the null value. If *array-index2* is specified and is a non-null value, then *array-index1* must be a non-null value that is less than the value of *array-index2* (SQLSTATE 42815).

The result of the function has the same data type as *array-variable*. If the optional arguments are not specified or they are the null value, all of the elements of *array-variable* are deleted and the cardinality of the result array value is 0. If only *array-index1* is specified with a non-null value, the array element at index value *array-index1* is deleted. If *array-index2* is also specified with a non-null value, then the elements ranging from index value *array-index1* to *array-index2* (inclusive) are deleted.

The result can be null; if *array-variable* is null, the result is the null value.

Notes

- The ARRAY_DELETE function can be used only on the right side of an assignment statement in contexts where arrays are supported.

Examples

- *Example 1*: Delete all the elements from the ordinary array variable RECENT_CALLS of array type PHONENUMBERS .

  SET RECENT_CALLS **= ARRAY_DELETE(** RECENT_CALLS **)**

- *Example 2*: A supplier has discontinued some of their products. Delete the elements from the associative array variable FLOOR_TILES of array type PRODUCTS from index value 'PK5100' to index value 'PS2500'.

  SET FLOOR_TILES **= ARRAY_DELETE(** FLOOR_TILES **,** 'PK5100' **,** 'PS2500' **)**

ARRAY_FIRST

The ARRAY_FIRST function returns the minimum array index value of the array.

```
►►──ARRAY_FIRST──(──array-variable──)────────────────────────────────►◄
```

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER for an ordinary array. If *array-variable* is not null and the cardinality of the array is greater than zero, the value of the result is the minimum array index value, which is 1 for an ordinary array.

The result can be null; if *array-variable* is null or the cardinality of the array is zero, the result is the null value.

Examples

- *Example 1*: Return the first index value in the ordinary array variable SPECIALNUMBERS to the SQL variable E_CONSTIDX.

  SET E_CONSTIDX **= ARRAY_FIRST(**SPECIALNUMBERS**)**

  The result is 1.

- *Example 2*: Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the minimum index in the array to the character string variable named X.

  SET X **= ARRAY_FIRST(**PHONELIST**)**

  The value of 'Home' is assigned to X. Access the element value associated with index value 'Home' and assign it to the SQL variable NUMBER_TO_CALL:

  SET NUMBER_TO_CALL **=** PHONELIST**[**X**]**

ARRAY_LAST

The ARRAY_LAST function returns the maximum array index value of the array.

►►—ARRAY_LAST—(—*array-variable*—)————————————————————————►◄

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The data type of the result is the data type of the array index, which is INTEGER for an ordinary array. If *array-variable* is not null and the cardinality of the array is greater than zero, the value of the result is the maximum array index value, which is the cardinality of the array for an ordinary array.

The result can be null; if array-variable is null or the cardinality of the array is zero, the result is the null value.

Examples

- *Example 1*: Return the last index value in the ordinary array variable SPECIALNUMBERS to the SQL variable PI_CONSTIDX.

  SET PI_CONSTIDX = **ARRAY_LAST(**SPECIALNUMBERS**)**

  The result is 10.

- *Example 2*: Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '4164789683', assign the value of the maximum index in the array to the character string variable named X.

  SET X = **ARRAY_LAST(**PHONELIST**)**

  The value of 'Work' is assigned to X. Access the element value associated with index value 'Work' and assign it to the SQL variable NUMBER_TO_CALL:

  SET NUMBER_TO_CALL = PHONELIST**[**X**]**

ARRAY_NEXT

The ARRAY_NEXT function returns the next larger array index value for an array relative to the specified array index argument.

►►──ARRAY_NEXT──(─*array-variable*─,─*array-index*─)──────────────────────►◄

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
> Specifies a value that is assignable to the data type of the index of the array. Valid values include any valid value for the data type.

The result is the next larger array index value defined in the array relative to the specified *array-index* value. If *array-index* is less than the minimum index array value in the array, the result is the first array index value defined in the array.

The data type of the result of the function is the data type of the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of *array-index* is greater than or equal to the value of the last index in the array, the result is the null value.

Examples

- *Example 1*: Return the next index value after the 9th index position in the ordinary array variable SPECIALNUMBERS to the SQL variable NEXT_CONSTIDX.

  SET NEXT_CONSTIDX **= ARRAY_NEXT(**SPECIALNUMBERS**,**9**)**

  The result is 10.

- *Example 2*: Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the next index after index value 'Dad', which does not exist for the array value, to the character string variable named X:

  SET X **= ARRAY_NEXT(**PHONELIST**,** 'Dad'**)**

  The value of 'Home' is assigned to X, since the value 'Dad' is a value smaller than any index value for the array variable. Assign the value of the index in the array that is the next index after index value'Work':

  SET X **= ARRAY_NEXT(**PHONELIST**,** 'Work'**)**

  The null value is assigned to X.

ARRAY_PRIOR

The ARRAY_PRIOR function returns the next smaller array index value for an array relative to the specified array index argument.

►►—ARRAY_PRIOR—(—*array-variable*—,—*array-index*—)————————————————►◄

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

array-index
> Specifies a value that is assignable to the data type of the index of the array. Valid values include any valid value for the data type.

The result is the next smaller array index value defined in the array relative to the specified *array-index* value. If *array-index* is greater than the maximum index array value in the array, the result is the last array index value defined in the array.

The data type of the result of the function is the data type the array index. The result can be null; if either argument is null, the cardinality of the first argument is zero, or the value of *array-index* is less than or equal to the value of the first index in the array, the result is the null value.

Examples

- *Example 1*: Return the previous index value before the 2nd index position in the ordinary array variable SPECIALNUMBERS to the SQL variable PREV_CONSTIDX.

  SET PREV_CONSTIDX = **ARRAY_PRIOR(**SPECIALNUMBERS,2**)**

  The result is 1.

- *Example 2*: Given the associative array variable PHONELIST with index values and phone numbers: 'Home' is '4163053745', 'Work' is '4163053746', and 'Mom' is '416-4789683', assign the value of the index in the array that is the previous index before index value 'Work' to the character string variable named X:

  SET X = **ARRAY_PRIOR(**PHONELIST, 'Work'**)**

  The value of 'Mom' is assigned to X. Assign the value of the index in the array that is the previous index before index value'Home':

  SET X = **ARRAY_PRIOR(**PHONELIST, 'Home'**)**

  The null value is assigned to X.

ASCII

Returns the ASCII code value of the leftmost character of the argument as an integer.

►►──ASCII──(──*expression*──)──►◄

The schema is SYSFUN.

expression
> An expression that returns a value of any built-in character string data type. In a Unicode database, if a supplied value is a graphic string data type, 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.

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

Returns the arcsine on the argument as an angle expressed in radians.

```
►►──ASIN──(──expression──)────────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the ASIN function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATAN

Returns the arctangent of the argument as an angle expressed in radians.

►►──ATAN──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the ATAN function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATAN2

Returns the arctangent of x and y coordinates as an angle expressed in radians. The x and y coordinates are specified by the first and second arguments, respectively.

►►──ATAN2──(──*expression1*──,──*expression2*──)────────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the ATAN2 function continues to be available.)

expression1
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

expression2
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

ATANH

Returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.

►►──ATANH──(──*expression*──)──►◄

The schema is SYSIBM.

expression
    An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

BIGINT

The BIGINT function returns a 64-bit integer representation of a number, a string representation of a number, or a datetime value.

Numeric to Big Integer

►►——BIGINT——(——*numeric-expression*——)——————————————————————►◄

String to Big Integer

►►——BIGINT——(——*string-expression*——)——————————————————————►◄

Datetime to Big Integer

►►——BIGINT——(——*datetime-expression*——)—————————————————————►◄

The schema is SYSIBM.

Numeric to Big Integer

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a big integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003).

String to Big Integer

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

The result is the same number that would result from CAST(*string-expresssion* AS BIGINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of big integers, an error is returned (SQLSTATE 22003). The data type of *string-expresssion* must not be CLOB or DBCLOB (SQLSTATE 42884).

Datetime to Big Integer

datetime-expression
An expression that is of one of the following data types:
- DATE. The result is a BIGINT value representing the date as *yyyymmdd*.
- TIME. The result is a BIGINT value representing the time as *hhmmss*.
- TIMESTAMP. The result is a BIGINT value representing the timestamp as *yyyymmddhhmmss*. The fractional seconds portion of the timestamp value is not included in the result.

BIGINT

The result of then function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- The CAST specification should be used to increase the portability of applications.

Examples

- *Example 1*: From ORDERS_HISTORY table, count the number of orders and return the result as a big integer value.

  ```
  SELECT BIGINT (COUNT_BIG(*))
    FROM ORDERS_HISTORY
  ```

- *Example 2*: Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application.

  ```
  SELECT BIGINT (EMPNO) FROM EMPLOYEE
  ```

- *Example 3*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

  ```
  BIGINT(RECEIVED)
  ```

  results in the value 19 881 222 140 721.
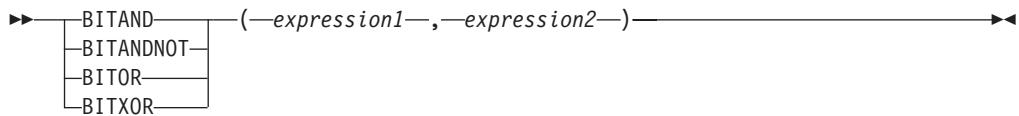
- *Example 4*: Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to '12:03:04'.

  ```
  BIGINT(STARTTIME)
  ```

  results in the value 120 304.

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value in a data type based on the data type of the input arguments.

```
►►──┬─BITAND────┬──(─expression1─,─expression2─)──────────────────►◄
    ├─BITANDNOT─┤
    ├─BITOR─────┤
    └─BITXOR────┘
```

```
►►──BITNOT─(─expression─)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

Table 48. 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.

expression **or** *expression1* **or** *expression2*

The arguments must be integer values represented by the data types SMALLINT, INTEGER, BIGINT, or DECFLOAT. Arguments of type DECIMAL, REAL, or DOUBLE are cast to DECFLOAT. The value is truncated to a whole number.

The bit manipulation functions can operate on up to 16 bits for SMALLINT, 32 bits for INTEGER, 64 bits for BIGINT, and 113 bits for DECFLOAT. The range of supported DECFLOAT values includes integers from -2^{112} to 2^{112} -1, and special values such as NaN or INFINITY are not supported (SQLSTATE 42815). If the two arguments have different data types, the argument supporting fewer bits is cast to a value with the data type of the argument supporting more bits. This cast impacts the bits that are set for negative values. For example, -1 as a SMALLINT value has 16 bits set to 1, which when cast to an INTEGER value has 32 bits set to 1.

BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

The result of the functions with two arguments has the data type of the argument that is highest in the data type precedence list for promotion. If either argument is DECFLOAT, the data type of the result is DECFLOAT(34). If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The result of the BITNOT function has the same data type as the input argument, except that DECIMAL, REAL, DOUBLE, or DECFLOAT(16) returns DECFLOAT(34). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Due to differences in internal representation between data types and on different hardware platforms, using functions (such as HEX) or host language constructs to view or compare internal representations of BIT function results and arguments is data type-dependent and not portable. The data type- and platform-independent way to view or compare BIT function results and arguments is to use the actual integer values.

Use of the BITXOR function is recommended to toggle bits in a value. Use the BITANDNOT function to clear bits. BITANDNOT(val, pattern) operates more efficiently than BITAND(val, BITNOT(pattern)).

Examples

The following examples are based on an ITEM table with a PROPERTIES column of type INTEGER.

- *Example 1*: Return all items for which the third property bit is set.

  ```
  SELECT ITEMID FROM ITEM
    WHERE BITAND(PROPERTIES, 4) = 4
  ```

- *Example 2*: Return all items for which the fourth or the sixth property bit is set.

  ```
  SELECT ITEMID FROM ITEM
    WHERE BITAND(PROPERTIES, 40) <> 0
  ```

- *Example 3*: Clear the twelfth property of the item whose ID is 3412.

  ```
  UPDATE ITEM
      SET PROPERTIES = BITANDNOT(PROPERTIES, 2048)
    WHERE ITEMID = 3412
  ```

- *Example 4*: Set the fifth property of the item whose ID is 3412.

  ```
  UPDATE ITEM
      SET PROPERTIES = BITOR(PROPERTIES, 16)
    WHERE ITEMID = 3412
  ```

- *Example 5*: Toggle the eleventh property of the item whose ID is 3412.

  ```
  UPDATE ITEM
      SET PROPERTIES = BITXOR(PROPERTIES, 1024)
    WHERE ITEMID = 3412
  ```

- *Example 6*: Switch all the bits in a 16-bit value that has only the second bit on.

  ```
  VALUES BITNOT(CAST(2 AS SMALLINT))
  ```

  returns -3 (with a data type of SMALLINT).

BLOB

The BLOB function returns a BLOB representation of a string of any type.

►►──BLOB──(──*string-expression*────────────)──────────────────────────►◄
                          └─,──*integer*─┘

The schema is SYSIBM.

string-expression
> An expression that returns a value of a character string, graphic string, or binary string data type.

integer
> An integer value specifying the length attribute of the resulting BLOB data type. If *integer* is not specified, the length attribute of the result is the same as the length of the input, except where the input is graphic. In this case, the length attribute of the result is twice the length of the input.

The result of the function is a BLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

Given a table with a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME, locate any maps that contain the string 'Pellow Island' and return a single binary string with the map name concatenated in front of the actual map.

```
SELECT BLOB(MAP_NAME  CONCAT ':  ') CONCAT TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Pellow Island%')
```

CARDINALITY

The CARDINALITY function returns a value of type BIGINT representing the number of elements of an array.

►►──CARDINALITY──(──*array-variable*──)────────────────────────────►◄

The schema is SYSIBM.

array-variable
    An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

For an ordinary array, the value returned by the CARDINALITY function is the highest array index for which the array has an assigned element. This includes elements that have been assigned the null value. For an associative array, the value returned by the CARDINALITY function is the actual number of unique array index values defined in *array-variable*.

The function returns 0 if the array is empty. The result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Return the number of calls that have been stored in the recent calls list so far:

    SET HOWMANYCALLS **= CARDINALITY(**RECENT_CALLS**)**

    The SQL variable HOWMANYCALLS contains the value 3.

- *Example 2*: Assume that the associative array variable CAPITALS of array type CAPITALSARRAY contains all of the capitals for the 10 provinces and 3 territories in Canada as well as the capital of the country, Ottawa. Return the cardinality of the array variable:

    SET NUMCAPITALS **= CARDINALITY(**CAPITALS**)**

    The SQL variable NUMCAPITALS contains the value 14.

CEILING or CEIL

Returns the smallest integer value greater than or equal to the argument.

```
►►──┬─CEILING─┬──(──expression──)────────────────────────────────────►◄
    └─CEIL────┘
```

The schema is SYSIBM. (The SYSFUN version of the CEILING function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes

- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - CEILING(NaN) returns NaN.
  - CEILING(-NaN) returns -NaN.
  - CEILING(Infinity) returns Infinity.
  - CEILING(-Infinity) returns -Infinity.
  - CEILING(sNaN) returns NaN and a warning.
  - CEILING(-sNaN) returns -NaN and a warning.

CHAR

The CHAR function returns a fixed-length character string representation of a variety of input data types.

Integer to character

►►──CHAR──(──*integer-expression*──)──────────────────────────►◄

Decimal to character

►►──CHAR──(──*decimal-expression*──┬──────────────────────┬──)────────────►◄
                                   └─,──*decimal-character*─┘

Floating-point to character

►►──CHAR──(──*floating-point-expression*──┬──────────────────────┬──)──────►◄
                                          └─,──*decimal-character*─┘

Decimal floating-point to character

►►──CHAR──(──*decimal-floating-point-expression*──┬──────────────────────┬──)────►◄
                                                  └─,──*decimal-character*─┘

Character to character

►►──CHAR──(──*character-expression*──┬──────────┬──)──────────────────►◄
                                     └─,──*integer*─┘

Graphic to character

►►──CHAR──(──*graphic-expression*──┬──────────┬──)────────────────────►◄
                                   └─,──*integer*─┘

Datetime to character

►►──CHAR──(──*datetime-expression*──┬──────┬──────┬──)────────────────────►◄
                                    └─,──┬─ISO───┬─┘
                                         ├─USA───┤
                                         ├─EUR───┤
                                         ├─JIS───┤
                                         └─LOCAL─┘

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:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a DECFLOAT
- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP

In a non-Unicode database, the string units of the result is OCTETS. Otherwise, the string units of the result are determined by the data type of the first argument.
- OCTETS, if the first argument is character string or a graphic string with string units of OCTETS, CODEUNITS16, or double bytes.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:
- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors because they might change in a future release.

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.

Integer to character

> *integer-expression*
>> An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).
>
> The result is a fixed-length character string representation of *integer-expression* in the form of an SQL integer constant. The result consists of *n* characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left-aligned.
>> - If the 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 single-byte blanks.
>
> The code page of the result is the code page of the section.

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 a fixed-length character string representation of *decimal-expression* in the form of an SQL decimal constant. The length of the result is 2+*p*, where *p* is the precision of *decimal-expression*. Leading zeros are not included. Trailing zeros are included. If *decimal-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of bytes in the result is less than the defined length of the result, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Floating-point to 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 a fixed-length character string representation of *floating-point-expression* in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If *floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *floating-point-expression* is zero, the result is 0E0. If the number of bytes in the result is less than 24, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Decimal floating-point to 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.

The result is a fixed-length character string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings 'INFINITY', 'SNAN', and 'NAN', respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in warning when converted to a string. If the number of characters in the result is less than 42, the result is padded on the right with single-byte blanks.

The code page of the result is the code page of the section.

Character to character

character-expression
> An expression that returns a value that is a built-in character string data type.

integer
> An integer constant that specifies the length attribute for the resulting fixed-length character string. The value must be between 0 and the maximum length for the CHAR data type in the string units of the result.
>
> If the second argument is not specified:
> - If the *character-expression* is the empty string constant, the length attribute of the result is 0.
> - Otherwise, the length attribute of the result is the lower of the following values:
>   - The maximum length for the CHAR data type in the string units of the result
>   - The length attribute of the first argument.

The result is a fixed-length character string. The length attribute of the result is determined by the value of integer. If *character-expression* is FOR BIT DATA, the result is FOR BIT DATA.

The actual length of the result is the same as the length attribute of the result. If the length of *character-expression* is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result. If the length of *character-expression* is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). When part of a multi-byte character is truncated, that partial character is replaced with the blank character. Do not rely on this behavior because it might change in a future release.

- If *integer* is not specified and *character-expression* is VARCHAR, truncation behavior is as follows:
  - If only blank characters must be truncated, truncation is performed with no warning returned.
  - If non-blank characters must be truncated, an error is returned (SQLSTATE 22001).
- If *integer* is not specified and *character-expression* is CLOB, an error is returned (SQLSTATE 22001).

Graphic to character

graphic-expression
    An expression that returns a value that is a built-in graphic string data type.

integer
    An integer constant that specifies the length attribute for the resulting fixed-length character string. The value must be between 0 and the maximum length for the CHAR data type in the string units of the result.

    If the second argument is not specified:

- If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
- If the string units of *graphic-expression* is CODEUNITS32, the length attribute of the result is the lower of the following values:
  - The maximum length for the CHAR data type in the string units of the result.
  - The length attribute of the first argument.
- Otherwise, the length attribute of the result is the lower of the following values:
  - The maximum length for the CHAR data type in the string units of the result.
  - 3 * length attribute of the first argument.

The result is a fixed-length character string that is converted from *graphic-expression*. The length attribute of the result is determined by the value of *integer*.

The actual length of the result is the same as the length attribute of the result.

If the length of *graphic-expression* that is converted to a character string is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result.

If the length of *graphic-expression* that is converted to a character string is greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified and *graphic-expression* is a GRAPHIC or VARGRAPHIC, truncation is performed with no warning returned.
- If *integer* is specified and graphic-expression is a DBCLOB, truncation is performed with a warning returned (SQLSTATE 01004).

- If *integer* is not specified, an error is returned (SQLSTATE 22001).

Datetime to character

> *datetime-expression*
>> An expression that is of one of the following data types:
>>
>> **DATE** The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).
>>
>> **TIME** The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).
>>
>> **TIMESTAMP**
>>> The result is the character string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).
>
> The code page of the result is the code page of the section.

Notes

- The CAST specification should be used to increase the portability of applications when the first argument is numeric, or the first argument is a string and the length argument is specified. For more information, see "CAST specification".
- A binary string is allowed as the first argument to the function, and the resulting fixed-length string is a FOR BIT DATA character string, padded with blanks if necessary.
- **Decimal to character and leading zeros**: In versions previous to version 9.7, the result for decimal input to this function includes leading zeros and a trailing decimal character. The database configuration parameter *dec_to_char_fmt* can be set to "V95" to have this function return the version 9.5 result for decimal input. The default value of the *dec_to_char_fmt* database configuration parameter for new databases is "NEW", which has this function return results which match the SQL standard casting rules and is consistent with results from the VARCHAR function.

Examples

- *Example 1*: Assume that the PRSTDATE column has an internal value equivalent to 1988-12-25. The following function returns the value '12/25/1988'.

      CHAR(PRSTDATE, USA)

- *Example 2*: Assume that the STARTING column has an internal value equivalent to 17:12:30, and that the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours). The following function returns the value '5:12 PM'.

      CHAR(STARTING, USA)

  The following function returns the value '10:12 PM'.

      CHAR(STARTING + :HOUR_DUR, USA)

- *Example 3*: Assume that the RECEIVED column (TIMESTAMP) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns. The following function returns the value '1988-12-25-17.12.30.000000'.

  CHAR(RECEIVED)

- *Example 4*: The LASTNAME column is defined as VARCHAR(15). The following function returns the values in this column as fixed-length character strings that are 10 bytes long. LASTNAME values that are more than 10 bytes long (excluding trailing blanks) are truncated and a warning is returned.

  SELECT CHAR(LASTNAME,10**) FROM** EMPLOYEE

- *Example 5*: The EDLEVEL column is defined as SMALLINT. The following function returns the values in this column as fixed-length character strings. An EDLEVEL value of 18 is returned as the CHAR(6) value '18' followed by four blanks.

  SELECT CHAR(EDLEVEL**) FROM** EMPLOYEE

- *Example 6*: The SALARY column is defined as DECIMAL with a precision of 9 and a scale of 2. The current value (18357.50) is to be displayed with a comma as the decimal character (18357,50). The following function returns the value '18357,50' followed by three blanks.

  CHAR(SALARY, ','**)**

- *Example 7*: Values in the SALARY column are to be subtracted from 20000.25 and displayed with the default decimal character. The following function returns the value '-0001642.75' followed by three blanks.

  CHAR(20000.25 - SALARY**)**

- *Example 8*: Assume that the host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. The following function returns the value '10000.00'.
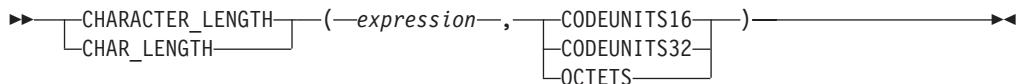
  CHAR(DECIMAL(:SEASONS_TICKETS,7,2)**)**

- *Example 9*: Assume that the host variable DOUBLE_NUM is defined as DOUBLE and has a value of -987.654321E-35. The following function returns the value '-9.87654321E-33' followed by nine trailing blanks because the result data type is CHAR(24).

  CHAR(:DOUBLE_NUM)

CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the length of *expression* in the specified string unit.

```
►►──┬─CHARACTER_LENGTH─┬──(─expression─,─┬─CODEUNITS16─┬─)──────────────►◄
    └─CHAR_LENGTH──────┘                 ├─CODEUNITS32─┤
                                         └─OCTETS──────┘
```

The schema is SYSIBM.

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 in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'
UTF-32BE	X'0001D11E'	X'0000004E'	X'00000303'	X'00000041'	X'00000042'

CHARACTER_LENGTH

Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT CHARACTER_LENGTH(UTF8_VAR, CODEUNITS16),
  CHARACTER_LENGTH(UTF8_VAR, CODEUNITS32),
  CHARACTER_LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT CHARACTER_LENGTH(UTF16_VAR, CODEUNITS16),
  CHARACTER_LENGTH(UTF16_VAR, CODEUNITS32),
  CHARACTER_LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.

CHR

Returns the character that has the ASCII code value specified by the argument.

►►──CHR──(──*expression*──)──►◄

The schema is SYSFUN.

expression
An expression that returns a value of INTEGER or SMALLINT data type.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value. If the argument value is between 1 and 255, the result is the character that has the ASCII code value corresponding to the argument. If the argument value is 0, the result is the blank character (X'20'). Otherwise the result is the same as CHR(255).

CLOB

The CLOB function returns a CLOB representation of a character string type.

►►—CLOB—(—*character-string-expression*——————————)————————————►◄
                                          └—,—*integer*—┘

The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first
converted to a character string data type before the function is executed.

character-string-expression
> An expression that returns a value that is a character string. The expression
> cannot be a character string defined as FOR BIT DATA (SQLSTATE 42846).

integer
> An integer value specifying the length attribute of the resulting CLOB data
> type. If the *character-string-expression* string unit is OCTETS, the value must be
> between 0 and 2 147 483 647. If the *character-string-expression* string unit is
> CODEUNITS32, the value must be between 0 and 536 870 911. If a value for
> *integer* is not specified, the length of the result is the same as the length of the
> first argument.

The result of the function is a CLOB in the string units of *character-string-expression*.
If the argument can be null, the result can be null; if the argument is null, the
result is the null value.

COALESCE

COALESCE returns the first argument that is not null.

```
>>—COALESCE—(—expression1——,—expression1——)—————————————><
```

The schema is SYSIBM.

expression1
> An expression that returns a value of any built-in or user-defined data type.

expression2
> An expression that returns a value of any built-in or user-defined data type that is compatible with the data type other arguments according to the rules for result data types.

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 attributes of the result are determined by all the operands as explained in "Rules for result data types".

Notes

- The COALESCE function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Examples

- *Example 1*: When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of 'ABSENT'.

  ```
  SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
    FROM DEPARTMENT
  ```

- *Example 2*: When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

  ```
  SELECT EMPNO, COALESCE(SALARY, 0)
    FROM EMPLOYEE
  ```

COLLATION_KEY_BIT

The COLLATION_KEY_BIT function returns a VARCHAR FOR BIT DATA string that represents the collation key of the *string-expression* in the specified *collation-name*.

```
►►──COLLATION_KEY_BIT──(──string-expression──,──collation-name─────────)────►◄
                                                              └─, length─┘
```

The schema is SYSIBM.

The results of COLLATION_KEY_BIT for two strings can be binary compared to determine their order within the specified *collation-name*. For the comparison to be meaningful, the results used must be from the same *collation-name*.

string-expression
> An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string for which the collation key should be determined. If *string-expression* is a CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 429BM).

> If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16. If the result of the code page conversion contains at least one substitution character, this function returns a collation key of the UTF-16 string with the substitution character or characters and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

collation-name
> A character constant that specifies the collation to use when determining the collation key.

> The value of *collation-name* is not case sensitive and must be one of the "Unicode Collation Algorithm-based collations" in *Globalization Guide* or "language-aware collations for Unicode data" in *Globalization Guide* (SQLSTATE 42616).

length
> An expression that specifies the length attribute of the result in bytes. If specified, *length* must be an integer between 1 and 32 672 (SQLSTATE 42815).

If a value for *length* is not specified, the length of the result is determined as follows:

Table 49. Determining the result length

String Argument Data Type	Result Data Type Length
CHAR(*n*) or VARCHAR(*n*)	Minimum of 12*n* bytes and 32 672 bytes
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of 12*n* bytes and 32 672 bytes

Regardless of whether *length* is specified or not, if the length of the collation key is longer than the length of the result data type, an error is returned (SQLSTATE 42815). The actual result length of the collation key is approximately six times the length of *string-expression* after it is converted to UTF-16.

If *string-expression* is an empty string, the result is a valid collation key that can have a nonzero length.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: The following query orders employees by their family names by using the language-aware collation for German in code page 923:

```
SELECT FIRSTNME, LASTNAME
  FROM EMPLOYEE
  ORDER BY COLLATION_KEY_BIT (LASTNAME, 'SYSTEM_923_DE')
```

- *Example 2*: The following query uses a culturally correct comparison to find the departments of employees in the province of Québec:

```
SELECT E.WORKDEPT
  FROM EMPLOYEE AS E INNER JOIN SALES AS S
  ON COLLATION_KEY_BIT(E.LASTNAME, 'CLDR181_LFR') =
    COLLATION_KEY_BIT(S.SALES_PERSON, 'CLDR181_LFR')
  WHERE S.REGION = 'Quebec'
```

COMPARE_DECFLOAT

The COMPARE_DECFLOAT function returns a SMALLINT value that indicates whether the two arguments are equal or unordered, or whether one argument is greater than the other.

►►──COMPARE_DECFLOAT──(──*expression1*──,──*expression2*──)──────────────────────►◄

The schema is SYSIBM.

expression1
>An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

expression2
>An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

The value of *expression1* is compared with the value of *expression2*, and the result is returned according to the following rules:

- If both arguments are finite, the comparison is algebraic and follows the procedure for decimal floating-point subtraction. If the difference is exactly zero with either sign, the arguments are equal. If a nonzero difference is positive, the first argument is greater than the second argument. If a nonzero difference is negative, the first argument is less than the second.
- Positive zero and negative zero compare as equal.
- Positive infinity compares equal to positive infinity.
- Positive infinity compares greater than any finite number.
- Negative infinity compares equal to negative infinity.
- Negative infinity compares less than any finite number.
- Numeric comparison is exact. The result is determined for finite operands as if range and precision were unlimited. No overflow or underflow condition can occur.
- If either argument is NaN or sNaN (positive or negative), the result is unordered.

The result value is as follows:

- 0 if the arguments are exactly equal
- 1 if *expression1* is less than *expression2*
- 2 if *expression1* is greater than *expression2*
- 3 if the arguments are unordered

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

The following examples show the values that are returned by the COMPARE_DECFLOAT function, given a variety of input decimal floating-point values:

```
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.17)) = 0
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(2.170)) = 2
COMPARE_DECFLOAT(DECFLOAT(2.170), DECFLOAT(2.17)) = 1
COMPARE_DECFLOAT(DECFLOAT(2.17), DECFLOAT(0.0)) = 2
COMPARE_DECFLOAT(INFINITY,INFINITY) = 0
COMPARE_DECFLOAT(INFINITY,-INFINITY) = 2
COMPARE_DECFLOAT(DECFLOAT(-2),INFINITY) = 1
COMPARE_DECFLOAT(NAN,NAN) = 3
COMPARE_DECFLOAT(DECFLOAT(-0.1),SNAN) = 3
```

CONCAT

The CONCAT function combines two arguments to form a string expression.

►►──CONCAT──(──*expression1*──,──*expression2*──)────────────────────────────◄

The schema is SYSIBM.

expression1
> An expression that returns a value of any string data type, any numeric data type, or any datetime data type.

expression2
> An expression that returns a value of any string data type, any numeric data type, or any datetime data type. However, some data types are not supported in combination with the data type of *expression1*, as described in the remainder of this topic.

The arguments can be any combination of string (except binary string), numeric, and datetime values. When an argument is a non-string value, it is implicitly cast to VARCHAR. A binary string can only be concatenated with another binary string. However, through the castable process of function resolution, a binary string can be concatenated with a character string defined as FOR BIT DATA when the first argument is the binary string.

Concatenation involving both a character string argument and a graphic string argument is supported only in a Unicode database. The character argument is first converted to the graphic data type before the concatenation. Character strings defined as FOR BIT DATA cannot be cast to the graphic data type. If an argument is defined with CODEUNITS32, the other argument cannot be defined as FOR BIT DATA.

The result of the function is a string that consists of the first argument followed by the second argument. The data type and the length of the result is determined by the data types and lengths of the arguments, after any applicable casting is done. For more information, refer to the "Data Type and Length of Concatenated Operands" table in "Expressions" on page 239.

If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Notes
- No check is made for improperly formed mixed data when doing concatenation.
- The CONCAT function is identical to the CONCAT operator. For more information, see "Expressions" on page 239.

Example

Concatenate the column FIRSTNME with the column LASTNAME.
```
SELECT CONCAT(FIRSTNME, LASTNAME)
FROM EMPLOYEE
WHERE EMPNO = '000010'
```

Returns the value "CHRISTINEHAAS".

COS

Returns the cosine of the argument, where the argument is an angle expressed in radians.

►►──COS──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the COS function continues to be available.)

expression
  An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

COSH

Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

►►──COSH──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

COT

Returns the cotangent of the argument, where the argument is an angle expressed in radians.

►►──COT──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the COT function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

CURSOR_ROWCOUNT

The CURSOR_ROWCOUNT function returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.

►►──CURSOR_ROWCOUNT──(──*cursor-variable-name*──)──────────────────────────►◄

The schema is SYSIBM.

cursor-variable-name
> The name of a SQL variable or SQL parameter of a cursor type. The underlying cursor of the *cursor-variable-name* must be open (SQLSTATE 24501).

The result is 0 if no FETCH action on the underlying cursor of the cursor-variable-name was performed before the evaluation of the function.

This function can only be used within a compound SQL (compiled) statement.

The data type of the result is BIGINT. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

The following example shows how to use the function to retrieve the count of rows associated with the cursor *curEmp* and assign it to a variable named *rows_fetched*:

```
SET rows_fetched = CURSOR_ROWCOUNT(curEmp);
```

DATAPARTITIONNUM

The DATAPARTITIONNUM function returns the sequence number (SYSDATAPARTITIONS.SEQNO) of the data partition in which the row resides.

►►──DATAPARTITIONNUM──(──*column-name*──)──────────────────────────────────────►◄

The schema is SYSIBM.

column-name
> The qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.
>
> If *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.

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 data type of the result is INTEGER and is never null.

Notes
- This function cannot be used as a source function when creating a user-defined function. Because the function accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The DATAPARTITIONNUM function cannot be used within check constraints or in the definition of generated columns (SQLSTATE 42881). The DATAPARTITIONNUM function cannot be used in a materialized query table (MQT) definition (SQLSTATE 428EC).
- The DATAPARTITIONNUM function cannot be used as part of an expression-based key in a CREATE INDEX statement.

Examples
- *Example 1:* Retrieve the sequence number of the data partition in which the row for EMPLOYEE.EMPNO resides.

  ```
  SELECT DATAPARTITIONNUM (EMPNO)
  FROM EMPLOYEE
  ```

- *Example 2:* To convert a sequence number that is returned by DATAPARTITIONNUM (for example, 0) to a data partition name that can be used in other SQL statements (such as ALTER TABLE...DETACH PARTITION), you can query the SYSCAT.DATAPARTITIONS catalog view. Include the SEQNO obtained from DATAPARTITIONNUM in the WHERE clause, as shown in the following example.

  ```
  SELECT DATAPARTITIONNAME
  FROM SYSCAT.DATAPARTITIONS
  WHERE TABNAME = 'EMPLOYEE' AND SEQNO = 0
  ```

  results in the value 'PART0'.

DATE

The DATE function returns a date from a value.

►►──DATE──(──*expression*──)──►◄

The schema is SYSIBM.

expression
>An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, numeric, or character string that is not a CLOB.

>A value with a numeric data type must be a positive number with an integral value less than or equal to 3 652 059.

>A character string must be a valid string representation of a date or timestamp or a string of length 7. If the value is a string of length 7, it must represent a valid date in the form *yyyynnn* , where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366, denoting a day of that year.

>In a Unicode database, if an expression returns a value of a graphic string data type, the value is first converted to a character string before the function is executed.

The result of the function is a DATE. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the date part of the value.
- If the argument is a number:
  - The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a string with a length of 7:
  - The result is the date represented by the string.

Examples

Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-25-17.12.30.000000'.
- *Example 1*: This example results in an internal representation of '1988-12-25'.
  > **DATE**(RECEIVED)
- *Example 2*: This example results in an internal representation of '1988-12-25'.
  > **DATE**('1988-12-25')
- *Example 3*: This example results in an internal representation of '1988-12-25'.
  > **DATE**('25.12.1988')
- *Example 4*: This example results in an internal representation of '0001-02-04'.
  > **DATE**(35)

DAY

The DAY function returns the day part of a value.

►►──DAY──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, numeric, or character string that is not a CLOB.
>
> If the value is a number, it must be a date duration or timestamp duration (SQLSTATE 42815).
>
> If the value is a character string, it must be a valid string representation of a date or timestamp. In a Unicode database, if the value is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:
  - The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

- *Example 1*: Using the PROJECT table, set the host variable END_DAY (smallint) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

  ```
  SELECT DAY(PRENDATE)
    INTO :END_DAY
    FROM PROJECT
    WHERE PROJNAME = 'WELD LINE PLANNING'
  ```

  Results in END_DAY being set to 15 when using the sample table.

- *Example 2*: Assume that the column DATE1 (whose data type is DATE) has an internal value equivalent to 2000-03-15 and the column DATE2 (whose data type is DATE) has an internal value equivalent to 1999-12-31.

  ```
  DAY(DATE1 - DATE2)
  ```

  Results in the value 15.

DAYNAME

The DAYNAME function returns a character string containing the name of the day (for example, Friday) for the day portion of the input value.

►►──DAYNAME──(──*expression*──────────────────)────────────────────►◄
                              └─,──*locale-name*─┘

The schema is SYSIBM. The SYSFUN version of the DAYNAME function continues to be available

The character string returned is based on *locale-name* or the value of the special register CURRENT LOCALE LC_TIME.

expression
An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

locale-name
A character constant that specifies the locale used to determine the language of the result. The value of *locale-name* is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the *expression* argument can be null, the result can be null; if the *expression* argument is null, the result is the null value. The code page of the result is the code page of the section. The string units of the result is determined by the string units of the environment

Notes

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the DAYNAME function assumes the Gregorian calendar for all calculations.
- **Determinism**: DAYNAME is a deterministic function. However, when *locale-name* is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Assume that the variable TMSTAMP is defined as TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

```
Function invocation                      Result
-------------------------                ----------
DAYNAME (TMSTAMP, 'CLDR181_en_US')       Friday
DAYNAME (TSMTAMP, 'CLDR181_de_DE')       Freitag
DAYNAME (TMSTAMP, 'CLDR181_fr_FR')       vendredi
```

DAYOFWEEK

The DAYOFWEEK function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.

►►──DAYOFWEEK──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYOFWEEK_ISO

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.

►►──DAYOFWEEK_ISO──(──*expression*──)────────────────────────────►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYOFYEAR

Returns the day of the year in the argument as an integer value in the range 1-366.

▶▶──DAYOFYEAR──(──*expression*──)──▶◀

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYS

The DAYS function returns an integer representation of a date.

►►──DAYS──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Examples

- *Example 1*: Using the PROJECT table, set the host variable EDUCATION_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

  ```
  SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
    INTO :EDUCATION_DAYS
    FROM PROJECT
    WHERE PROJNO = 'IF2000'
  ```

  Results in EDUCATION_DAYS being set to 396.

- *Example 2*: Using the PROJECT table, set the host variable TOTAL_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

  ```
  SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
    INTO :TOTAL_DAYS
    FROM PROJECT
    WHERE DEPTNO = 'E21'
  ```

  Results in TOTAL_DAYS being set to 1584 when using the sample table.

DBCLOB

The DBCLOB function returns a DBCLOB representation of a graphic string type.

```
►►──DBCLOB──(──graphic-expression──┬──────────────┬──)──────────────────►◄
                                   └─,──integer──┘
```

The schema is SYSIBM.

graphic-expression
   An expression that returns a value that is a graphic string.

integer
   An integer value specifying the length attribute of the resulting DBCLOB data
   type. The value must be between 0 and 1 073 741 823 if the *graphic-expression*
   string unit is double bytes or CODEUNITS16, or between 0 and 536 870 911 if
   graphic-expression string unit is CODEUNITS32. If *integer* is not specified, the
   length of the result is the same as the length of the first argument.

In a Unicode database, if a supplied argument is a character string, it is first
converted to a graphic string before the function is executed. When the output
string is truncated, such that the last character is a high surrogate, that surrogate is
either:

- Left as is, if the supplied argument is a character string
- Converted to the blank character (X'0020'), if the supplied argument is a graphic
  string

Do not rely on these behaviors, because they might change in a future release.

The result of the function is a DBCLOB in the string units of *graphic-expression*. If
the argument can be null, the result can be null; if the argument is null, the result
is the null value.

DBPARTITIONNUM

The DBPARTITIONNUM function returns the database partition number for a row. For example, if used in a SELECT clause, it returns the database partition number for each row in the result set.

►►─── DBPARTITIONNUM─(─*column-name*─)───────────────────────────────────────►◄

The schema is SYSIBM.

column-name
> The qualified or unqualified name of any column in the table. Because row-level information is returned, the result is the same regardless of which column is specified. The column can have any data type.

> If *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.

Notes
- The DBPARTITIONNUM function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).
- The DBPARTITIONNUM function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The DBPARTITIONNUM function cannot be used as part of an expression-based key in a CREATE INDEX statement.
- *Syntax alternatives:* For compatibility with previous versions of DB2 products, the function name NODENUMBER is a synonym for DBPARTITIONNUM.

Examples
- *Example 1*: Count the number of instances in which the row for a given employee in the EMPLOYEE table is on a different database partition than the description of the employee's department in the DEPARTMENT table.
  ```
  SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.DEPTNO=E.WORKDEPT
    AND DBPARTITIONNUM(E.LASTNAME) <> DBPARTITIONNUM(D.DEPTNO)
  ```

DBPARTITIONNUM

- *Example 2*: Join the EMPLOYEE and DEPARTMENT tables so that the rows of the two tables are on the same database partition.

```
SELECT * FROM DEPARTMENT D, EMPLOYEE E
   WHERE DBPARTITIONNUM(E.LASTNAME) = DBPARTITIONNUM(D.DEPTNO)
```
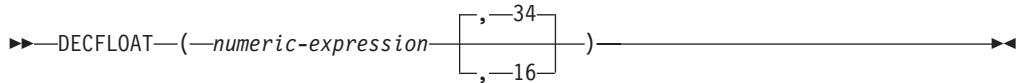
- *Example 3*: Using a before trigger on the EMPLOYEE table, log the employee number and the projected database partition number of any new row in the EMPLOYEE table in a table named EMPINSERTLOG1.

```
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG1
VALUES(NEWTABLE.EMPNO, DBPARTITIONNUM
(NEWTABLE.EMPNO))
```
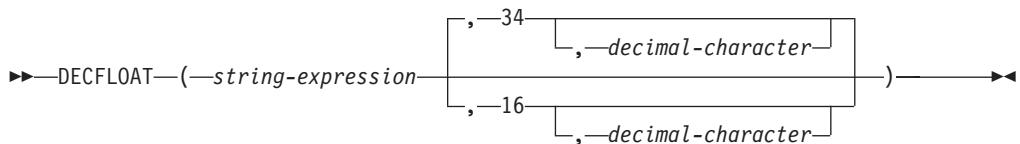
DECFLOAT

The DECFLOAT function returns a decimal floating-point representation of a number or a string representation of a number.

Numeric to Decimal floating-point

```
>>--DECFLOAT--(--numeric-expression--+----------+--)----------------><
                                      | ,--34    |
                                      +--,--16---+
```

Character to Decimal floating-point

```
>>--DECFLOAT--(--string-expression--+-,--34-------------------------+--)----><
                                    |          +--,--decimal-character--+ |
                                    +-,--16------------------------------+
                                               +--,--decimal-character--+
```

The schema is SYSIBM.

numeric-expression
>   An expression that returns a value of any built-in numeric data type.

string-expression
>   An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant. The data type of string-expression must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are removed from the string. The resulting substring is folded to uppercase and must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820).

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.

Notes
- The CAST specification should be used to increase the portability of applications.

DECFLOAT

- 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
```

DECFLOAT_FORMAT

The DECFLOAT_FORMAT function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

►►──DECFLOAT_FORMAT──(──*string-expression*─────────────────────)───────────►◄
                                        └─,──*format-string*─┘

The schema is SYSIBM.

string-expression
> An expression that returns a value that is a CHAR and VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.Leading and trailing blanks are removed from the string. If *format-string* is not specified, the resulting substring must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and not be greater than 42 bytes (SQLSTATE 42820); otherwise, the resulting substring must contain the components of a number that correspond to the format specified by *format-string* (SQLSTATE 22018).

format-string
> An expression that returns a value that is a built-in character string data type (except CLOB). In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22018). The value is a template for how *string-expression* is to be interpreted for conversion to a DECFLOAT value. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The format elements are case sensitive. The template must not contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

Table 50. Format elements for the DECFLOAT_FORMAT function

Format element	Description
0 or 9	Each 0 or 9 represents a digit.
MI	Suffix: If *string-expression* is to represent a negative number, a trailing minus sign (-) is expected. If *string-expression* is to represent a positive number, a trailing blank can be specified.
S	Prefix: If *string-expression* is to represent a negative number, a leading minus sign (-) is expected. If *string-expression* is to represent a positive number, a leading plus sign (+) or leading blank can be specified.
PR	Suffix: If *string-expression* is to represent a negative number, a leading less than character (<) and a trailing greater than character (>) are expected. If *string-expression* is to represent a positive number, a leading space and a trailing space can be specified.
$	Prefix: A leading dollar sign ($) must be specified.
,	Specifies the expected location of a comma. This comma is used as a group separator.
.	Specifies the expected location of the period. This period is used as a decimal point.

If *format-string* is not specified, *string-expression* must conform to the rules for forming an SQL integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018) and have a length not greater than 42 bytes (SQLSTATE 42820).

The result is a DECFLOAT(34). If the first or second argument can be null, the result can be null; if the first or second argument is null, the result is the null value.

Notes

- Syntax alternatives: TO_NUMBER is a synonym for DECFLOAT_FORMAT.
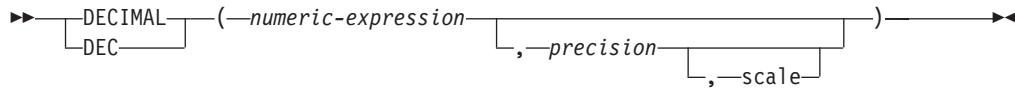
Examples

- *Example 1*: The following example returns 123.45
  DECFLOAT_FORMAT('123.45')
- *Example 2*: The following example returns -123456.78
  DECFLOAT_FORMAT('-123456.78')
- *Example 3*: The following example returns 123456.78
  DECFLOAT_FORMAT('+123456.78')
- *Example 4*: The following example returns 12300
  DECFLOAT_FORMAT('1.23E4')
- *Example 5*: The following example returns 123.40
  DECFLOAT_FORMAT('123.4', '9999.99')
- *Example 6*: The following example returns 1234
  DECFLOAT_FORMAT('001,234', '000,000')
- *Example 7*: The following example returns 1234
  DECFLOAT_FORMAT('1234 ', '9999MI')
- *Example 8*: The following example returns -1234
  DECFLOAT_FORMAT('1234-', '9999MI')
- *Example 9*: The following example returns 1234
  DECFLOAT_FORMAT('+1234', 'S9999')
- *Example 10*: The following example returns -1234
  DECFLOAT_FORMAT('-1234', 'S9999')
- *Example 11*: The following example returns 1234
  DECFLOAT_FORMAT(' 1234 ', '9999PR')
- *Example 12*: The following example returns -1234
  DECFLOAT_FORMAT('<1234>', '9999PR')
- *Example 13*: The following example returns 123456.78
  DECFLOAT_FORMAT('$123,456.78', '$999,999.99')
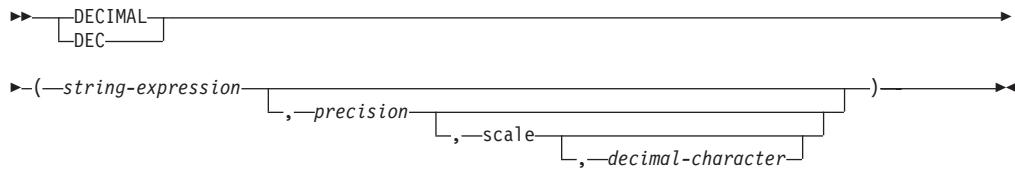
DECIMAL or DEC

The DECIMAL function returns a decimal representation of a number, a string representation of a number, or a datetime value

Numeric to Decimal

```
►►─┬─DECIMAL─┬─(─numeric-expression─┬──────────────────────────┬─)──────►◄
   └─DEC─────┘                      └─,─precision─┬──────────┬──┘
                                                 └─,─scale──┘
```

String to Decimal

```
►►─┬─DECIMAL─┬──────────────────────────────────────────────────────────►
   └─DEC─────┘

►─(─string-expression─┬────────────────────────────────────┬─)──────►◄
                      └─,─precision─┬──────────────────────┬┘
                                   └─,─scale─┬─────────────────────┬┘
                                            └─,─decimal-character──┘
```

Datetime to Decimal

```
►►─┬─DECIMAL─┬─(─datetime-expression─┬────────────────────────┬─)────►◄
   └─DEC─────┘                       └─,─precision─┬────────┬─┘
                                                  └─,─scale─┘
```
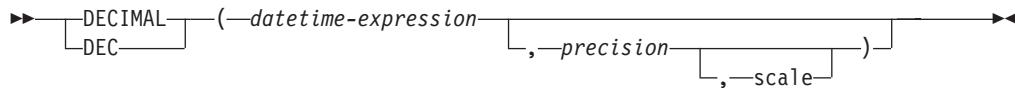
The schema is SYSIBM.

Numeric to Decimal

numeric-expression
> An expression that returns a value of any built-in numeric data type.

precision
> An integer constant with a value in the range of 1 to 31.
>
> The default for *precision* 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
> An integer constant in the range of 0 to the *precision* value. The default is zero.

> The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *precision* and a scale of *scale*. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant decimal digits required to represent the whole part of the number is greater than *precision - scale* (SQLSTATE 22003).

String to Decimal

string-expression
> An *expression* that returns a value that is a character string or a Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant. The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884). Leading and trailing blanks are eliminated from the string and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018).
>
> The *string-expression* is converted to the section code page if required to match the code page of the constant *decimal-character*.

precision
> An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default is 15.

scale
> An integer constant with a value in the range 0 to *precision* that specifies the scale of the result. If not specified, the default is 0.

decimal-character
> Specifies the single-byte character constant used to delimit the decimal digits in *string-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (-), or blank, and it can appear at most once in *string-expression* (SQLSTATE 42815).

The result is the same number that would result from CAST(*string-expression* AS DECIMAL(*precision*, *scale*)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *precision - scale* (SQLSTATE 22003). The default decimal character is not valid in the substring if a different value for the *decimal-character* argument is specified (SQLSTATE 22018).

Datetime to Decimal

datetime-expression
> An expression that returns a value of type DATE, TIME or TIMESTAMP.

precision
> An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default for the precision and scale depends on the data type of *datetime-expression* as follows:
> - Precision is 8 and scale is 0 for a DATE. The result is a DECIMAL(8,0) value representing the date as *yyyymmdd*.
> - Precision is 6 and scale is 0 for a TIME. The result is a DECIMAL(6,0) value representing the time as *hhmmss*.
> - Precision is $14+tp$ and scale is *tp* for a TIMESTAMP(*tp*). The result is a DECIMAL($14+tp,tp$) value representing the timestamp as *yyyymmddhhmmss.nnnnnnnnnnnn*.

scale
> An integer constant with a value in the range 0 to *precision* that specifies the scale of the result. If not specified and a *precision* is specified, the default is 0.

The result is the same number that would result from CAST(*datetime - expression* AS DECIMAL(*precision*, *scale*)). Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal separator character is greater than the scale *scale*. An error is returned if the number of significant digits to the left of the decimal character (the whole part of the number) in *string-expression* is greater than *precision - scale* (SQLSTATE 22003).

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Note: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification".

Examples

- *Example 1*: Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

  ```
  SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
    FROM EMPLOYEE
  ```

- *Example 2*: Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as decimal(8,0).

  ```
  SELECT PRSTDATE + DECIMAL(:PERIOD,8)
    FROM PROJECT
  ```

- *Example 3*: Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable *newsalary* which is defined as CHAR(10).

  ```
  UPDATE STAFF
    SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
    WHERE ID = :empid;
  ```

  The value of *newsalary* becomes 21400.50.

- *Example 4*: Add the default decimal character (.) to a value.

  ```
  DECIMAL('21400,50', 9, 2, '.')
  ```

  This fails because a period (.) is specified as the decimal character, but a comma (,) appears in the first argument as a delimiter.

- *Example 5*: Assume that the column STARTING (whose data type is TIME) has an internal value equivalent to '12:10:00'.

  ```
  DECIMAL(STARTING)
  ```

  results in the value 121 000.

- *Example 6*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

  ```
  DECIMAL(RECEIVED)
  ```

  results in the value 19 881 222 140 721.136421.

- *Example 7*: This example shows the decimal result and resulting precision and scale for various datetime input values. Assume the existence of the following columns with associated values:
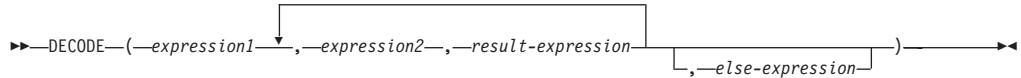
Column name	Data type	Value
ENDDT	DATE	2000-03-21
ENDTM	TIME	12:02:21
ENDTS	TIMESTAMP	2000-03-21-12.02.21.123456
ENDTS0	TIMESTAMP(0)	2000-03-21-12.02.21
ENDTS9	TIMESTAMP(9)	2000-03-21-12.02.21.123456789

The following table shows the decimal result and resulting precision and scale for various datetime input values.

DECIMAL(arguments)	Precision and Scale	Result
DECIMAL(ENDDT)	(8,0)	20000321.
DECIMAL(ENDDT, 10)	(10,0)	20000321.
DECIMAL(ENDDT, 12, 2)	(12,2)	20000321.00
DECIMAL(ENDTM)	(6,0)	120221.
DECIMAL(ENDTM, 10)	(10,0)	120221.
DECIMAL(ENDTM, 10, 2)	(10,2)	120221.00
DECIMAL(ENDTS)	(20, 6)	20000321120221.123456
DECIMAL(ENDTS, 23)	(23, 0)	20000321120221.
DECIMAL(ENDTS, 23, 4)	(23, 4)	20000321120221.1234
DECIMAL(ENDTS0)	(14,0)	20000321120221.
DECIMAL(ENDTS9)	(23,9)	20000321120221.123456789

DECODE

The DECODE function does equality comparisons between arguments, also treating null values as equal, to determine which argument to return as the result.

▶▶──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-expresssion* is returned. If no *expression2* matches *expression1*, the value of *else-expression* is returned; otherwise a null value is returned.

The DECODE function is similar to the CASE expression except for the handling of null values:

- A null value of *expression1* will match a corresponding null value of *expression2*.
- If the NULL keyword is used as an argument in the DECODE function, it must be cast to an appropriate data type.

The rules for determining the result type of a DECODE expression are based on the corresponding CASE expression.

Examples

- *Example 1*: The DECODE expression:

```
DECODE (c1, 7, 'a', 6, 'b', 'c')
```

achieves the same result as the following CASE expression:

```
CASE c1
  WHEN 7 THEN 'a'
  WHEN 6 THEN 'b'
  ELSE 'c'
END
```

- *Example 2*: The DECODE expression:

```
DECODE (c1, var1, 'a', var2, 'b')
```

where the values of c1, var1, and var2 could be null values, achieves the same result as the following CASE expression:

```
CASE
  WHEN c1 = var1 OR (c1 IS NULL AND var1 IS NULL) THEN 'a'
  WHEN c1 = var2 OR (c1 IS NULL AND var2 IS NULL) THEN 'b'
  ELSE NULL
END
```

- *Example 3*: Consider also the following query:

```
SELECT ID, DECODE(STATUS, 'A', 'Accepted',
                          'D', 'Denied',
                          CAST(NULL AS VARCHAR(1)), 'Unknown',
                          'Other')
FROM CONTRACTS
```

Here is the same statement using a CASE expression:

```
SELECT ID,
  CASE
    WHEN STATUS = 'A' THEN 'Accepted'
```

```
        WHEN STATUS = 'D' THEN 'Denied'
        WHEN STATUS IS NULL THEN 'Unknown'
        ELSE 'Other'
    END
FROM CONTRACTS
```

DECRYPT_BIN and DECRYPT_CHAR

The DECRYPT_BIN and DECRYPT_CHAR functions both return a value that is the result of decrypting *encrypted-data*.

```
►►──┬─DECRYPT_BIN──┬──(──encrypted-data───────────────────────────────────)──►◄
    └─DECRYPT_CHAR─┘                     └─,──password-string-expression─┘
```

The schema is SYSIBM.

The password used for decryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the DECRYPT_BIN and DECRYPT_CHAR functions in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

The DECRYPT_BIN and DECRYPT_CHAR functions can only decrypt values that are encrypted using the ENCRYPT function (SQLSTATE 428FE).

encrypted-data
An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value as a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function.

password-string-expression
An expression that returns a CHAR or VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). This expression must be the same password used to encrypt the data (SQLSTATE 428FD). If the value of the password argument is null or not provided, the data will be decrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

The result of the DECRYPT_BIN function is VARCHAR FOR BIT DATA. The result of the DECRYPT_CHAR function is VARCHAR. If *encrypted-data* included a hint, the hint is not returned by the function. The length attribute of the result is the length of the data type of *encrypted-data* minus 8 bytes. The actual length of the value returned by the function will match the length of the original string that was encrypted. If *encrypted-data* includes bytes beyond the encrypted string, these bytes are not returned by the function.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

If the data is decrypted on a different system, which uses a code page that is different from the code page in which the data was encrypted, expansion might occur when converting the decrypted value to the database code page. In such situations, the *encrypted-data* value should be cast to a VARCHAR string with a larger number of bytes.

Examples

- *Example 1*: The following example demonstrates the use of the DECRYPT_CHAR function by showing code fragments from an embedded SQL application.

```
EXEC SQL BEGIN DECLARE SECTION;
      char hostVarCreateTableStmt[100];
      char hostVarSetEncPassStmt[200];
      char hostVarPassword[128];
      char hostVarInsertStmt1[200];
      char hostVarInsertStmt2[200];
      char hostVarSelectStmt1[200];
      char hostVarSelectStmt2[200];
EXEC SQL END DECLARE SECTION;

/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;

/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;
```

- *Example 2*: Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:

```
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;

/* prepare the statement */
strcpy(hostVarSelectStmt1, "SELECT DECRYPT_CHAR(SSN) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt1 FROM :hostVarSelectStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarSelectStmt1;
```

This query returns the value '289-46-8832'.

- *Example 3*: Pass the encryption password explicitly:

```
/* prepare the statement */
strcpy(hostVarInsertStmt2, "INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832',?)");
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarSelectStmt2, "SELECT DECRYPT_CHAR(SSN,?) FROM EMP");
EXEC SQL PREPARE hostVarSelectStmt2 FROM :hostVarSelectStmt2;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSelectStmt2 USING :hostVarPassword;
```

This query returns the value '289-46-8832'.

DEGREES

The DEGREES function returns the number of degrees of the argument, which is an angle expressed in radians.

►►──DEGREES──(──*expression*──)────────────────────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the DEGREES function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example

Assume that RAD is a DECIMAL(4,3) host variable with a value of 3.142.

    VALUES DEGREES(:RAD)

Returns the approximate value 180.0.

DEREF

The DEREF function returns an instance of the target type of the argument.

►►──DEREF──(──*expression*──)──►◄

expression
> An expression that returns a value with a reference data type that has a defined scope (SQLSTATE 428DT).

The static data type of the result is the target type of the argument. The dynamic data type of the result is a subtype of the target type of the argument. The result can be null. The result is the null value if *expression* is a null value or if *expression* is a reference that has no matching OID in the target table.

The result is an instance of the subtype of the target type of the reference. The result is determined by finding the row of the target table or target view of the reference that has an object identifier that matches the reference value. The type of this row determines the dynamic type of the result. Since the type of the result can be based on a row of a subtable or subview of the target table or target view, the authorization ID of the statement must have SELECT privilege on the target table and all of its subtables or the target view and all of its subviews (SQLSTATE 42501).

Example

Assume that EMPLOYEE is a table of type EMP, and that its object identifier column is named EMPID. Then the following query returns an object of type EMP (or one of its subtypes), for each row of the EMPLOYEE table (and its subtables). This query requires SELECT privilege on EMPLOYEE and all its subtables.

```
SELECT DEREF(EMPID) FROM EMPLOYEE
```

DIFFERENCE

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

```
►►──DIFFERENCE──(──expression──,──expression──)────────────────────►◄
```

The schema is SYSFUN.

expression
> 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

The following code:
```
VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONSTANT')),
(DIFFERENCE('CONSTRAINT','CONTRITE'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONTRITE'))
```

returns the following output:
```
1           2    3
----------- ---- ----
          4 C523 C523
          2 C523 C536
```

In the first row, the words have the same result from SOUNDEX while in the second row the words have only some similarity.

DIGITS

The DIGITS function returns a character-string representation of a number.

►►──DIGITS──(──*expression*──)──►◄

The schema is SYSIBM.

expression

An expression that returns a value of one of the following built-in data types: SMALLINT, INTEGER, BIGINT, DECIMAL, CHAR, or VARCHAR. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed. A CHAR or VARCHAR value is implicitly cast to DECIMAL(31,6) before evaluating the function.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal character. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*.

Examples

- *Example 1*: Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all distinct four digit combinations of the first four digits contained in column INTCOL.

  ```
  SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
     FROM TABLEX
  ```

- *Example 2*: Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28. Then, for this value:

  ```
  DIGITS(COLUMNX)
  ```
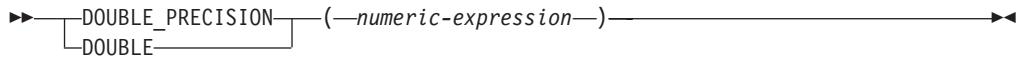
returns the value '000628'.

The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.
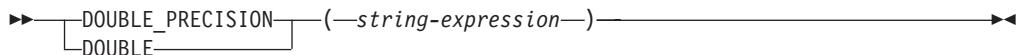
DOUBLE_PRECISION or DOUBLE

The DOUBLE_PRECISION and DOUBLE functions return a double-precision floating-point representation of either a number or a string representation of a number.

Numeric to double

```
►►──┬─DOUBLE_PRECISION─┬──(──numeric-expression──)──────────────────────►◄
    └─DOUBLE───────────┘
```

Character string to double

```
►►──┬─DOUBLE_PRECISION─┬──(──string-expression──)───────────────────────►◄
    └─DOUBLE───────────┘
```

The schema is SYSIBM.

Numeric to double

numeric-expression
>An expression that returns a value of any built-in numeric data type.
>
>The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable. If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

Character string to double

string-expression
>An expression that returns a value that is character-string or Unicode graphic-string representation of a number. The data type of *string-expression* must not be CLOB (SQLSTATE 42884).
>
>The result is the same number that would result from CAST(*string-expression* AS DOUBLE PRECISION). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of double-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- The CAST specification should be used to increase the portability of applications.
- FLOAT is a synonym for DOUBLE_PRECISION and DOUBLE.
- The SYSFUN version of DOUBLE (*string_expression*) continues to be available.

Example

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have

DOUBLE_PRECISION or DOUBLE

DECIMAL data types. To eliminate the possibility of out-of-range results, DOUBLE is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

EMPTY_BLOB, EMPTY_CLOB, EMPTY_DBCLOB, and EMPTY_NCLOB

These functions return a zero-length value with a data type of BLOB, CLOB, or DBCLOB.

```
►►──┬─EMPTY_BLOB───┬──(──)────────────────────────────────────────────────►◄
    ├─EMPTY_CLOB───┤
    ├─EMPTY_DBCLOB─┤
    └─EMPTY_NCLOB──┘
```

The schema is SYSIBM.

The empty value functions return a zero-length value of the associated data type. There are no arguments to these functions (the empty parentheses must be specified).
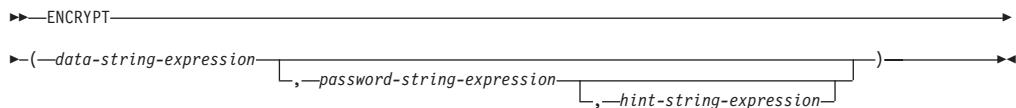
- The EMPTY_BLOB function returns a zero-length value with a data type of BLOB(1).
- The EMPTY_CLOB function returns a zero-length value with a data type of CLOB(1).
- The EMPTY_DBCLOB and EMPTY_NCLOB functions return a zero-length value with a data type of DBCLOB(1).

The result of these functions can be used in assignments to provide zero-length values where needed.

The EMPTY_NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

ENCRYPT

The ENCRYPT function returns a value that is the result of encrypting *data-string-expression*.

```
►►──ENCRYPT──────────────────────────────────────────────────────────────────►

►──(──data-string-expression──┬──────────────────────────────────────────┬──)──►◄
                              └─,──password-string-expression──┬────────────────────────────┘
                                                               └─,──hint-string-expression──┘
```

The schema is SYSIBM.

The password used for encryption is either the *password-string-expression* value or the encryption password value that was assigned by the SET ENCRYPTION PASSWORD statement. To maintain the best level of security on your system, it is recommended that you do not pass the encryption password explicitly with the ENCRYPT function in your query; instead, use the SET ENCRYPTION PASSWORD statement to set the password, and use a host variable or dynamic parameter markers when you use the SET ENCRYPTION PASSWORD statement, rather than a literal string.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

data-string-expression
> An expression that returns a CHAR or a VARCHAR value that is to be encrypted. The length attribute for the data type of *data-string-expression* is limited to 32663 without a *hint-string-expression* argument, and 32631 when the *hint-string-expression* argument is specified (SQLSTATE 42815).

password-string-expression
> An expression that returns a CHAR or a VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). The value represents the password used to encrypt *data-string-expression*. If the value of the password argument is null or not provided, the data is encrypted using the encryption password value that was assigned for the session by the SET ENCRYPTION PASSWORD statement (SQLSTATE 51039).

hint-string-expression
> An expression that returns a CHAR or a VARCHAR value with at most 32 bytes that will help data owners remember passwords (for example, 'Ocean' as a hint to remember 'Pacific'). If a hint value is given, the hint is embedded into the result and can be retrieved using the GETHINT function. If this argument is null or not provided, no hint will be embedded in the result.

The result data type of the function is VARCHAR FOR BIT DATA.
- When the optional hint parameter is specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary + 32 bytes for the length of the hint.
- When the optional hint parameter is not specified, the length attribute of the result is equal to the length attribute of the unencrypted data + 8 bytes + the number of bytes until the next 8-byte boundary.

If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Note that the encrypted result is longer than the *data-string-expression* value. Therefore, when assigning encrypted values, ensure that the target is declared with sufficient size to contain the entire encrypted value.

Notes

- *Encryption algorithm:* The internal encryption algorithm is RC2 block cipher with padding; the 128-bit secret key is derived from the password using an MD5 message digest.

- *Encryption passwords and data:* Password management is the user's responsibility. Once the data is encrypted, only the password that was used when encrypting it can be used to decrypt it (SQLSTATE 428FD).

  The encrypted result might contain null terminator and other unprintable characters. Any assignment or cast to a length that is shorter than the suggested data length might result in failed decryption in the future, and lost data. Blanks are valid encrypted data values that might be truncated when stored in a column that is too short.

- *Administration of encrypted data:* Encrypted data can only be decrypted on servers that support the decryption functions corresponding to the ENCRYPT function. Therefore, replication of columns with encrypted data should only be done to servers that support the DECRYPT_BIN or the DECRYPT_CHAR function.

Examples

- *Example 1*: The following example demonstrates the use of the ENCRYPT function by showing code fragments from an embedded SQL application.

```
EXEC SQL BEGIN DECLARE SECTION;
      char hostVarCreateTableStmt[100];
      char hostVarSetEncPassStmt[200];
      char hostVarPassword[128];
      char hostVarInsertStmt1[200];
      char hostVarInsertStmt2[200];
      char hostVarInsertStmt3[200];
EXEC SQL END DECLARE SECTION;

/* prepare the statement */
strcpy(hostVarCreateTableStmt, "CREATE TABLE EMP (SSN VARCHAR(24) FOR BIT DATA)");
EXEC SQL PREPARE hostVarCreateTableStmt FROM :hostVarCreateTableStmt;

/* execute the statement */
EXEC SQL EXECUTE hostVarCreateTableStmt;
```

- *Example 2*: Use the SET ENCRYPTION PASSWORD statement to set an encryption password for the session:

```
/* prepare the statement with a parameter marker */
strcpy(hostVarSetEncPassStmt, "SET ENCRYPTION PASSWORD = ?");
EXEC SQL PREPARE hostVarSetEncPassStmt FROM :hostVarSetEncPassStmt;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarSetEncPassStmt USING :hostVarPassword;

/* prepare the statement */
strcpy(hostVarInsertStmt1, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832')");
EXEC SQL PREPARE hostVarInsertStmt1 FROM :hostVarInsertStmt1;

/* execute the statement */
EXEC SQL EXECUTE hostVarInsertStmt1;
```

- *Example 3*: Pass the encryption password explicitly:

```
/* prepare the statement */
strcpy(hostVarInsertStmt2, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832',?)");
EXEC SQL PREPARE hostVarInsertStmt2 FROM :hostVarInsertStmt2;
```

```
/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt2 USING :hostVarPassword;
```

- *Example 4*: Define a password hint:

```
/* prepare the statement */
strcpy(hostVarInsertStmt3, "INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832',?,'Ocean')");
EXEC SQL PREPARE hostVarInsertStmt3 FROM :hostVarInsertStmt3;

/* execute the statement for hostVarPassword = 'Pac1f1c' */
strcpy(hostVarPassword, "Pac1f1c");
EXEC SQL EXECUTE hostVarInsertStmt3 USING :hostVarPassword;
```

EVENT_MON_STATE

The EVENT_MON_STATE function returns the current state of an event monitor.

►►──EVENT_MON_STATE──(──*string-expression*──)──────────────────────────────►◄

The schema is SYSIBM.

string-expression
　　An expression that returns a value of CHAR or VARCHAR data type. In a Unicode database, if the value is a graphic string, it is first converted to a character string before the function is executed. The value must be the name of an event monitor that is equal to an event monitor name in the SYSCAT.EVENMONITORS catalog view (SQLSTATE 42704).

The result is an integer with one of the following values:

0　　　　The event monitor is inactive.

1　　　　The event monitor is active.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

The following example selects all of the defined event monitors, and indicates whether each is active or inactive:

```
SELECT EVMONNAME,
  CASE
    WHEN EVENT_MON_STATE(EVMONNAME) = 0 THEN 'Inactive'
    WHEN EVENT_MON_STATE(EVMONNAME) = 1 THEN 'Active'
  END
  FROM SYSCAT.EVENTMONITORS
```

EXP

The EXP function returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument. The EXP and LN functions are inverse operations.

$$\blacktriangleright\blacktriangleright\text{---EXP---(---}expression\text{---)}\text{---}\blacktriangleright\blacktriangleleft$$

The schema is SYSIBM. (The SYSFUN version of the EXP function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes
- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - EXP(NaN) returns NaN.
  - EXP(-NaN) returns -NaN.
  - EXP(Infinity) returns Infinity.
  - EXP(-Infinity) returns 0.
  - EXP(sNaN) returns NaN and a warning.
  - EXP(-sNaN) returns -NaN and a warning.

Example

Assume that E is a DECIMAL(10,9) host variable with a value of 3.453789832.

```
VALUES EXP(:E)
```

Returns the DOUBLE value +3.16200000069145E+001.

EXTRACT

The EXTRACT function returns a portion of a datetime based on its arguments.

Extract date values

```
►►──EXTRACT──(──┬─YEAR──┬──FROM──┬─date-expression───────┬──)────────────────►◄
                ├─MONTH─┤        └─timestamp-expression──┘
                └─DAY───┘
```

Extract time values

```
►►──EXTRACT──(──┬─HOUR───┬──FROM──┬─time-expression───────┬──)────────────────►◄
                ├─MINUTE─┤        └─timestamp-expression──┘
                └─SECOND─┘
```

The schema is SYSIBM.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Extract date values

YEAR
  Specifies that the year portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the YEAR scalar function.

MONTH
  Specifies that the month portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the MONTH scalar function.

DAY
  Specifies that the day portion of *date-expression* or *timestamp-expression* is returned. The result is identical to the DAY scalar function.

date-expression
  An expression that returns the value of either a built-in DATE or built-in character string data type.

  If *date-expression* is a character string, it must be a valid string representation of a date that is not a CLOB. In a Unicode database, if *date-expression* is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
  An expression that returns the value of either a built-in TIMESTAMP or built-in character string data type.

  If *timestamp-expression* is a character string, it must be a valid string representation of a timestamp that is not a CLOB. In a Unicode database, if *timestamp-expression* is a graphic string, it is first converted to a character string before the function is executed.

Extract time values

HOUR
  Specifies that the hour portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the HOUR scalar function.

MINUTE
> Specifies that the minute portion of *time-expression* or *timestamp-expression* is returned. The result is identical to the MINUTE scalar function.

SECOND
> Specifies that the second portion of *time-expression* or *timestamp-expression* is returned. The result is identical to:
> - SECOND(*expression*, 6) when the data type of *expression* is a TIME value or a string representation of a TIME or TIMESTAMP
> - SECOND(*expression*, *s*) when the data type of *expression* is a TIMESTAMP(*s*) value

time-expression
> An expression that returns the value of either a built-in TIME or built-in character string data type.
>
> If *time-expression* is a character string, it must be a valid string representation of a time that is not a CLOB. In a Unicode database, if *time-expression* is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
> An expression that returns the value of a built-in DATE, TIMESTAMP or character string data type.
>
> If *timestamp-expression* is a DATE, it is converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00).
>
> If *timestamp-expression* is a character string, it must be a valid string representation of a timestamp or date that is not a CLOB. In a Unicode database, if *timestamp-expression* is a graphic string, it is first converted to a character string before the function is executed. The string is converted to a TIMESTAMP(6) value.

The data type of the result of the function depends on the part of the datetime value that is specified:
- If YEAR, MONTH, DAY, HOUR, or MINUTE is specified, the data type of the result is INTEGER.
- If SECOND is specified with a TIMESTAMP(*p*) value, the data type of the result is DECIMAL(2+*p*, *p*) where *p* is the fractional seconds precision.
- If SECOND is specified with a TIME value or a string representation of a TIME or TIMESTAMP, the data type of the result is DECIMAL(8,6).

Example

Assume that the column PRSTDATE has an internal value that is equivalent to 1988-12-25:

```
SELECT EXTRACT(MONTH FROM PRSTDATE)
   FROM PROJECT;
```

This statement returns the integer value 12.

FLOAT

The FLOAT function returns a floating-point representation of a number. FLOAT is a synonym for DOUBLE.

```
►►──FLOAT──(──numeric-expression──)──────────────────────────────►◄
```

The schema is SYSIBM.

FLOOR

Returns the largest integer value less than or equal to the argument.

►►──FLOOR──(──*expression*──)──◄

The schema is SYSIBM. (The SYSFUN version of the FLOOR function continues to be available.)

expression
   An expression that returns a value of any built-in numeric data type.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes
- **Results involving DECFLOAT special values:** For decimal floating-point values, the special values are treated as follows:
  - FLOOR(NaN) returns NaN.
  - FLOOR(-NaN) returns -NaN.
  - FLOOR(Infinity) returns Infinity.
  - FLOOR(-Infinity) returns -Infinity.
  - FLOOR(sNaN) returns NaN and a warning.
  - FLOOR(-sNaN) returns -NaN and a warning.

Examples
- *Example 1*: Use the FLOOR function to truncate any digits to the right of the decimal point.
  ```
  SELECT FLOOR(SALARY)
     FROM EMPLOYEE
  ```
- *Example 2*: Use the FLOOR function on both positive and negative numbers.
  ```
  VALUES FLOOR(3.5), FLOOR(3.1),
     FLOOR(-3.1), FLOOR(-3.5)
  ```

This example returns 3., 3., -4., and -4., respectively.

GENERATE_UNIQUE

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function.

►►──GENERATE_UNIQUE─(─)──►◄

The schema is SYSIBM.

The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the database partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.

The function is defined as non-deterministic.

There are no arguments to this function (the empty parentheses must be specified).

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the database partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the database partition number where the function executed so that a table partitioned across multiple database partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement, an insert statement with a fullselect, or an INSERT operation in a MERGE statement.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP scalar function with the result of GENERATE_UNIQUE as an argument.

Example

Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column has "FOR BIT DATA" specified to identify the column as a bit data character string.

```
CREATE TABLE EMP_UPDATE
  (UNIQUE_ID CHAR(13) FOR BIT DATA,
  EMPNO CHAR(6),
  TEXT VARCHAR(1000))
  INSERT INTO EMP_UPDATE
    VALUES (GENERATE_UNIQUE(), '000020', 'Update entry...'),
    (GENERATE_UNIQUE(), '000050', 'Update entry...')
```

This table will have a unique identifier for each row provided that the UNIQUE_ID column is always set using GENERATE_UNIQUE. This can be done by introducing a trigger on the table.

```
CREATE TRIGGER EMP_UPDATE_UNIQUE
  NO CASCADE BEFORE INSERT ON EMP_UPDATE
  REFERENCING NEW AS NEW_UPD
  FOR EACH ROW
  SNEW_UPD.UNIQUE_ID = GENERATE_UNIQUE()
```

With this trigger defined, the previous INSERT statement could be issued without the first column as follows.

```
INSERT INTO EMP_UPDATE (EMPNO, TEXT)
  VALUES ('000020', 'Update entry 1...'),
  ('000050', 'Update entry 2...')
```

The timestamp (in UTC) for when a row was added to EMP_UPDATE can be returned using:

```
SELECT TIMESTAMP (UNIQUE_ID), EMPNO, TEXT
  FROM EMP_UPDATE
```

Therefore, there is no need to have a timestamp column in the table to record when a row is inserted.

GETHINT

The GETHINT function will return the password hint if one is found in the *encrypted-data*.

►►——GETHINT——(——*encrypted-data*——)————————————————————————►◄

The schema is SYSIBM.

A password hint is a phrase that will help data owners remember passwords; for example, 'Ocean' as a hint to remember 'Pacific'. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

encrypted-data
> An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value that is a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function (SQLSTATE 428FE).

The result of the function is VARCHAR(32 OCTETS). The result can be null; if the hint parameter was not added to the *encrypted-data* by the ENCRYPT function or the first argument is null, the result is the null value.

Example

In this example the hint 'Ocean' is stored to help the user remember the encryption password 'Pacific'.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific','Ocean');
SELECT GETHINT(SSN)
  FROM EMP;
```

The value returned is 'Ocean'.

GRAPHIC

The GRAPHIC function returns a fixed-length graphic string representation of a variety of input data types.

Integer to graphic:

►►──GRAPHIC──(──*integer-expression*──)──────────────────────────────►◄

Decimal to graphic:

►►──GRAPHIC──(──*decimal-expression*──┬──────────────────────┬──)──────────►◄
                              └──,──*decimal-character*──┘

Floating-point to graphic:

►►──GRAPHIC──(──*floating-point-expression*──┬──────────────────────┬──)──────►◄
                                     └──,──*decimal-character*──┘

Decimal floating-point to graphic:

►►──GRAPHIC──(──*decimal-floating-point-expression*──┬──────────────────────┬──)──────►◄
                                           └──,──*decimal-character*──┘

Character to graphic:

►►──GRAPHIC──(──*character-expression*──┬──────────────┬──)──────────────────►◄
                                └──,──*integer*──┘

Graphic to graphic:

►►──GRAPHIC──(──*graphic-expression*──┬──────────────┬──)────────────────────►◄
                              └──,──*integer*──┘

Datetime to graphic:

►►──GRAPHIC──(──*datetime-expression*──┬──────────────┬──)────────────────────►◄
                               └──,──┬──ISO──┬──┘
                                     ├──USA──┤
                                     ├──EUR──┤
                                     ├──JIS──┤
                                     └──LOCAL──┘

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:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT

- A decimal number (Unicode database only), if the first argument is a decimal number
- A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
- A decimal floating-point number (Unicode database only), if the argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A datetime value (Unicode database only), if the first argument is a DATE, TIME, or TIMESTAMP

In a non-Unicode database, the string units of the result is double bytes. Otherwise, the string units of the result are determined by the data type of the first argument.
- CODEUNITS16, if the first argument is character string or a graphic string with string units of OCTETS or CODEUNITS16.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

The result of the function is a fixed-length graphic string. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Integer to graphic

> *integer-expression*
>> An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).
>>
>> The result is a fixed-length graphic string representation of *integer-expression* in the form of an SQL integer constant. The result consists of *n* double-byte characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left-aligned.
>> - If the first argument is a small integer, the length of the result is 6.
>> - If the first argument is a large integer, the length of the result is 11.
>> - If the first argument is a big integer, the length of the result is 20.
>>
>> If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.
>>
>> The code page of the result is the DBCS code page of the section.

Decimal to graphic

> *decimal-expression*
>> An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.
>
> *decimal-character*
>> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

Chapter 4. Built-in functions **475**

The result is a fixed-length graphic string representation of *decimal-expression* in the form of an SQL decimal constant. The length of the result is 2+*p*, where *p* is the precision of *decimal-expression*. Leading zeros are not included. Trailing zeros are included. If *decimal-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned. If the number of double-byte characters in the result is less than the defined length of the result, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Floating-point to graphic

floating-point-expression
> An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of *floating-point-expression* in the form of an SQL floating-point constant. The length of the result is 24. The result is the smallest number of double-byte characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by a period and a sequence of digits. If *floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *floating-point-expression* is zero, the result is 0E0. If the number of double-byte characters in the result is less than 24, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Decimal floating-point to graphic

decimal-floating-point-expression
> An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a fixed-length graphic string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The length attribute of the result is 42. The result is the smallest number of double-byte characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings G'INFINITY', G'SNAN', and G'NAN', respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string. If the number of double-byte characters in the result is less than 42, the result is padded on the right with blanks.

The code page of the result is the DBCS code page of the section.

Character to graphic
In Unicode databases:

character-expression
> An expression that returns a value that is a built-in character string data type. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42846).

integer
> An integer constant that specifies the length attribute for the resulting fixed-length graphic string. The value must be between 0 and the maximum length for the GRAPHIC data type in the string units of the result.

> If the second argument is not specified:
> * If the *character-expression* is the empty string constant, the length attribute of the result is 0.
> * Otherwise, the length attribute of the result is the lower of the following values:
>   - The maximum length for the GRAPHIC data type in the string units of the result
>   - The length attribute of the first argument

The result is a fixed-length graphic string that is converted from *character-expression*. The length attribute of the result is determined by the value of *integer*.

The actual length of the result is the same as the length attribute of the result.

If the length of *character-expression* that is converted to a graphic string is less than the length attribute of the result, the result is padded with blanks up to the length attribute of the result.

If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, several scenarios exist:
* If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
* If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior because it might change in a future release.
* If *integer* is not specified, an error is returned (SQLSTATE 22001).

For details about the conversion process, see "VARGRAPHIC" on page 709.

In non-Unicode databases:

character-expression
> An expression that returns a value that is a built-in CHAR or
> VARCHAR data type.

The result is a fixed-length graphic string that is converted from
character-expression. The length attribute of the result is the minimum of the
length attribute of *character-expression* and the maximum length for the
GRAPHIC data type.
The actual length of the result is the same as the length attribute of the
result. If the length of *character-expression* that is converted to a graphic
string is less than the length attribute of the result, the result is padded
with blanks up to the length attribute of the result. If the length of
character-expression that is converted to a graphic string is greater than the
length attribute of the result, an error is returned (SQLSTATE 22001).
For details about the conversion process, see "VARGRAPHIC" on page 709.

Graphic to graphic

graphic-expression
> An expression that returns a built-in value that is a graphic string data
> type.

integer
> An integer constant that specifies the length attribute for the resulting
> fixed-length graphic string. The value must be between 0 and the
> maximum length for the GRAPHIC data type in the string units of the
> result.
>
> If the second argument is not specified:
> - If the *graphic-expression* is the empty string constant, the length
>   attribute of the result is 0.
> - Otherwise, the length attribute of the result is the lower of the
>   following values:
>   – The maximum length for the GRAPHIC data type in the string
>   units of the result
>   – The length attribute of the first argument

The result is a fixed-length graphic string. The length attribute of the result
is determined by the value of *integer*.

The actual length of the result is the same as the length attribute of the
result.

If the length of *graphic-expression* is less than the length attribute of the
result, the result is padded with blanks up to the length attribute of the
result.

If the length of *graphic-expression* is greater than the length attribute of the
result, several scenarios exist:
- If the string unit of the result is CODEUNITS32, truncation is performed.
  If only blank characters are truncated and *graphic-expression* is GRAPHIC
  or VARGRAPHIC, no warning is returned. Otherwise, a warning is
  returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed. If only blank characters
  are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no

warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). In a Unicode database, when the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior because it might change in a future release.

- If *integer* is not specified and *graphic-expression* is VARGRAPHIC, truncation behavior is as follows:
  - If only blank characters must be truncated, truncation is performed with no warning returned.
  - If non-blank characters must be truncated, an error is returned (SQLSTATE 22001).
- If *integer* is not specified and *graphic-expression* is DBCLOB, an error is returned (SQLSTATE 22001).

Datetime to graphic

datetime-expression
An expression that is of one of the following data types:

DATE The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the graphic string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26.

The code page of the string is the code page of the section.

Notes

- **Increasing portability of applications:** The CAST specification should be used to increase the portability of applications when the first argument is numeric, or if the first argument is a string and the length argument is specified.

Examples

- *Example 1*: The EDLEVEL column is defined as SMALLINT. The following returns the value as a fixed-length graphic string.

```
SELECT GRAPHIC(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

Results in the value G'18 '.

- *Example 2*: The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT GRAPHIC(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

GRAPHIC

Results in the value G'56970,00 '.

GREATEST

The GREATEST function returns the maximum value in a set of values.

```
►►─GREATEST─(─expression─┬─,─expression─┬─)───────────────────►◄
```

The schema is SYSIBM.

expression
> The arguments must be compatible and each argument must be an expression that returns a value of any data type other than ARRAY, LOB, XML, a distinct type based on any of these types, or a structured type (SQLSTATE 42815).

An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The GREATEST scalar function is a synonym for the MAX scalar function.
- The GREATEST function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

```
SELECT GREATEST (C1, C2, C3) FROM T1
```

returns 7.

If column C3 has a value of null instead of 4, the same query returns the null value.

HASHEDVALUE

The HASHEDVALUE function returns the distribution map index of the row obtained by applying the partitioning function on the distribution key value of the row.

►►──HASHEDVALUE──(──*column-name*──)──────────────────────────────────►◄

The schema is SYSIBM.

column-name
> The qualified or unqualified name of a column in a table. The column can have any data type.
>
> If *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.

An example application for this function is in a SELECT clause, where it returns the distribution map index for each row of the table that was used to form the result of the SELECT statement.

The distribution map index returned on transition variables and tables is derived from the current transition values of the distribution key columns. For example, in a before insert trigger, the function will return the projected distribution map index given the current values of the new transition variables. However, the values of the distribution key columns may be modified by a subsequent before insert trigger. Thus, the final distribution map index of the row when it is inserted into the database may differ from the projected value.

The specific row (and table) for which the distribution map index is returned by the HASHEDVALUE function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 32767. For a table with no distribution key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

Notes

- The HASHEDVALUE function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).
- The HASHEDVALUE function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.
- The HASHEDVALUE function cannot be used as part of an expression-based key in a CREATE INDEX statement.
- *Syntax alternatives:* For compatibility with previous versions of DB2 products, the function name PARTITION is a synonym for HASHEDVALUE.

Examples

- *Example 1*: List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a distribution map index of 100.

```
SELECT EMPNO FROM EMPLOYEE
  WHERE HASHEDVALUE(PHONENO) = 100
```

- *Example 2*: Log the employee number and the projected distribution map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG2
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AW NEWTABLE
  FOR EACH ROW
  INSERT INTO EMPINSERTLOG2
    VALUES(NEWTABLE.EMPNO, HASHEDVALUE(NEWTABLE.EMPNO))
```

HEX

The HEX function returns a hexadecimal representation of a value as a character string.

►►──HEX──(─*expression*─)──►◄

The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type that is not XML, with a maximum length of 16 336 bytes.

The result of the function is a character string with string units of OCTETS. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The code page is the section code page.

The result is a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value or a numeric value the result is the hexadecimal representation of the internal form of the argument. The hexadecimal representation that is returned may be different depending on the application server where the function is executed. Cases where differences would be evident include:

- Character string arguments when the HEX function is performed on an ASCII client with an EBCDIC server or on an EBCDIC client with an ASCII server.
- Numeric arguments (in some cases) when the HEX function is performed where client and server systems have different byte orderings for numeric values.

The type and length of the result vary based on the type, length, and string units of the character and graphic string arguments.

Table 51. Data type of the result as a function of the data types of the argument data type and the length attribute

Argument data type[1]	Length attribute[2]	Result data type
CHAR(A)	A<128	CHAR(A*2)
CHAR(A)	A>127	VARCHAR(A*2)
VARCHAR(A) or CLOB(A)	A<16337	VARCHAR(A*2)
GRAPHIC(A)	A<64	CHAR(A*2*2)
GRAPHIC(A)	A>63	VARCHAR(A*2*2)
VARGRAPHIC(A) or DBCLOB(A)	A<8169	VARCHAR(A*2*2)
CHAR(A CODEUNITS32)	A<64	VARCHAR(A*4*2)
VARCHAR(A CODEUNITS32) or CLOB(A CODEUNITS32)	A<4085	VARCHAR(A*4*2)
GRAPHIC(A CODEUNITS32)	A<64	VARCHAR(A*2*2*2)

Table 51. Data type of the result as a function of the data types of the argument data type and the length attribute (continued)

Argument data type[1]	Length attribute[2]	Result data type
VARGRAPHIC(A CODEUNITS32) or DBCLOB(A CODEUNITS32)	A<4085	VARCHAR(A*2*2*2)
1. If string units are not specified, then the string units for the data type are not CODEUNITS32. 2. The maximum length attributes reflect a data type limit or the limit of 16336 bytes for the input argument.		

Examples

Assume the use of a database application server on AIX or Linux for the following examples.

- *Example 1*: Using the DEPARTMENT table set the host variable HEX_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the 'PLANNING' department (DEPTNAME).

    ```
    SELECT HEX(MGRNO)
      INTO :HEX_MGRNO
      FROM DEPARTMENT
      WHERE DEPTNAME = 'PLANNING'
    ```

    HEX_MGRNO will be set to '303030303230' when using the sample table (character value is '000020').

- *Example 2*: Suppose COL_1 is a column with a data type of char(1) and a value of 'B'. The hexadecimal representation of the letter 'B' is X'42'. HEX(COL_1) returns a two byte long string '42'.

- *Example 3*: Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight byte long string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

HEXTORAW

The HEXTORAW function returns a bit string representation of a hexadecimal character string.

►►—HEXTORAW—(—*character-expression*—)————————————————————►◄

The schema is SYSIBM.

The HEXTORAW function is a synonym for the VARCHAR_BIT_FORMAT function with a single argument.

HOUR

The HOUR function returns the hour part of a value.

►►──HOUR──(──*expression*──)──►◄

The schema is SYSIBM.

expression
>An expression that returns a value of one of the following built-in data types: a DATE, a TIME, a TIMESTAMP, a character string, or an exact numeric data type.

>If *expression* is a character string, it must not be a CLOB and its value must be a valid string representation of a datetime value. For the valid formats of string representations of datetime values, see "String representations of datetime values" in "Datetime values".

>If *expression* is an exact numeric value, it must be a time duration or timestamp duration. For information about valid time durations and timestamp durations, see "Datetime operands and durations".

>Only Unicode databases support an expression that is a valid graphic string representation of a datetime value that is not a DBCLOB. The graphic string is converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
* If the argument is a TIME, TIMESTAMP or valid string representation of a time or timestamp:
    - The result is the hour part of the value, which is an integer between 0 and 24.
* If the argument is a DATE or valid string representation of a date:
    - The result is 0.
* If the argument is a time duration or timestamp duration:
    - The result is the hour part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Example

Using the CL_SCHED sample table, select all the classes that start in the afternoon.
```
SELECT * FROM CL_SCHED
  WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single INSERT statement using a VALUES clause.

►►──IDENTITY_VAL_LOCAL──(──)──►◄

The schema is SYSIBM.

The function has no input parameters.

The result is a DECIMAL(31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the function is the value assigned to the identity column of the table identified in the most recent single row insert operation. The INSERT statement must contain a VALUES clause on a table containing an identity column. The INSERT statement must also be issued at the same level; that is, the value must be available locally at the level it was assigned, until it is replaced by the next assigned value. (A new level is initiated each time a trigger or routine is invoked.)

The assigned value is either a value supplied by the user (if the identity column is defined as GENERATED BY DEFAULT), or an identity value generated by the database manager.

It is recommended that a SELECT FROM data-change-table-reference statement be used to obtain the assigned value for an identity column. See "table-reference" in "subselect" for more information.

The function returns a null value if a single row INSERT statement with a VALUES clause has not been issued at the current processing level against a table containing an identity column.

The result of the function is not affected by the following operations:
- A single row INSERT statement with a VALUES clause for a table without an identity column
- A multiple row INSERT statement with a VALUES clause
- An INSERT statement with a fullselect
- A ROLLBACK TO SAVEPOINT statement

Notes
- Expressions in the VALUES clause of an INSERT statement are evaluated before the assignments for the target columns of the insert operation. Thus, an invocation of an IDENTITY_VAL_LOCAL function inside the VALUES clause of an INSERT statement will use the most recently assigned value for an identity column from a previous insert operation. The function returns the null value if no previous single row INSERT statement with a VALUES clause for a table containing an identity column has been executed within the same level as the IDENTITY_VAL_LOCAL function.
- The identity column value of the table for which the trigger is defined can be determined within a trigger by referencing the trigger transition variable for the identity column.

- The result of invoking the IDENTITY_VAL_LOCAL function from within the trigger condition of an insert trigger is a null value.
- It is possible that multiple before or after insert triggers exist for a table. In this case, each trigger is processed separately, and identity values assigned by one triggered action are not available to other triggered actions using the IDENTITY_VAL_LOCAL function. This is true even though the multiple triggered actions are conceptually defined at the same level.
- It is not generally recommended to use the IDENTITY_VAL_LOCAL function in the body of a before insert trigger. The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of a before insert trigger is the null value. The value for the identity column of the table for which the trigger is defined cannot be obtained by invoking the IDENTITY_VAL_LOCAL function within the triggered action of a before insert trigger. However, the value for the identity column can be obtained in the triggered action by referencing the trigger transition variable for the identity column.
- The result of invoking the IDENTITY_VAL_LOCAL function from within the triggered action of an after insert trigger is the value assigned to an identity column of the table identified in the most recent single row insert operation invoked in the same triggered action that had a VALUES clause for a table containing an identity column. (This applies to both FOR EACH ROW and FOR EACH STATEMENT after insert triggers.) If a single row INSERT statement with a VALUES clause for a table containing an identity column was not executed within the same triggered action, before the invocation of the IDENTITY_VAL_LOCAL function, the function returns a null value.
- Because IDENTITY_VAL_LOCAL is a non-deterministic function, the result of invoking this function within the SELECT statement of a cursor can vary for each FETCH statement.
- The assigned value is the value actually assigned to the identity column (that is, the value that would be returned on a subsequent SELECT statement). This value is not necessarily the value provided in the VALUES clause of the INSERT statement, or a value generated by the database manager. The assigned value could be a value specified in a SET transition variable statement, within the body of a before insert trigger, for a trigger transition variable associated with the identity column.
- *Scope of IDENTITY_VAL_LOCAL*: The IDENTITY_VAL_LOCAL value persists until the next insert in the current session into a table that has an identity column defined on it, or the application session ends. The value is unaffected by COMMIT or ROLLBACK statements. The IDENTITY_VAL_LOCAL value cannot be directly set and is a result of inserting a row into a table.

  A technique commonly used, especially for performance, is for an application or product to manage a set of connections and route transactions to an arbitrary connection. In these situations, the availability of the IDENTITY_VAL_LOCAL value should be relied on only until the end of the transaction. Examples of where this type of situation can occur include applications that use XA protocols, use connection pooling, use the connection concentrator, and use HADR to achieve failover.
- The value returned by the function following a failed single row INSERT statement with a VALUES clause into a table with an identity column is unpredictable. It could be the value that would have been returned from the function had it been invoked before the failed insert operation, or it could be the value that would have been assigned had the insert operation succeeded. The actual value returned depends on the point of failure, and is therefore unpredictable.

Examples

- *Example 1:* Create two tables, T1 and T2, each with an identity column named C1. Start the identity sequence for table T2 at 10. Insert some values for C2 into T1.

```
CREATE TABLE T1
  (C1 INTEGER GENERATED ALWAYS AS IDENTITY,
   C2 INTEGER)

CREATE TABLE T2
  (C1 DECIMAL(15,0) GENERATED BY DEFAULT AS IDENTITY (START WITH 10),
   C2 INTEGER)

INSERT INTO T1 (C2) VALUES (5)

INSERT INTO T1 (C2) VALUES (6)

SELECT * FROM T1
```

This query returns:

```
C1          C2
----------- -----------
          1           5
          2           6
```

Insert a single row into table T2, where column C2 gets its value from the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL())

SELECT * FROM T2
```

This query returns:

```
C1                C2
----------------- -----------
              10.           2
```

- *Example 2:* In a nested environment involving a trigger, use the IDENTITY_VAL_LOCAL function to retrieve the identity value assigned at a particular level, even though there might have been identity values assigned at lower levels. Assume that there are three tables, EMPLOYEE, EMP_ACT, and ACCT_LOG. There is an after insert trigger defined on EMPLOYEE that results in additional inserts into the EMP_ACT and ACCT_LOG tables.

```
CREATE TABLE EMPLOYEE
  (EMPNO SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1000),
   NAME CHAR(30),
   SALARY DECIMAL(5,2),
   DEPTNO SMALLINT)

CREATE TABLE EMP_ACT
  (ACNT_NUM SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 1),
   EMPNO SMALLINT)

CREATE TABLE ACCT_LOG
  (ID SMALLINT GENERATED ALWAYS AS IDENTITY (START WITH 100),
   ACNT_NUM SMALLINT,
   EMPNO SMALLINT)

CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_EMP
  FOR EACH ROW
  BEGIN ATOMIC
```

```
       INSERT INTO EMP_ACT (EMPNO) VALUES (NEW_EMP.EMPNO);
       INSERT INTO ACCT_LOG (ACNT_NUM, EMPNO)
         VALUES (IDENTITY_VAL_LOCAL(), NEW_EMP.EMPNO);
     END
```

The first triggered insert operation inserts a row into the EMP_ACT table. The statement uses a trigger transition variable for the EMPNO column of the EMPLOYEE table to indicate that the identity value for the EMPNO column of the EMPLOYEE table is to be copied to the EMPNO column of the EMP_ACT table. The IDENTITY_VAL_LOCAL function could not be used to obtain the value assigned to the EMPNO column of the EMPLOYEE table, because an INSERT statement has not been issued at this level of the nesting. If the IDENTITY_VAL_LOCAL function were invoked in the VALUES clause of the INSERT statement for the EMP_ACT table, it would return a null value. The insert operation against the EMP_ACT table also results in the generation of a new identity value for the ACNT_NUM column.

The second triggered insert operation inserts a row into the ACCT_LOG table. The statement invokes the IDENTITY_VAL_LOCAL function to indicate that the identity value assigned to the ACNT_NUM column of the EMP_ACT table in the previous insert operation in the triggered action is to be copied to the ACNT_NUM column of the ACCT_LOG table. The EMPNO column is assigned the same value as the EMPNO column of the EMPLOYEE table.

After the following INSERT statement and all of the triggered actions have been processed:

```
    INSERT INTO EMPLOYEE (NAME, SALARY, DEPTNO)
      VALUES ('Rupert', 989.99, 50)
```

the contents of the three tables are as follows:

```
    SELECT EMPNO, SUBSTR(NAME,1,10) AS NAME, SALARY, DEPTNO
      FROM EMPLOYEE

EMPNO  NAME        SALARY  DEPTNO
------ ---------- ------- ------
  1000 Rupert      989.99     50

    SELECT ACNT_NUM, EMPNO
      FROM EMP_ACT

ACNT_NUM EMPNO
-------- ------
       1   1000

    SELECT * FROM ACCT_LOG

ID     ACNT_NUM EMPNO
------ -------- ------
   100        1  1000
```

The result of the IDENTITY_VAL_LOCAL function is the most recently assigned value for an identity column at the same nesting level. After processing the original INSERT statement and all of the triggered actions, the IDENTITY_VAL_LOCAL function returns a value of 1000, because this is the value that was assigned to the EMPNO column of the EMPLOYEE table.

INITCAP

The INITCAP function returns a string with the first character of each *word* converted to uppercase, using the UPPER function semantics, and the other characters converted to lowercase, using the LOWER function semantics.

►►──INITCAP──(──*string-expression*──)────────────────────────────────►◄

The schema is SYSIBM.

A *word* is delimited by any of the following characters:

Table 52. Word delimiter characters

Character or range of characters	Unicode code points or range of Unicode code points
(blank)	U+0020
! " # $ % & ' ( ) * + , -. /	U+0021 to U+002F
: ; < = > ? @	U+003A to U+0040
[\ ] ^ _ `	U+005B to U+0060
{	} ~	U+007B to U+007E
Control characters, including the following SQL control characters: • tab • new line • form feed • carriage return • line feed	U+0009, U+000A, U+000B, U+000C, U+000D, U+0085

Note: A character listed in the preceding table might not have an allocated code point in a particular database code page.

string-expression
> An expression that returns a CHAR or VARCHAR data type. In a Unicode database, the expression can return a GRAPHIC or VARGRAPHIC data type.

The data type of the result depends on the data type of *string-expression*, as described in the following table:

Table 53. Data type of string-expression compared to the data type of the result

Data type of *string-expression*	Data type of the result
CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	VARGRAPHIC

The length attribute of the result is the same as the length attribute of *string-expression*.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

• *Example 1*: Input the string "a prospective book title" to return the string "A Prospective Book Title".

```
VALUES INITCAP ('a prospective book title')
1
-----------------------
A Prospective Book Title
```

- *Example 2*: Input the string "YOUR NAME" to return the string "Your Name".

```
VALUES INITCAP ('YOUR NAME')
1
---------
Your Name
```

- *Example 3*: Input the string "my_résumé" to return the string "My_Résumé".

```
VALUES INITCAP ('my_résumé')
1
---------
My_Résumé
```

- *Example 4*: Input the string "élégant" to return the string "Élégant".

```
VALUES INITCAP ('FORMAT:élégant')
1
--------------
Format:Élégant
```
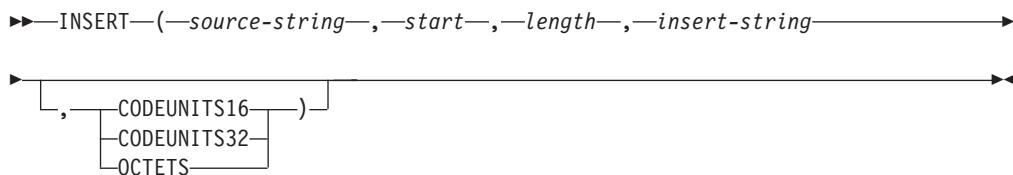
INSERT

The INSERT function returns a string where, beginning at *start* in *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.

```
►►──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 is identical to the OVERLAY function, except that the length argument is mandatory.

source-string
> An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

start
> An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion of code units and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value is the starting point in code units using the specified string units. The integer value must be between 1 and the actual length of *source-string* in implicit or explicit string units plus one (SQLSTATE 22001). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and the actual octet length of *source-string* plus one (SQLSTATE 428GC or 22011).

length
> An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be positive integer or zero (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number or zero (SQLSTATE 428GC).

insert-string
> An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

CODEUNITS16, CODEUNITS32, or OCTETS
> Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or bit data, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, and *insert-string* and *source-string* are binary strings, an error is returned (SQLSTATE 42815). If the string unit is specified as CODEUNITS16 or OCTETS, and the string units of source-string is CODEUNITS32, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*. If a string unit is not explicitly specified, the string unit of the *source-string* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following tables of supported type combinations. The string unit of the result is the string unit of *source-string*. If either *source-string* or *insert-string* is defined as FOR BIT DATA the other argument cannot be defined with string units of CODEUNITS32.The second table applies only to Unicode databases.

Table 54. Data type of the result as a function of the data types of source-string and insert-string

source-string	insert-string	Result
CHAR or VARCHAR	CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	GRAPHIC or VARGRAPHIC	VARGRAPHIC
CLOB	CHAR, VARCHAR, or CLOB	CLOB
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
BLOB	BLOB	BLOB

Table 55. Data type of the result as a function of the data types of source-string and insert-string (Unicode databases only)

source-string	insert-string	Result
CHAR or VARCHAR	GRAPHIC or VARGRAPHIC	VARCHAR
GRAPHIC or VARGRAPHIC	CHAR or VARCHAR	VARGRAPHIC
CLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	CLOB
DBCLOB	CHAR, VARCHAR, or CLOB	DBCLOB

A *source-string* can have a length of 0; in this case, *start* must be 1 (as implied by the bounds for *start* described previously), and the result of the function is a copy of the *insert-string*.

An *insert-string* can also have a length of 0. This has the effect of deleting the code units identified by *start* and *length* from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string* when the string units of the *source-string* and *insert-string* are the same or the result string units is CODEUNITS32. Special cases are listed in the following table.

Table 56. Data type of the result as a function of the data types of source-string and insert-string (special cases)

source-string		insert-string		Result	
Data type	String units	Data type	String units	Length attribute	String units
Character string with length attribute A	OCTETS	Graphic string with length attribute B	CODEUNITS16	A+3*B	OCTETS
			CODEUNITS32	A+4*B	
		Character with length attribute B	CODEUNITS32	A+4*B	
Graphic string with length attribute A	CODEUNITS16	Character with length attribute B	OCTETS	A+B	CODEUNITS16
			CODEUNITS32	A+2*B	
		Graphic string with length attribute B	CODEUNITS32	A+2*B	

The actual length of the result depends on the actual length of *source-string*, the actual length of the of the deleted string, the actual length of the *insert-string*, and string units used for the *start* and *length* arguments. For example, if the string arguments are character strings in OCTETS and the OCTETS is used as the fourth argument, the actual length of the result is A1 - MIN((A1 - V2 + 1), V3) + A4, where:
- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.
  ```
  SELECT INSERT('INSERTING',4,2,'IS'),
    INSERT('INSERTING',4,0,'IS'),
    INSERT('INSERTING',4,2,'')
    FROM SYSIBM.SYSDUMMY1
  ```
- *Example 2*: Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.

```
    SELECT INSERT('INSERTING',1,0,'XX'),
       INSERT('INSERTING',1,1,'XX'),
       INSERT('INSERTING',1,2,'XX'),
       INSERT('INSERTING',1,3,'XX')
       FROM SYSIBM.SYSDUMMY1
```

- *Example 3*: Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
    SELECT INSERT('ABCABC',7,0,'XX')
       FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: Change the string 'Hegelstraße' to 'Hegelstrasse'.

```
    SELECT INSERT('Hegelstraße',10,1,'ss',CODEUNITS16)
       FROM SYSIBM.SYSDUMMY1
```

- *Example 5*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the INSERT function to insert a 'C' into the Unicode string '&N~AB'.

```
    SELECT INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS16),
       INSERT(UTF8_VAR, 1, 4, 'C', CODEUNITS32),
       INSERT(UTF8_VAR, 1, 4, 'C', OCTETS)
    FROM SYSIBM.SYSDUMMY1
```

returns the values 'CAB', 'CB', and 'CN~AB', respectively.

```
    SELECT INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS16),
       INSERT(UTF8_VAR, 5, 1, 'C', CODEUNITS32),
       INSERT(UTF8_VAR, 5, 1, 'C', OCTETS)
    FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB', '&N~AC', and '&C~AB', respectively.

```
    SELECT INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS16),
       INSERT(UTF16_VAR, 1, 4, 'C', CODEUNITS32),
       INSERT(UTF16_VAR, 1, 4, 'C', OCTETS)
    FROM SYSIBM.SYSDUMMY1
```

returns the values 'CAB', 'CB', and 'CN~AB', respectively.

```
    SELECT INSERT(UTF16_VAR, 5, 2, 'C', CODEUNITS16),
       INSERT(UTF16_VAR, 5, 1, 'C', CODEUNITS32),
       INSERT(UTF16_VAR, 5, 4, 'C', OCTETS)
    FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~C', '&N~AC', and '&CAB', respectively.

INSTR

The INSTR function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*).

```
►►—INSTR—(—source-string—,—search-string─┬──────────────────────┬─┬──────────────────┬—)—————►◄
                                          └─,—start─┬──────────────┘ └─,─┬─CODEUNITS16─┬─┘
                                                    └─,—instance─┘       ├─CODEUNITS32─┤
                                                                         └─OCTETS──────┘
```

The schema is SYSIBM

The INSTR scalar function is a synonym for the LOCATE_IN_STRING scalar function.

INSTR2

The INSTR2 function returns the starting position, in 16-bit UTF-16 string units (CODEUNITS16), of a string within another string.

```
►►──INSTR2──(──source-string──,──search-string──────────────────────)────►◄
                                          └─,──start──┬─────────────────┘
                                                      └─,──instance──┘
```

The schema is SYSIBM.

source-string
   An expression that specifies the string in which the search is to take place.

search-string
   An expression that specifies the string that is the object of the search.

start
   An expression that specifies the position within *source-string* at which the search for a match is to start.

instance
   An expression that specifies which instance of *search-string* to search for within *source-string*.

The INSTR2 scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with CODEUNITS16 specified. The INSTR2 scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INSTR4

The INSTR4 function returns the starting position, in 32-bit UTF-32 string units (CODEUNITS32), of a string within another string.

```
►►──INSTR4──(──source-string──,──search-string──────────────────────────)────►◄
                                              └─,──start──────────┘
                                                     └─,──instance─┘
```

The schema is SYSIBM.

source-string
>   An expression that specifies the string in which the search is to take place.

search-string
>   An expression that specifies the string that is the object of the search.

start
>   An expression that specifies the position within *source-string* at which the search for a match is to start.

instance
>   An expression that specifies which instance of *search-string* to search for within *source-string*.

The INSTR4 scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with CODEUNITS32 specified. The INSTR4 scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INSTRB

The INSTRB function returns the starting position, in bytes, of a string within another string.

```
►►─INSTRB─(─source-string─,─search-string─────────────────────────)─────►◄
                                           └─,─start─┬───────────────┘
                                                     └─,─instance─┘
```

The schema is SYSIBM.

source-string
    An expression that specifies the string in which the search is to take place.

search-string
    An expression that specifies the string that is the object of the search.

start
    An expression that specifies the position within *source-string* at which the search for a match is to start.

instance
    An expression that specifies which instance of *search-string* to search for within *source-string*.

The INSTRB scalar function invoked with character or graphic strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function with OCTETS specified. The INSTRB scalar function invoked with binary strings as the first two arguments is equivalent to invoking the LOCATE_IN_STRING function without a string units argument.

INTEGER or INT

The INTEGER function returns an integer representation of a number, a string representation of a number, a date value, or a time value.

Numeric to Integer:

```
►►─┬─INTEGER─┬─(─numeric-expression─)────────────────────────►◄
   └─INT─────┘
```

String to Integer:

```
►►─┬─INTEGER─┬─(─string-expression─)─────────────────────────►◄
   └─INT─────┘
```

Date to Integer:

```
►►─┬─INTEGER─┬─(─date-expression─)────────────────────────────►◄
   └─INT─────┘
```

Time to Integer:

```
►►─┬─INTEGER─┬─(─time-expression─)────────────────────────────►◄
   └─INT─────┘
```

The schema is SYSIBM.

Numeric to Integer:

numeric-expression
An expression that returns a value of any built-in numeric data type.

The result is the same number that would occur if the argument were assigned to a large integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of integers, an error is returned (SQLSTATE 22003).

String to Integer:

string-expression
An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a of length not greater than the maximum length of a character constant.

The result is the same number that would result from CAST(*string-expresssion* AS INTEGER). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of integers, an error is returned (SQLSTATE 22003). The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884).

Date to Integer:

date-expression
>   An expression that returns a value of the DATE data type. The result is an INTEGER value representing the date as *yyyymmdd*.

Time to Integer:

time-expression
>   An expression that returns a value of the TIME data type. The result is an INTEGER value representing the time as *hhmmss*.

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.

Note: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification".

Examples

- *Example 1*: Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value.

  ```
  SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
    FROM EMPLOYEE
    ORDER BY 1 DESC
  ```

- *Example 2*: Using the EMPLOYEE table, select the EMPNO column in integer form for further processing in the application.

  ```
  SELECT INTEGER(EMPNO) FROM EMPLOYEE
  ```

- *Example 3*: Assume that the column BIRTHDATE (whose data type is DATE) has an internal value equivalent to '1964-07-20'.

  ```
  INTEGER(BIRTHDATE)
  ```

  results in the value 19 640 720.

- *Example 4*: Assume that the column STARTTIME (whose data type is TIME) has an internal value equivalent to '12:03:04'.

  ```
  INTEGER(STARTTIME)
  ```

  results in the value 120 304.

JULIAN_DAY

Returns an integer value representing the number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date value specified in the argument.

►►─JULIAN_DAY─(─*expression*─)──────────────────────────────────►◄

The schema is SYSFUN.

expression
    An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

LAST_DAY

The LAST_DAY scalar function returns a date or timestamp value that represents the last day of the month of the argument.

►►—LAST_DAY—(—*expression*—)—————————————————————————————►◄

The schema is SYSIBM.

expression
>   An expression that specifies the starting date. The expression must return a value of one of the following built-in data types: a DATE or a TIMESTAMP.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is DATE. The result can be null; if the value of *date-expression* is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function.

Examples

- *Example 1*: Set the host variable *END_OF_MONTH* with the last day of the current month.

      SET :END_OF_MONTH = **LAST_DAY**(**CURRENT_DATE**);

  The host variable *END_OF_MONTH* is set with the value representing the end of the current month. If the current day is 2000-02-10, then *END_OF_MONTH* is set to 2000-02-29.

- *Example 2*: Set the host variable *END_OF_MONTH* with the last day of the month in EUR format for the given date.

      SET :END_OF_MONTH = **CHAR**(**LAST_DAY**('1965-07-07'), **EUR**);

  The host variable *END_OF_MONTH* is set with the value '31.07.1965'.

LCASE

The LCASE function returns a string in which all the SBCS characters have been converted to lowercase characters.

►►──LCASE──(──*string-expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

LCASE is a synonym for LOWER.

LCASE (locale sensitive)

The LCASE function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

```
>>─LCASE─(─string-expression─,─locale-name─┬──────────────┬─┬──────────────────┬─)──><
                                           └─,─code-units─┘ └─,─┬─CODEUNITS16─┬─┘
                                                               ├─CODEUNITS32─┤
                                                               └─OCTETS──────┘
```

The schema is SYSIBM.

LCASE is a synonym for LOWER.

LCASE (SYSFUN schema)

The LCASE or LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters. That is, the characters A-Z will be converted to the characters a-z (characters with diacritical marks are not converted). Note that LCASE(UCASE(string)) will therefore not necessarily return the same result as LCASE(string).

$$\blacktriangleright\!\blacktriangleright\text{---LCASE---(---}expression\text{---)}\text{---------------------------------------}\blacktriangleright\!\blacktriangleleft$$

The schema is SYSFUN.

expression
>   The *expression* can be of any built-in character string type. For a VARCHAR, the maximum length is 4 000 bytes, and for a CLOB, the maximum length is 1 048 576 bytes.

The result of the function is:
- VARCHAR(4000) if *expression* is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if *expression* is CLOB or LONG VARCHAR
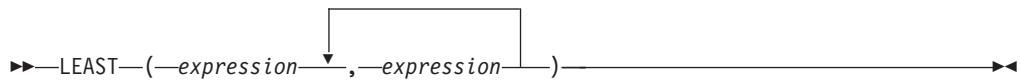
The result can be null; if *expression* is null, the result is the null value.

In a Unicode database, if *expression* is a graphic string, it is first converted to a character string before the function is executed.

LEAST

The LEAST function returns the minimum value in a set of values.

►►—LEAST—(—*expression*—,—*expression*—)—————————————►◄

The schema is SYSIBM.

expression
> An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the smallest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes
- The LEAST scalar function is a synonym for the MIN scalar function.
- The LEAST function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4, respectively. The query:

```
SELECT LEAST (C1, C2, C3) FROM T1
```
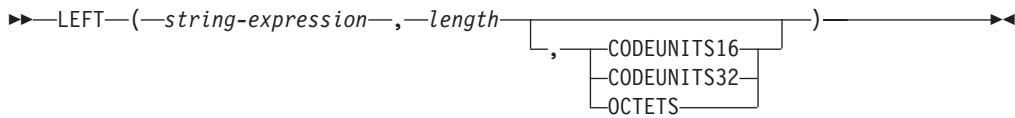
returns 1.

If column C3 has a value of null instead of 4, the same query returns the null value.

LEFT

The LEFT function returns the leftmost string of *string-expression* of length *length*, expressed in the specified string unit.

```
►►──LEFT──(──string-expression──,──length──┬─────────────────────┬──)──────────►◄
                                           │  ┌──CODEUNITS16──┐    │
                                           └─,─┼──CODEUNITS32──┤────┘
                                              └──OCTETS───────┘
```

The schema is SYSIBM. The SYSFUN version of the LEFT function continues to be available.

If *string-expression* is a character string, the result is a character string. If *string-expression* is a graphic string, the result is a graphic string.

string-expression
> An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string-expression* is zero or more contiguous code points of *string-expression*.

length
> An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be between 0 and the length of *string-expression*, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011) with one exception. If the value is specified as a constant without explicitly specifying the string unit, the value can exceed the length attribute of *string-expression* in the implicit string unit. 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 string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information about padding, see "String assignments" in "Assignments and comparisons".

The result of the function is a varying-length string with a length attribute that depends on how *length* and the string unit are specified. If *length* is not specified using a constant or the string unit is explicitly specified, then the length attribute is the same as the length attribute of *string-expression*. If *length* is specified using a constant and the string unit is not specified, then the length attribute is the maximum of *length* and the length attribute of *string-expression* up to the maximum length for the result data type in the string units of *string-expression*. The data type of the result 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
* BLOB if *string-expression* is BLOB

The string unit of the result is the string unit of *string-expression*. The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

* *Example 1*: Assume that variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT LEFT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
```

returns 'ABC', which are the three leftmost characters in ALPHA.

* *Example 2*: Assume that variable NAME, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN', and that the integer variable FIRSTNAME_LEN has a value of 5. The following statement:

```
SELECT LEFT(NAME,FIRSTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
```

returns the value 'KATIE'.

* *Example 3*: The following statement returns a zero-length string.

```
SELECT LEFT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
```

* *Example 4*: The FIRSTNME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT LEFT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
```

returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

* *Example 5*: In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function...                        Returns...

LEFT(FIRSTNAME,2,CODEUNITS32)    'Jü' -- x'4AC3BC'
LEFT(FIRSTNAME,2,CODEUNITS16)    'Jü' -- x'4AC3BC'
LEFT(FIRSTNAME,2,OCTETS)         'J'  -- x'4A20', a truncated string
```

- *Example 6*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT LEFT(UTF8_VAR, 2, CODEUNITS16),
  LEFT(UTF8_VAR, 2, CODEUNITS32),
  LEFT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&', '&N', and 'bb', respectively, where 'b' represents the blank character.

```
SELECT LEFT(UTF8_VAR, 5, CODEUNITS16),
  LEFT(UTF8_VAR, 5, CODEUNITS32),
  LEFT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~A', '&N~AB', and '&N', respectively.

```
SELECT LEFT(UTF8_VAR, 10, CODEUNITS16),
  LEFT(UTF8_VAR, 10, CODEUNITS32),
  LEFT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbbb', '&N~ABbbbbb', and '&N~ABb', respectively, where 'b' represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT LEFT(UTF16_VAR, 2, CODEUNITS16),
  LEFT(UTF16_VAR, 2, CODEUNITS32),
  HEX (LEFT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values '&', '&N', and X'D834', respectively, where X'D834' is an unmatched high surrogate.

```
SELECT LEFT(UTF16_VAR, 5, CODEUNITS16),
  LEFT(UTF16_VAR, 5, CODEUNITS32),
  LEFT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```
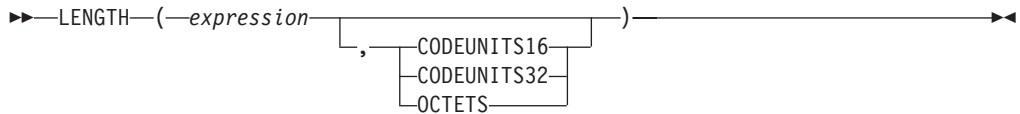
returns the values '&N~A', '&N~AB', and '&N', respectively.

```
SELECT LEFT(UTF16_VAR, 10, CODEUNITS16),
  LEFT(UTF16_VAR, 10, CODEUNITS32),
  LEFT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbbb', '&N~ABbbbbb', and '&N~A', respectively, where 'b' represents the blank character.

LENGTH

The LENGTH function returns the length of *expression* in the implicit or explicit string unit.

►►─LENGTH─(─*expression*─┬───────────────────────┬─)────────────────────────────►◄
                        └─,─┬─CODEUNITS16─┬──────┘
                            ├─CODEUNITS32─┤
                            └─OCTETS──────┘

The schema is SYSIBM.

expression
> An expression that returns a value that is a built-in data type. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

CODEUNITS16, CODEUNITS32, or OCTETS
> Specifies the string unit of the result. CODEUNITS16 specifies that the result is to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that the result is to be expressed in 32-bit UTF-32 code units. OCTETS specifies that the result is to be expressed in bytes.

> If a string unit is explicitly specified, and if *expression* is not string data, an error is returned (SQLSTATE 428F5). If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *expression* is a binary string or bit data, an error is returned (SQLSTATE 428GC). If a string unit is specified as OCTETS and *expression* is a binary string, an error is returned (SQLSTATE 42815). For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

If a string unit argument is not explicitly specified and if *expression* is a character or graphic string, the string units of *expression* determines the string unit that is used for the result. Otherwise, the value returned specifies the length in bytes.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character and graphic strings includes trailing blanks. The length of binary strings includes binary zeros. The length of varying-length strings is the actual length and not the maximum length. The length of all other values is the number of bytes used to represent the value:
- 2 for small integer (SMALLINT)
- 4 for large integer (INTEGER)
- 8 for big integer (BIGINT)
- $(p/2)+1$ for decimal numbers with precision p
- 8 for DECFLOAT(16)
- 16 for DECFLOAT(34)
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for DATE
- 3 for TIME
- $7+(p+1)/2$ for TIMESTAMP(p)

Examples

- *Example 1*: Assume that the host variable ADDRESS is a varying-length character string with a value of '895 Don Mills Road'.

  ```
  LENGTH(:ADDRESS)
  ```

  returns the value 18.
- *Example 2*: Assume that START_DATE is a column of type DATE.

  ```
  LENGTH(START_DATE)
  ```

  returns the value 4.
- *Example 3*: The following example returns the value 10.

  ```
  LENGTH(CHAR(START_DATE, EUR))
  ```

- *Example 4*: The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'
UTF-32BE	X'0001D11E'	X'0000004E'	X'00000303'	X'00000041'	X'00000042'

Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT LENGTH(UTF8_VAR, CODEUNITS16),
  LENGTH(UTF8_VAR, CODEUNITS32),
  LENGTH(UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 9, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT LENGTH(UTF16_VAR, CODEUNITS16),
  LENGTH(UTF16_VAR, CODEUNITS32),
  LENGTH(UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 12, respectively.

LENGTH2

The LENGTH2 function returns the length of *expression* in 16-bit UTF-16 string units (CODEUNITS16).

►►—LENGTH2—(—*expression*—)————————————————————————►◄

The schema is SYSIBM.

expression
    An expression that returns a value that is a built-in data type.

The LENGTH2 scalar function invoked with a character or graphic string as the argument is equivalent to invoking the LENGTH function with CODEUNITS16 specified. The LENGTH2 scalar function invoked with any other data type as the argument is equivalent to invoking the LENGTH function without a string units argument.

LENGTH4

The LENGTH4 function returns the length of *expression* in 32-bit UTF-32 string units (CODEUNITS32).

►►──LENGTH4──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSIBM.

expression
   An expression that returns a value that is a built-in data type.

The LENGTH4 scalar function invoked with a character or graphic string as the argument is equivalent to invoking the LENGTH function with CODEUNITS32 specified. The LENGTH4 scalar function invoked with any other data type as the argument is equivalent to invoking the LENGTH function without a string units argument.

LENGTHB

The LENGTHB function returns the length of *expression* in bytes.

►►——LENGTHB——(——*expression*——)————————————————————————►◄

The schema is SYSIBM.

expression
>   An expression that returns a value that is a built-in data type.

The LENGTHB scalar function invoked with a character or graphic string as the
argument is equivalent to invoking the LENGTH function with OCTETS specified.
The LENGTHB scalar function invoked with any other data type as the argument
is equivalent to invoking the LENGTH function without a string units argument.

LN

The LN function returns the natural logarithm of a number. The LN and EXP functions are inverse operations.

►►──LN──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the LN function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Notes
- *Results involving DECFLOAT special values:* For decimal floating-point values, the special values are treated as follows:
  - LN(NaN) returns NaN.
  - LN(-NaN) returns -NaN.
  - LN(Infinity) returns Infinity.
  - LN(-Infinity) returns NaN and a warning.
  - LN(sNaN) returns NaN and a warning.
  - LN(-sNaN) returns -NaN and a warning.
  - LN(DECFLOAT('0')) returns -Infinity.
- *Syntax alternatives:* LOG can be specified in place of LN. It is supported only for compatibility with previous versions of DB2 products. 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
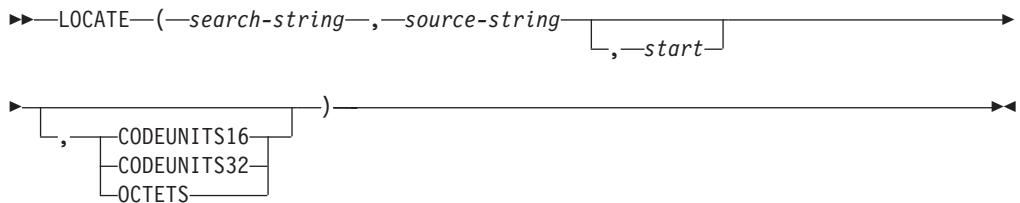
Assume that NATLOG is a DECIMAL(4,2) host variable with a value of 31.62.
```
   VALUES LN(:NATLOG)
```

Returns the approximate value 3.45.

LOCATE

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*).

```
►►──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.

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 CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be a BLOB file reference variable. The expression can be specified by any of the following elements:
> - A constant
> - A special register
> - A global variable
> - A host variable
> - A scalar function whose operands are any of the previously listed items
> - An expression that concatenates (using CONCAT or ||) any of the previously listed items
> - 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 string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified by any of the following elements:
> - 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 previously listed items

start
> An expression that specifies the position within *source-string* at which the search is to start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be greater than or equal to zero. If *start* is specified, the LOCATE function is similar to:
>
> ```
> POSITION(search-string,
>   SUBSTRING(source-string, start, string-unit),
>   string-unit) + start - 1
> ```
>
> where *string-unit* is either CODEUNITS16, CODEUNITS32, or OCTETS.
>
> If *start* is not specified, the search begins at the first position of the source string, and the LOCATE function is similar to:
>
> ```
> POSITION(search-string, source-string, string-unit)
> ```

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 the string unit is specified as CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC).
>
> If a string unit is not explicitly specified and if *source-string* is a character or graphic string, the string units of *source-string* determines the unit that is used for the result and for *start* (if specified). Otherwise, they are expressed in bytes.
>
> If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.
>
> For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Find the location of the first occurrence of the character 'N' in the string 'DINING'.

  ```
  SELECT LOCATE('N', 'DINING')
  FROM SYSIBM.SYSDUMMY1
  ```

  The result is the value 3.

- *Example 2*: For all the rows in the table named IN_TRAY, select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD' within the NOTE_TEXT column.

  ```
  SELECT RECEIVED, SUBJECT, LOCATE('GOOD', NOTE_TEXT)
  FROM IN_TRAY
  WHERE LOCATE('GOOD', NOTE_TEXT) <> 0
  ```

- *Example 3*: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

  ```
  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS32)
  ```

  The value of host variable LOCATION is set to 26.

- *Example 4*: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS16 units, within the string.

  ```
  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, CODEUNITS16)
  ```

  The value of host variable LOCATION is set to 26.

- *Example 5*: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

  ```
  SET :LOCATION = LOCATE('ß', 'Jürgen lives on Hegelstraße', 1, OCTETS)
  ```

  The value of host variable LOCATION is set to 27.

- *Example 6*: The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the non-spacing combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

  Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT LOCATE('~', UTF8_VAR, CODEUNITS16),
    LOCATE('~', UTF8_VAR, CODEUNITS32),
    LOCATE('~', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 4, 3, and 6, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT LOCATE('~', UTF16_VAR, CODEUNITS16),
    LOCATE('~', UTF16_VAR, CODEUNITS32),
    LOCATE('~', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 4, 3, and 7, respectively.
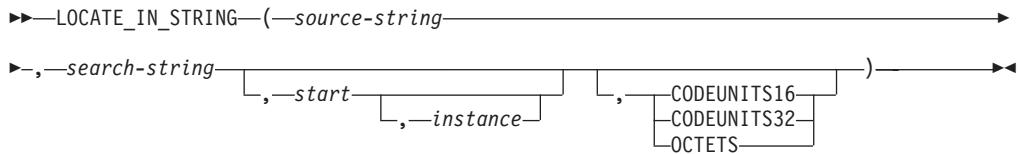
- *Example 7*: In a Unicode database created with the case insensitive collation CLDR181_LEN_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = LOCATE('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.

LOCATE_IN_STRING

The LOCATE_IN_STRING function returns the starting position of a string (called the *search-string*) within another string (called the *source-string*).

►►—LOCATE_IN_STRING—(—*source-string*————————————————————►

►—,—*search-string*————————————————————————————)—————►◄
            └─,—*start*————┘   └─,——CODEUNITS16—┐
                  └─,—*instance*—┘   ├──CODEUNITS32─┤
                                    └──OCTETS────┘

The schema is SYSIBM

If the *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of the *source-string*. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as FOR BIT DATA, in which case the search is done using a binary comparison.

If the optional *start* is specified, it indicates the character position in the *source-string* at which the search is to begin. If the *start* is specified, an instance number can also be specified. The *instance* argument is used to determine the position of a specific occurrence of *search-string* within *source-string*. An optional string unit can be specified to indicate in what units the *start* and result of the function are expressed.

If the *search-string* has a length of zero, the result returned by the function is 1. If the *source-string* has a length of zero, the result returned by the function is 0. If neither condition exists, and if the value of *search-string* is equal to an identical length of a substring of contiguous positions within the value of *source-string*, the result returned by the function is the starting position of that substring within the *source-string* value; otherwise, the result returned by the function is 0.

source-string
    An expression that specifies the string in which the search is to take place. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified in any of the following ways:
- A constant
- A special register
- A global variable
- A host variable (including a LOB 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 previous items

search-string
    An expression that specifies the string that is the object of the search. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The actual length

Chapter 4. Built-in functions   **523**

must not be greater than the maximum length of a VARCHAR. The *search-string* cannot be a BLOB file reference variable. The expression can be specified in any of the following ways:

- A constant
- A special register
- A global variable
- A host variable
- A scalar function whose arguments are any of the previous items
- An expression that concatenates (using CONCAT or | |) any of the previous items

These rules are similar to those that are described for a *pattern-expression* for the LIKE predicate.

start
    An expression that specifies the position within *source-string* at which the search for a match is to start. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

    If the value of the integer is greater than zero, the search begins at *start* and continues for each position to the end of the string. If the value of the integer is less than zero, the search begins at LENGTH(*source-string*) + *start* + 1 and continues for each position to the beginning of the string.

    If *start* is not specified, the default is 1. If the value of the integer is zero, an error is returned (SQLSTATE 42815).

instance
    An expression that specifies which instance of *search-string* to search for within *source-string*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. If *instance* is not specified, the default is 1. The value of the integer must be greater than or equal to 1 (SQLSTATE 42815).

CODEUNITS16, CODEUNITS32, or OCTETS
    Specifies the string unit of *start* and the result. CODEUNITS16 specifies that *start* and the result are to be expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and the result are to be expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and the result are to be expressed in bytes.

    If a string unit is specified as CODEUNITS16 or CODEUNITS32, and *search-string* or *source-string* is a binary string or 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 the string unit is specified as CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC).

    If a string unit is not explicitly specified and if *source-string* is a character or graphic string, the string units of *source-string* determines the unit that is used for the result and for *start* (if specified). Otherwise, they are expressed in bytes.

    If a locale-sensitive UCA-based collation is used for this function, then the CODEUNITS16 option offers the best performance characteristics.

For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The first and second arguments must have compatible string types. For more information about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

At each search position, a match is found when the substring at that position and LENGTH(*search-string*) - 1 values to the right of the search position in *source-string*, is equal to *search-string*.

The result of the function is a large integer. The result is the starting position of the instance of *search-string* within *source-string*. The value is relative to the beginning of the string (regardless of the specification of *start*). If any argument can be null, the result can be null; if any argument is null, the result is the null value.

INSTR can be used as a synonym for LOCATE_IN_STRING.

The INSTRB scalar function is equivalent to invoking the LOCATE_IN_STRING function with OCTETS (where allowed) specified to indicate that start position and the result are expressed in bytes.

Examples

- *Example 1*: Locate the character 'ß' in the string 'Jürgen lives on Hegelstraße' by searching from the end of the string, and set the host variable POSITION with the position, as measured in CODEUNITS32 units, within the string.

```
SET :POSITION = LOCATE_IN_STRING('Jürgen lives on Hegelstraße',
                                 'ß',-1,CODEUNITS32);
```

The value of host variable POSITION is set to 26.

- *Example 2*: Find the location of the third occurrence of the character 'N' in the string 'WINNING' by searching from the start of the string and then set the host variable POSITION with the position of the character, as measured in bytes, within the string.

```
SET :POSITION =
LOCATE_IN_STRING('WINNING','N',1,3,OCTETS);
```

The value of host variable POSITION is set to 6.

LOG10

The LOG10 function returns the common logarithm (base 10) of a number.

►►──LOG10──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the LOG10 function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type. If the value is of decimal floating-point data type, the operation is performed in decimal floating-point; otherwise, the value is converted to double-precision floating-point for processing by the function. The value of the argument must be greater than zero (SQLSTATE 22003).

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Notes
- *Results involving DECFLOAT special values:* For decimal floating-point values, the special values are treated as follows:
  - LOG10(NaN) returns NaN.
  - LOG10(-NaN) returns -NaN.
  - LOG10(Infinity) returns Infinity.
  - LOG10(-Infinity) returns NaN and a warning.
  - LOG10(sNaN) returns NaN and a warning.
  - LOG10(-sNaN) returns -NaN and a warning.
  - LOG10(DECFLOAT('0')) returns -Infinity.

Example

Assume that L is a DECIMAL(4,2) host variable with a value of 31.62.
```
VALUES LOG10(:L)
```

Returns the DOUBLE value +1.49996186559619E+000.

LONG_VARCHAR

The LONG_VARCHAR function is deprecated and might be removed in a future release.

►►──LONG_VARCHAR──(──*character-string-expression*──)────────────────────►◄

The function is compatible with earlier DB2 versions.

LONG_VARGRAPHIC

The LONG_VARGRAPHIC function is deprecated and might be removed in a future release.

►►──LONG_VARGRAPHIC──(──*graphic-expression*──)──────────────────────────►◄

The function is compatible with earlier DB2 versions.

LOWER

The LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters.

►►—LOWER—(—*string-expression*—)—————————————————►◄

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for CLOB arguments.)

string-expression
    An expression that returns a value of built-in CHAR or VARCHAR 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.

With the LOWER function, 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 result of the function has the same data type, string unit, and length attribute as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example

Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.
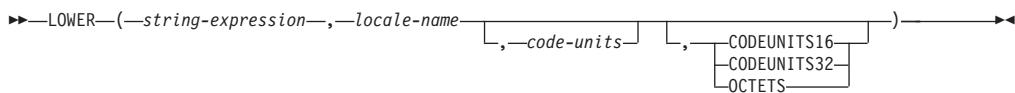
```
SELECT LOWER(JOB)
  FROM EMPLOYEE
  WHERE EMPNO = '000020';
```

The result is the value 'manager'.

LOWER (locale sensitive)

The LOWER function returns a string in which all characters have been converted to lowercase characters using the rules associated with the specified locale.

►►—LOWER—(—*string-expression*—,—*locale-name*————————————————————————————————————
          └─,—*code-units*─┘ └─,—┬─CODEUNITS16─┬─┘)—————►◄
                                 ├─CODEUNITS32─┤
                                 └─OCTETS──────┘

The schema is SYSIBM.

string-expression
> An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If *string-expression* is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

locale-name
> A character constant that specifies the locale that defines the rules for conversion to lowercase characters. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery".

code-units
> An integer constant that specifies the number of code units in the result. If specified, *code-units* must be an integer between:
> - 1 and 32 672 if the string unit of the result is OCTETS
> - 1 and 16 336 if the string unit of the result is double bytes or CODEUNITS16
> - 1 and 8 168 if the string unit of the result is CODEUNITS32
>
> otherwise an error is returned (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). If OCTETS is specified and the string units of the result is CODEUNIT32, the value of *code-units* must be a multiple of 4 (SQLSTATE 428GC). If CODEUNITS16 is specified and the string units of the result is CODEUNITS32, the value of *code-units* must be a multiple of 2 (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
> Specifies the string unit of *code-units*.
>
> CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.
>
> If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC. The string units of the result is the same as the string units of *string-expression*.

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, the result data type, and the result string units, as shown in the following table:

Table 57. Length attribute of the result of LOWER as a function of string unit and result type

Data type and string units of result	Length attribute for *code-units* in CODEUNITS16	Length attribute for *code-units* in CODEUNITS32	Length attribute for *code-units* in OCTETS
VARCHAR in OCTETS	MIN(*code-units* * 3, 32672)	MIN(*code-units* * 4, 32672)	*code-units*
VARCHAR in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)
VARGRAPHIC in CODEUNITS16 or double bytes	*code-units*	MIN(*code-units* * 2, 16336)	MIN(*code-units* / 2, 16336)
VARGRAPHIC in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of *string-expression*. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.
```
SELECT LOWER(JOB, 'en_US')
  FROM EMPLOYEE
  WHERE EMPNO = '000020'
```

The result is the value 'manager'.

- *Example 2*: Find the lowercase characters for all the 'I' characters in a Turkish string.
```
VALUES LOWER('I⌂ıi', 'tr_TR', CODEUNITS16)
```

The result is the string 'ııii'.

LPAD

The LPAD function returns a string composed of *string-expression* that is padded on the left, with *pad* or blanks.

```
►►──LPAD──(──string-expression──,──integer──────────────)────────────────►◄
                                          └─, pad─┘
```

The schema is SYSIBM.

The LPAD function treats leading or trailing blanks in *string-expression* as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression
> An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

integer
> An integer expression that specifies the length of the result in the string units of *string-expression*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to the maximum length for the result data type in the string units of *string-expression*.

pad
> An expression that specifies the string with which to pad. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.
>
> If *pad* is not specified, the pad character is determined as follows:
> - SBCS blank character if *string-expression* is a character string.
> - Ideographic blank character, if *string-expression* is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.

The result of the function is a varying length string that has the same string unit and code page as *string-expression*. The value for *string-expression* and the value for *pad* must have compatible data types. If the *string-expression* and *pad* have different code pages, then *pad* is converted to the code page of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on whether the value for *integer* is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if *integer* is greater than zero, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the

result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:

Table 58. Determining the result length when integer is available only when the function is executed

Data type of *string-expression*	Result data type length
CHAR(*n*) or VARCHAR(*n*)	Minimum of *n*+100 and 32 672
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of *n*+100 and 16 336
CHAR(n) or VARCHAR(*n*) or GRAPHIC(*n*) or VARGRAPHIC(*n*) with string units of CODEUNITS32 (Unicode database only)	Minimum of *n*+100 and 8 168

The actual length of the result is determined from *integer*. If *integer* is 0 the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the left with periods:

  SELECT LPAD(NAME,15,'.') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  ---------------
  ..........Chris
  ............Meg
  ...........Jeff
  ```

- *Example 2*: Assume that NAME is a VARCHAR(15) column that contains the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

  SELECT LPAD(NAME,5,'.') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  -----
  Chris
  ..Meg
  .Jeff
  ```

- *Example 3*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". The LPAD function does not pad because NAME is a fixed length character field and is blank padded already. However, since the length of the result is 5, the columns are truncated:

  SELECT LPAD(NAME,5,'.') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  -----
  Chris
  Meg
  Jeff
  ```

- *Example 4*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". In some cases, a partial instance of the pad specification is returned:

```
SELECT LPAD(NAME,15,'123' ) AS NAME FROM T1;
```

returns:
```
NAME
---------------
1231231231Chris
123123123123Meg
12312312312Jeff
```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

```
SELECT LPAD(NAME,4,'.' ) AS NAME FROM T1;
```

returns:
```
NAME
----
Chri
.Meg
Jeff
```

LTRIM

The LTRIM function removes any of the specified characters from the beginning of a string.

The LTRIM function removes any of the characters contained in a *trim-expression* from the beginning of a *string-expression*. The character search compares the binary representation of each character (consisting of one or more bytes) in the *trim-expression* to the binary representation of each character (consisting of one or more bytes) at the beginning of the *string-expression*. The database collation does not affect the search. If the *string-expression* is defined as FOR BIT DATA, the search compares each byte in the *trim-expression* to the byte at the beginning of the *string-expression*.

►►──LTRIM──(──*string-expression*──┬──────────────────────┬──)──────────────────►◄
                                    └─,──*trim-expression*──┘

The schema is SYSIBM. (The SYSFUN version of this function that uses a single parameter continues to be available with support for CLOB arguments.)

string-expression

An expression that specifies the source string.

- If only one argument is specified, the expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the data type of the *string-expression* value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, the value is implicitly cast to VARCHAR before evaluating the function.
- If both arguments are specified, the expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the data type of the *string-expression* value is not a character string or graphic string data type, the value is implicitly cast to VARCHAR before evaluating the function. The actual length of a CLOB value is limited to the maximum size of a VARCHAR data type (SQLSTATE 22001). The actual length of a DBCLOB value is limited to the maximum size of a VARGRAPHIC data type (SQLSTATE 22001).

trim-expression

An expression that specifies the characters that are being removed from the beginning of a *string-expression*. The expression must be a value that is a built-in character string, graphic string, numeric, or datetime data type.

- If the data type of the *trim-expression* is not a character string or graphic string, then the value is implicitly cast to VARCHAR before evaluating the function.
- If the data type of the *trim-expression* is a CLOB, then the actual length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001).
- If the data type of the *trim-expression* is a DBCLOB, then the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001).
- If the *string-expression* is not defined as FOR BIT DATA, then the *trim-expression* cannot be defined as FOR BIT DATA (SQLSTATE 42815).

When a *trim-expression* is not specified, the data type of the *string-expression* determines the default value used:

LTRIM

- A double byte blank if the *string-expression* is a graphic string in a DBCS or EUC database
- A UCS-2 blank if the *string-expression* is a graphic string in a Unicode database
- A value of X'20' if the *string-expression* is a FOR BIT DATA string
- A single-byte blank for all other cases

If one function argument is character FOR BIT DATA, then the other argument cannot be a graphic (SQLSTATE 42846). A combination of character string and graphic string arguments can be used only in a Unicode database (SQLSTATE 42815).

The data type of the result depends on the data type of the *string-expression*.
- VARCHAR if the data type is VARCHAR or CHAR
- CLOB if the data type is CLOB
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- DBCLOB if the data type is DBCLOB

The length attribute of the result data type is the same as the length attribute of the *string-expression* data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of string units removed. The actual length of the result for graphic strings is the length of *string-expression* minus the number of string units removed. If all of the characters are removed, the result is an empty string with a length of zero.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples
- *Example 1*: Use the LTRIM function when the host variable HELLO is defined as CHAR(6) and has a value of " Hello".

      VALUES LTRIM(:HELLO**)**

  The result is 'Hello'. When a *trim-expression* is not specified only blanks are removed.
- *Example 2*: Use the LTRIM function to remove the characters specified in the *trim-expression* from the beginning of the *string-expression*.

      VALUES LTRIM('...$V..$AR'**, **'$.'**)**

  The result is 'V..$AR'. The function stops when it encounters a character not in the *trim-expression*.
- *Example 3*: Use the LTRIM function to remove the characters specified in the *trim-expression* from the beginning of the *string-expression*

      VALUES LTRIM('[[-78]]'**, **'- []'**)**

  The result is '78]]'. When removing characters and blanks, you must include a blank in the *trim-expression*.

LTRIM (SYSFUN schema)

Returns the characters of the argument with leading blanks removed.

►►—LTRIM—(—*expression*—)————————————————————————◄◄

The schema is SYSFUN.

expression
> The *expression* can be of any built-in character string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes.
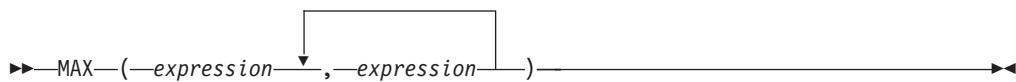
The result of the function is:
- VARCHAR(4000) if *expression* is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if *expression* is CLOB or LONG VARCHAR

The result can be null; if *expression* is null, the result is the null value.

MAX

The MAX function returns the maximum value in a set of values.

```
►►──MAX──(──expression──┬──,──expression──┬──)──────────────────►◄
                        └────────◄─────────┘
```

The schema is SYSIBM.

expression

An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the largest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The MAX scalar function is a synonym for the GREATEST scalar function.
- The MAX function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Return the bonus for an employee, the greater of 500 and 5% of the employee's salary.

```
SELECT EMPNO, MAX(SALARY * 0.05, 500)
  FROM EMPLOYEE
```

MAX_CARDINALITY

The MAX_CARDINALITY function returns a value of type BIGINT representing the maximum number of elements that an array can contain. This is the cardinality that was specified in the CREATE TYPE statement for the ordinary array type.

►►──MAX_CARDINALITY──(──*array-variable*──)────────────────────────────►◄

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an array type, or a CAST specification of a parameter marker to an array type.

The result can be null; if the argument is an associative array, the result is the null value.

Example

Return the maximum cardinality of the RECENT_CALLS array variable of array type PHONENUMBERS:

```
SET LIST_SIZE = MAX_CARDINALITY(RECENT_CALLS)
```

The SQL variable LIST_SIZE is set to 50, which is the maximum cardinality that the array type PHONENUMBERS was defined with.

MICROSECOND

The MICROSECOND function returns the microsecond part of a value.

►►──MICROSECOND──(──*expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a DATE, TIMESTAMP, or a valid string representation of a date or timestamp:
  - The integer ranges from 0 through 999 999.
  - If the precision of the timestamp exceeds 6, the value is truncated.
- If the argument is a duration:
  - The result reflects the microsecond part of the value which is an integer between -999 999 through 999 999. A nonzero result has the same sign as the argument.

Example

Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT * FROM TABLEA
  WHERE MICROSECOND(TS1) <> 0
    AND
  SECOND(TS1) = SECOND(TS2)
```

MIDNIGHT_SECONDS

Returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the argument.

►►─── MIDNIGHT_SECONDS──(──*expression*──)──────────────────────────────────►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If the expression is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Examples

- *Example 1*: Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

    ```
    VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))
    ```

    This example returns the following:

    ```
    1           2
    ----------- -----------
            610       47410
    ```

    Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47 410 seconds between the specified time and midnight.

- *Example 2*: Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

    ```
    VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))
    ```

    This example returns the following:
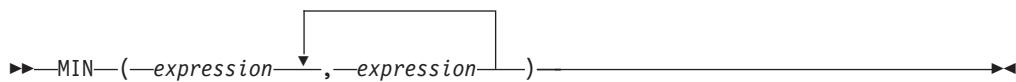
    ```
    1           2
    ----------- -----------
          86400           0
    ```

    Note that these two values represent the same point in time, but return different MIDNIGHT_SECONDS values.

MIN

The MIN function returns the minimum value in a set of values.

```
►►──MIN──(──expression──┬──,──expression──┬──)───────────────────►◄
```

The schema is SYSIBM.

expression
An expression that returns a value of any built-in data type or user-defined data type that is comparable with data type of the other arguments. The data type cannot be a LOB, distinct type base on a LOB, XML, array, cursor, row, or structured type.

The result of the function is the smallest argument value. The result can be null if at least one argument can be null; the result is the null value if any argument is null.

The selected argument is converted, if necessary, to the attributes of the result. The attributes of the result are determined by the data types of all the arguments as explained in "Rules for result data types".

Notes

- The MIN scalar function is a synonym for the LEAST scalar function.
- The MIN function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example

Return the bonus for an employee, the LESSER of 5000 and 5% of the employee's salary.

```
SELECT EMPNO, MIN(SALARY * 0.05, 5000)
  FROM EMPLOYEE
```

MINUTE

The MINUTE function returns the minute part of a value.

►►──MINUTE──(──*expression*──)────────────────────────────►◄

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If a supplied argument is a DATE, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time or timestamp:
  - The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the minute part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

Using the CL_SCHED sample table, select all classes with a duration less than 50 minutes.

```
SELECT * FROM CL_SCHED
  WHERE HOUR(ENDING - STARTING) = 0
    AND
  MINUTE(ENDING - STARTING) < 50
```

MOD

Returns the remainder of the first argument divided by the second argument.

►►──MOD──(──*expression1*──,──*expression2*──)────────────────────────────◄

The schema is SYSIBM. (The SYSFUN version of this function continues to be available.)

expression1
> An expression that returns a value of any built-in numeric data type.

expression2
> An expression that returns a value of any built-in numeric data type. This expression can only be zero when at least one of the function arguments is a decimal floating point.

The formula for calculating the remainder is:
```
MOD(x,y) = x-(x/y)*y
```

where x/y is the truncated integer result of the division.

The result only is negative when the first argument is negative.

The result can be null if either argument can be null or if neither argument is a decimal floating-point number and the dft_sqlmathwarn database configuration parameter is set to YES; the result is the null value when either argument is NULL.

The data type of the result depends on the data types of the arguments.
- Small integer if both arguments are small integers.
- Large integer if one argument is a large integer and the other argument is either a small integer or a large integer.
- Big integer if both arguments are integers and at least one argument is a big integer.
- Decimal if one argument is an integer and the other argument is a decimal. The result has the same precision as the decimal argument.
- Decimal if both arguments are decimals. The precision of the result is MIN (p-s,p'-s')+(MAX(s,s') and the scale is (MAX(s,s'). Where p and s represent the precision and scale of the first argument and p' and s' represent the precision and scale of the second argument.
- Double-precision floating point if one argument is a floating-point number and the other is not a DECFLOAT. The arguments are converted to double-precision floating point numbers before performing the MOD function. For example, if one argument is a floating-point number and the other is an integer or decimal, the function is performed with a temporary copy of the integer or decimal, which has been converted to double-precision floating-point. The result must be in the range of floating-point numbers.
- Double-precision floating point if both arguments are floating-point numbers. The result must be in the range of floating-point numbers.
- DECFLOAT(34) if either argument is a decimal floating-point. If *expression2* evaluates to zero, the result is not a number (NaN) and an invalid operation warning with the associated SQLSTATE is issued.

  If either argument is a special decimal floating-point value, the general arithmetic operation rules for decimal floating-point apply (see Expressions).

- Results when arguments include infinity:
  - `MOD(x, +/-INFINITY)` returns the value x.
  - `MOD(+/-INFINITY, +/-INFINITY)` returns NaN and an invalid operation warning with the associated SQLSTATE.
  - `MOD(+/-INFINITY, x)` returns NaN and an invalid operation warning with the associated SQLSTATE.

Examples

- *Example 1*: Assume the host variable M1 is an INTEGER with a value of 5 and host variable M2 is an INTEGER with a value of 2

  ```
  SELECT MOD(:M1, :M2)
     FROM SYSIBM.SYSDUMMY1
  ```

  The result is 1 with a data type of INTEGER.
- *Example 2*: Assume the host variable M1 is an INTEGER with a value of 5 and host variable M2 is a DECIMAL(3,2) with a value of 2.20

  ```
  SELECT MOD(:M1, :M2)
     FROM SYSIBM.SYSDUMMY1
  ```

  The result is 0.60 with a data type of DECIMAL(3,2).
- *Example 3*: Assume the host variable M1 is a DECIMAL(4,2) with a value of 5.5 and host variable M2 is a DECIMAL(4,1) with a value of 2.0

  ```
  SELECT MOD(:M1, :M2)
     FROM SYSIBM.SYSDUMMY1
  ```

  The result is 1.50 with a data type of DECIMAL(4,2).

MOD (SYSFUN schema)

Returns the remainder of the first argument divided by the second argument.

►►──MOD──(──*expression*──,──*expression*──)──────────────────────────────►◄

The schema is SYSFUN.

The result is negative only if first argument is negative. The result of the function is:
- SMALLINT if both arguments are SMALLINT
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT.

The result can be null; if any argument is null, the result is the null value.

MONTH

The MONTH function returns the month part of a value.

►►—MONTH—(—*expression*—)———————————————————————————◄

The schema is SYSIBM.

expression
   An expression that returns a value of one of the following built-in data types:
   DATE, TIMESTAMP, date duration, timestamp duration, or a valid character
   string representation of a date or timestamp that is not a CLOB. In a Unicode
   database, if a supplied argument is a graphic string (except DBCLOB), it is first
   converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result
can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a DATE, TIMESTAMP, or a valid string representation of a
  date or timestamp:
  – The result is the month part of the value, which is an integer between 1 and
    12.
- If the argument is a date duration or timestamp duration:
  – The result is the month part of the value, which is an integer between -99 and
    99. A nonzero result has the same sign as the argument.

Example

Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE)
in DECEMBER.

```
SELECT * FROM EMPLOYEE
  WHERE MONTH(BIRTHDATE) = 12
```

MONTHNAME

The MONTHNAME function returns a character string containing the name of the month (for example, January) for the month portion of the input value.

►►──MONTHNAME──(──*expression*──┬─────────────────────┬──)──────────────────────►◄
                                └─,──*locale-name*────┘

The schema is SYSIBM. The SYSFUN version of the **MONTHNAME** function continues to be available.

The character string returned is based on *locale-name* or the value of the special register CURRENT LOCALE LC_TIME.

expression
> An expression that returns a value of one of the following built-in data types: a DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

locale-name
> A character constant that specifies the locale used to determine the language of the result. The value of *locale-name* is not case-sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result is a varying-length character string. The length attribute is 100. If the resulting string exceeds the length attribute of the result, the result will be truncated. If the *expression* argument can be null, the result can be null; if the *expression* argument is null, the result is the null value. The code page of the result is the code page of the section. The string units of the result is determined by the string units of the environment.

Notes

- **Julian and Gregorian calendar**: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function. However, the SYSFUN version of the MONTHNAME function assumes the Gregorian calendar for all calculations.
- **Determinism**: MONTHNAME is a deterministic function. However, when *locale-name* is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. This invocation that depends on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Assume that the variable TMSTAMP is defined as TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result type in each case is VARCHAR(100).

```
Function invocation                        Result
-------------------------                  ----------
MONTHNAME (TMSTAMP, 'CLDR181_en_US')        March
MONTHNAME (TSMTAMP, 'CLDR181_de_DE')        Marz
MONTHNAME (TMSTAMP, 'CLDR181_fr_FR')        mars
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns an estimate of the number of months between *expression1* and *expression2*.

▶▶──MONTHS_BETWEEN──(──*expression1*──,──*expression2*──)────────────────────◀◀

The schema is SYSIBM.

expression1 **or** *expression2*
> Expressions that return a value of either a DATE or TIMESTAMP data type.

If *expression1* represents a date that is later than *expression2*, the result is positive. If *expression1* represents a date that is earlier than expression2, the result is negative.

- If *expression1* and *expression2* represent dates or timestamps with the same day of the month, or both arguments represent the last day of their respective months, the result is a the whole number difference based on the year and month values ignoring any time portions of timestamp arguments.

- Otherwise, the whole number part of the result is the difference based on the year and month values. The fractional part of the result is calculated from the remainder based on an assumption that every month has 31 days. If either argument represents a timestamp, the arguments are effectively processed as timestamps with maximum precision, and the time portions of these values are also considered when determining the result.

The result of the function is a DECIMAL(31,15). If either argument can be null, the result can be null. If either argument is null, the result is the null value.

Examples

- *Example 1*: Calculate the number of months that project AD3100 will take. Assume that the start date is 1982-01-01 and the end date is 1983-02-01.

      SELECT MONTHS_BETWEEN (PRENDATE, PRSDATE)
         FROM PROJECT
         WHERE PROJNO='AD3100'

  The result is 13.000000000000000.

- *Example 2*: Here are some additional examples to consider:

Table 59. Additional examples using MONTHS_BETWEEN

Value for argument *e1*	Value for argument *e2*	Value returned by MONTHS_BETWEEN (*e1*,*e2*)	Value returned by ROUND (MONTHS_BETWEEN (*e1*,*e2*)*31,2)	Comment
2005-02-02	2005-01-01	1.032258064516129	32.00	
2007-11-01-09.00.00.00000	2007-12-07-14.30.12.12345	-1.200945386592741	-37.23	
2007-12-13-09.40.30.00000	2007-11-13-08.40.30.00000	1.000000000000000	31.00	See Note 1
2007-03-15	2007-02-20	0.838709677419354	26.00	See Note 2
2008-02-29	2008-02-28-12.00.00	0.016129032258064	0.50	
2008-03-29	2008-02-29	1.000000000000000	31.00	
2008-03-30	2008-02-29	1.032258064516129	32.00	
2008-03-31	2008-02-29	1.000000000000000	31.00	See Note 3

Note:

1. The time difference is ignored because the day of the month is the same for both values.
2. The result is not 23 because, even though February has 28 days, the assumption is that all months have 31 days.
3. The result is not 33 because both dates are the last day of their respective month, and so the result is only based on the year and month portions.

MULTIPLY_ALT

The MULTIPLY_ALT scalar function returns the product of the two arguments.

►►──MULTIPLY_ALT──(──*numeric_expression1*──,──*numeric_expression2*──)──────────►◄

The schema is SYSIBM.

numeric_expression1
    An expression that returns a value of any built-in numeric data type.

numeric_expression2
    An expression that returns a value of any built-in numeric data type.

The MULTIPLY_ALT function is provided as an alternative to the multiplication operator, especially when the sum of the decimal precisions of the arguments exceeds 31.

The result of the function is DECIMAL when both arguments are exact numeric data types (DECIMAL, BIGINT, INTEGER, or SMALLINT); otherwise the operation is carried out using decimal floating-point arithmetic and the result of the function is decimal floating-point with a precision determined by the data type of the arguments in the same way the precision is determined for decimal floating-point arithmetic. A floating-point or string argument is cast to DECFLOAT(34) before evaluating the function.

When the result of the function is DECIMAL, the precision and scale of the result are determined as follows, using the symbols p and s to denote the precision and scale of the first argument, and the symbols p' and s' to denote the precision and scale of the second argument.
- The precision is $MIN(31, p + p')$
- The scale is:
    - 0 if the scale of both arguments is 0
    - $MIN(31, s + s')$ if $p + p'$ is less than or equal to 31
    - $MAX(MIN(3, s + s'), 31 - (p - s + p' - s'))$ if $p + p'$ is greater than 31.

The result can be null if at least one argument can be null, or if the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if one of the arguments is null.

The MULTIPLY_ALT function is a preferable choice to the multiplication operator when performing decimal arithmetic where a scale of at least 3 is required and the sum of the precisions exceeds 31. In these cases, the internal computation is performed so that overflows are avoided. The final result is then assigned to the result data type, using truncation where necessary to match the scale. Note that overflow of the final result is still possible when the scale is 3.

The following table is a sample comparing the result types using MULTIPLY_ALT and the multiplication operator.

Type of argument 1	Type of argument 2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)

Type of argument 1	Type of argument 2	Result using MULTIPLY_ALT	Result using multiplication operator
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

Example

Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

```
values multiply_alt(98765432109876543210987.654,5.43210987)
1
---------------------------------
536504678578875294857887.5277415
```

Note that the complete product of these two numbers is 536504678578875294857887.52774154498, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.

NCHAR

The NCHAR function returns a fixed-length national character string representation of a variety of data types.

Integer to nchar

►►──NCHAR──(──*integer-expression*──)────────────────────────►◄

Decimal to nchar

►►──NCHAR──(──*decimal-expression*──┬──────────────────────┬──)────────────►◄
                                  └──,──*decimal-character*──┘

Floating-point to nchar

►►──NCHAR──(──*floating-point-expression*──┬──────────────────────┬──)───────►◄
                                         └──,──*decimal-character*──┘

Decimal floating-point to nchar

►►──NCHAR──(──*decimal-floating-point-expression*──┬──────────────────────┬──)───►◄
                                                 └──,──*decimal-character*──┘

Character to nchar

►►──NCHAR──(──*character-expression*──┬──────────────┬──)────────────────────►◄
                                      └──,──*integer*──┘

Graphic to nchar

►►──NCHAR──(──*graphic-expression*──┬──────────────┬──)──────────────────────►◄
                                └──,──*integer*──┘

Nchar to nchar

►►──NCHAR──(──*national-character-expression*──┬──────────────┬──)───────────►◄
                                          └──,──*integer*──┘

Datetime to nchar

►►──NCHAR──(──*datetime-expression*──┬──────────────┬──)─────────────────────►◄
                                  └──,──┬──ISO───┬──┘
                                           ├──USA───┤
                                           ├──EUR───┤
                                           ├──JIS───┤
                                           └──LOCAL─┘

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The NCHAR function can be specified only in a Unicode database (SQLSTATE 560AA).

The NCHAR function returns a fixed-length national character string representation of:
- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A national character string, if the first argument is any type of national character string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP

The NCHAR scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

Table 60. NCHAR scalar function synonyms

NCHAR_MAPPING value	Synonym function	Result string units
CHAR_CU32	CHAR	CODEUNITS32
GRAPHIC_CU32	GRAPHIC	CODEUNITS32
GRAPHIC_CU16	GRAPHIC[1]	CODEUNITS16
1. When the first argument has string units of CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the GRAPHIC function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is determined by the GRAPHIC function.		

NCLOB

The NCLOB function returns a NCLOB representation of any type of national character string.

```
►►──NCLOB──(──national-character-expression──────────────)─────────────►◄
                                          └─,──integer─┘
```

The schema is SYSIBM.

The NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

The NCLOB scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

Table 61. NCLOB scalar function synonyms

NCLOB_MAPPING value	Synonym function	Result string units
CHAR_CU32	CLOB	CODEUNITS32
GRAPHIC_CU32	DCLOB	CODEUNITS32
GRAPHIC_CU16	DCLOB[1]	CODEUNITS16
1. When the first argument has string units CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the DCLOB function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is otherwise determined by the DCLOB function.		

NVARCHAR

The NVARCHAR function returns a varying-length national character string representation of a variety of data types.

Integer to nvarchar

►►──NVARCHAR──(──*integer-expression*──)──────────────────────────►◄

Decimal to nvarchar

►►──NVARCHAR──(──*decimal-expression*──┬─────────────────────┬──)──────────►◄
                                        └─,──*decimal-character*─┘

Floating-point to nvarchar

►►──NVARCHAR──(──*floating-point-expression*──┬─────────────────────┬──)──────►◄
                                              └─,──*decimal-character*─┘

Decimal floating-point to nvarchar

►►──NVARCHAR──(──*decimal-floating-point-expression*──┬─────────────────────┬──)──────►◄
                                                      └─,──*decimal-character*─┘

Character to nvarchar

►►──NVARCHAR──(──*character-expression*──┬───────────┬──)──────────────►◄
                                         └─,──*integer*─┘

Graphic to nvarchar

►►──NVARCHAR──(──*graphic-expression*──┬───────────┬──)────────────────►◄
                                       └─,──*integer*─┘

Nchar to nvarchar

►►──NVARCHAR──(──*national-character-expression*──┬───────────┬──)──────────►◄
                                                  └─,──*integer*─┘

Datetime to nvarchar

►►──NVARCHAR──(──*datetime-expression*──┬─────────────┬──)──────────────►◄
                                        └─,──┬─ISO───┬─┘
                                             ├─USA───┤
                                             ├─EUR───┤
                                             ├─JIS───┤
                                             └─LOCAL─┘

The schema is SYSIBM.

The function name cannot be specified as a qualified name when keywords are used in the function signature.

NVARCHAR can be specified only in a Unicode database (SQLSTATE 560AA).

The NVARCHAR function returns a varying-length national character string representation of:

- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- An national character string, if the first argument is any type of national character string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP

The NVARCHAR scalar function is a synonym for a scalar cast function with result string units as specified in the following table.

Table 62. NVARCHAR scalar function synonyms

NVARCHAR_MAPPING value	Synonym function	Result string units
CHAR_CU32	VARCHAR	CODEUNITS32
GRAPHIC_CU32	VARGRAPHIC	CODEUNITS32
GRAPHIC_CU16	VARGRAPHIC[1]	CODEUNITS16
1. When the first argument has string units CODEUNITS32 and the second argument is not specified, the length attribute of the result is different from the VARGRAPHIC function because the string units of the result is CODEUNITS16. In this case, the length attribute of the result is 2 times the length attribute of the result that is otherwise determined by the VARGRAPHIC function.		

NEXT_DAY

The NEXT_DAY scalar function returns a datetime value that represents the first weekday, named by *string-expression*, that is later than the date in *expression*.

►►──NEXT_DAY──(──*expression*──,──*string-expression*────────────)──────────►◄
                                              └─*locale-name*─┘

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

string-expression
> An expression that returns a built-in character data type. The value must be a valid day of the week for the *locale-name*. The value can be specified either as the full name of the day or the associated abbreviation. For example, if the locale is 'en_US' then the following values are valid:

Table 63. Valid day names and abbreviations for the 'en_US' locale

Day of week	Abbreviation
MONDAY	MON
TUESDAY	TUE
WEDNESDAY	WED
THURSDAY	THU
FRIDAY	FRI
SATURDAY	SAT
SUNDAY	SUN

> The minimum length of the input value is the length of the abbreviation. The characters can be specified in lower or upper case and any characters immediately following a valid abbreviation are ignored.

locale-name
> A character constant that specifies the locale used to determine the language of the *string-expression* value. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same data type as *expression*, unless *expression* is a string, in which case the result data type is TIMESTAMP(6) . The result can be null; if any argument is null, the result is the null value.

Any hours, minutes, seconds or fractional seconds information included in *expression* is not changed by the function. If *expression* is a string representing a date, the time information in the resulting TIMESTAMP value is all set to zero.

Notes

- **Determinism:** NEXT_DAY is a deterministic function. However, when locale-name is not explicitly specified, the invocation of the function depends on the value of the special register CURRENT LOCALE LC_TIME. Invocations of

the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Set the variable NEXTDAY with the date of the Tuesday following April 24, 2007.

```
   SET NEXTDAY = NEXT_DAY(DATE '2007-04-24', 'TUESDAY')
```

The variable NEXTDAY is set with the value of '2007-05-01', since April 24, 2007 is itself a Tuesday.

- *Example 2*: Set the variable vNEXTDAY with the timestamp of the first Monday in May, 2007. Assume the variable vDAYOFWEEK = 'MON'.

```
   SET vNEXTDAY = NEXT_DAY(LAST_DAY(CURRENT_TIMESTAMP),vDAYOFWEEK)
```

The variable vNEXTDAY is set with the value of '2007-05-07-12.01.01.123456', assuming that the value of the CURRENT_TIMESTAMP special register is '2007-04-24-12.01.01.123456'.

NORMALIZE_ DECFLOAT

The NORMALIZE_DECFLOAT function returns a decimal floating-point value equal to the input argument set to its simplest form; that is, a nonzero number with trailing zeros in the coefficient has those zeros removed.

►►—NORMALIZE_DECFLOAT—(—*expression*—)————————————————————►◄

The schema is SYSIBM.

Returning a decimal floating-point value equal to the input argument set to its simplest form may require representing the number in normalized form by dividing the coefficient by the appropriate power of ten and adjusting the exponent accordingly. A zero value has its exponent set to 0.

expression
> An expression that returns a value of any built-in numeric data type. Arguments of type SMALLINT, INTEGER, REAL, DOUBLE, or DECIMAL(p,s), where $p <= 16$, are converted to DECFLOAT(16) for processing. Arguments of type BIGINT or DECIMAL(p,s), where $p > 16$, are converted to DECFLOAT(34) for processing.

The result of the function is a DECFLOAT(16) value if the data type of expression after conversion to decimal floating-point is DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. If the argument is a special decimal floating-point value, the result is the same special decimal floating-point value. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

The following examples show the values that are returned by the NORMALIZE_DECFLOAT function, given a variety of input decimal floating-point values:

```
NORMALIZE_DECFLOAT(DECFLOAT(2.1)) = 2.1
NORMALIZE_DECFLOAT(DECFLOAT(-2.0)) = -2
NORMALIZE_DECFLOAT(DECFLOAT(1.200)) = 1.2
NORMALIZE_DECFLOAT(DECFLOAT(-120)) = -1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(120.00)) = 1.2E+2
NORMALIZE_DECFLOAT(DECFLOAT(0.00)) = 0
NORMALIZE_DECFLOAT(-NAN) = -NaN
NORMALIZE_DECFLOAT(-INFINITY) = -Infinity
```

NULLIF

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

►►──NULLIF──(──*expression1*──,──*expression2*──)──────────────────────────►◄

The schema is SYSIBM.

expression1
> An expression that returns a value of any built-in or user-defined data type.

expression2
> An expression that returns a value of any built-in or user-defined data type that is comparable with the data type of the other argument according to the rules for equality comparison.

The result of using NULLIF(e1,e2) is the same as using the following expression:

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

When e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Notes

- The NULLIF function cannot be used as a source function when creating a user-defined function. Because this function accepts any comparable data types as arguments, it is not necessary to create additional signatures to support user-defined data types.

Example
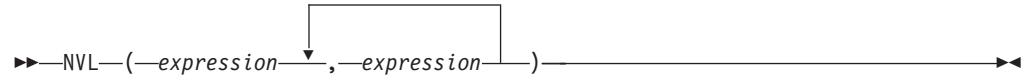
Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

```
NULLIF (:PROFIT + :CASH , :LOSSES )
```

Returns a null value.

NVL

The NVL function returns the first argument that is not null.

```
►►─NVL─(─expression──,─expression──)──────────────────────►◄
```

The schema is SYSIBM.

NVL is a synonym for COALESCE.

NVL2

The NVL2 function returns the second argument when the first argument is not NULL. If the first argument is NULL, the third argument is returned.

►►—NVL2—(—*expression*—,—*result-expression*—,—*else-expression*—)————————►◄

The schema is SYSIBM.

The NVL2 function is a synonym for the following statement:

```
CASE WHEN expression IS NOT NULL
     THEN result-expression
     ELSE else-expression
```

OCTET_LENGTH

The OCTET_LENGTH function returns the length of *expression* in octets (bytes).

►►──OCTET_LENGTH──(──*expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

expression
    An expression that returns a value that is a built-in string data type.

The result of the function is INTEGER. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The length of character or graphic strings includes trailing blanks. The length of binary strings includes binary zeros. The length of varying-length strings is the actual length and not the maximum length.

For greater portability, code your application to be able to accept a result of data type DECIMAL(31).

Examples

- *Example 1*: Assume that table T1 has a GRAPHIC(10) column named C1.

    SELECT OCTET_LENGTH(C1) **FROM** T1

    returns the value 20.
- *Example 2*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

    Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively.
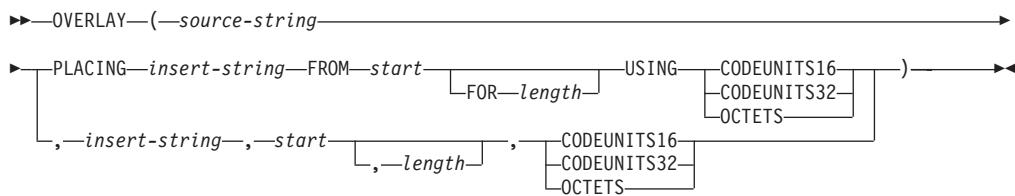
    SELECT OCTET_LENGTH(UTF8_VAR)**,**
      OCTET_LENGTH(UTF16_VAR)
    FROM SYSIBM.SYSDUMMY1

    returns the values 9 and 12, respectively.

OVERLAY

The OVERLAY function returns a string in which, beginning at *start* in *source-string*, *length* of the specified code units have been deleted and *insert-string* has been inserted.

```
►►──OVERLAY──(──source-string──────────────────────────────────────────────────►

►──┬─PLACING──insert-string──FROM──start────┬──────────────┬──USING──┬─CODEUNITS16─┬──)──►◄
   │                          └─FOR──length──┘                        ├─CODEUNITS32─┤
   └─,──insert-string──,──start──┬──────────────┬──,──┬─CODEUNITS16─┬─┘ └─OCTETS──────┘
                                 └─,──length─────┘     ├─CODEUNITS32─┤
                                                       └─OCTETS──────┘
```

The schema is SYSIBM.

source-string
> An expression that specifies the source string. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function.

insert-string
> An expression that specifies the string to be inserted into *source-string*, starting at the position identified by *start*. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. If the code page of the *insert-string* differs from that of the *source-string*, *insert-string* is converted to the code page of the *source-string*.

start
> An expression that returns an integer value. The integer value specifies the starting point within the source string where the deletion and the insertion of another string is to begin. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value is the starting point in code units using the specified string units. The integer value must be between 1 and the actual length of *source-string* in the specified string units plus one (SQLSTATE 42815). If OCTETS is specified and the result is graphic data, the value must be an odd number between 1 and the actual octet length of *source-string* plus one (SQLSTATE 428GC or 22011).

length
> An expression that specifies the number of code units (in the specified string units) that are to be deleted from the source string, starting at the position identified by *start*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be a positive integer or zero (SQLSTATE 22011). If OCTETS is specified and the result is graphic data, the value must be an even number or zero (SQLSTATE 428GC).
>
> Not specifying *length* is equivalent to specifying a value of 1, except when OCTETS is specified and the result is graphic data, in which case, not specifying *length* is equivalent to specifying a value of 2.

CODEUNITS16, CODEUNITS32, or OCTETS
> Specifies the string unit of *start* and *length*.

CODEUNITS16 specifies that *start* and *length* are expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *start* and *length* are expressed in 32-bit UTF-32 code units. OCTETS specifies that *start* and *length* are expressed in bytes.

If the string unit is specified as CODEUNITS16 or CODEUNITS32, and the result is a binary string or 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 CODEUNITS16 or OCTETS, and the string units of *source-string* is CODEUNITS32, an error is returned (SQLSTATE 428GC). If the string unit is specified as OCTETS, the operation is performed in the code page of the *source-string*. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The data type of the result depends on the data types of *source-string* and *insert-string*, as shown in the following tables of supported type combinations. The string unit of the result is the string unit of *source-string*. If either *source-string* or *insert-string* is defined as FOR BIT DATA the other argument cannot be defined with string units of CODEUNITS32. The second table applies to Unicode databases only.

Table 64. Data type of the result as a function of the data types of source-string and insert-string

source-string	insert-string	**Result**
CHAR or VARCHAR	CHAR or VARCHAR	VARCHAR
GRAPHIC or VARGRAPHIC	GRAPHIC or VARGRAPHIC	VARGRAPHIC
CLOB	CHAR, VARCHAR, or CLOB	CLOB
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
BLOB	BLOB	BLOB

Table 65. Data type of the result as a function of the data types of source-string and insert-string (Unicode databases only)

source-string	insert-string	**Result**
CHAR or VARCHAR	GRAPHIC or VARGRAPHIC	VARCHAR
GRAPHIC or VARGRAPHIC	CHAR or VARCHAR	VARGRAPHIC
CLOB	GRAPHIC, VARGRAPHIC, or DBCLOB	CLOB
DBCLOB	CHAR, VARCHAR, or CLOB	DBCLOB

A *source-string* can have a length of 0; in this case, *start* must be 1 (as implied by the bounds provided in the description for *start*), and the result of the function is a copy of the *insert-string*.

An *insert-string* can also have a length of 0. This has the effect of deleting the code units identified by *start* and *length* from the *source-string*.

The length attribute of the result is the length attribute of *source-string* plus the length attribute of *insert-string* when the string units of the *source-string* and *insert-string* are the same or the result string units is CODEUNITS32. Special cases are listed in the following table.

Table 66. Data type of the result as a function of the data types of source-string and insert-string (special cases, Unicode databases only)

source-string		insert-string		Result	
Data type	String units	Data type	String units	Length attribute	String units
Character string with length attribute A	OCTETS	Graphic string with length attribute B	CODEUNITS16	A+3*B	OCTETS
			CODEUNITS32	A+4*B	
		Character with length attribute B	CODEUNITS32	A+4*B	
Graphic string with length attribute A	CODEUNITS16	Character with length attribute B	OCTETS	A+B	CODEUNITS16
			CODEUNITS32	A+2*B	
		Graphic string with length attribute B	CODEUNITS32	A+2*B	

The actual length of the result depends on the actual length of *source-string*, the actual length of the of the deleted string, the actual length of the *insert-string*, and string units used for the *start* and *length* arguments. For example, if the string arguments are character strings in OCTETS and the OCTETS is used as the fourth argument, the actual length of the result is A1 - MIN((A1 - V2 + 1), V3) + A4, where:

- A1 is the actual length of *source-string*
- V2 is the value of *start*
- V3 is the value of *length*
- A4 is the actual length of *insert-string*

If the actual length of the result string exceeds the maximum for the return data type, an error is returned (SQLSTATE 54006).

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Create the strings 'INSISTING', 'INSISERTING', and 'INSTING' from the string 'INSERTING' by inserting text into the middle of the existing text.

```
SELECT OVERLAY('INSERTING','IS',4,2,OCTETS),
  OVERLAY('INSERTING','IS',4,0,OCTETS),
  OVERLAY('INSERTING','',4,2,OCTETS)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 2*: Create the strings 'XXINSERTING', 'XXNSERTING', 'XXSERTING', and 'XXERTING' from the string 'INSERTING' by inserting text before the existing text, using 1 as the starting point.

```
SELECT OVERLAY('INSERTING','XX',1,0,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,1,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,2,CODEUNITS16)),
  OVERLAY('INSERTING','XX',1,3,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 3*: Create the string 'ABCABCXX' from the string 'ABCABC' by inserting text after the existing text. Because the source string is 6 characters long, set the starting position to 7 (one plus the length of the source string).

```
SELECT OVERLAY('ABCABC','XX',7,0,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: Change the string 'Hegelstraße' to 'Hegelstrasse'.

```
SELECT OVERLAY('Hegelstraße','ss',10,1,CODEUNITS16))
  FROM SYSIBM.SYSDUMMY1
```

- *Example 5*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variables UTF8_VAR and UTF16_VAR contain the UTF-8 and the UTF-16BE representations of the string, respectively. Use the OVERLAY function to insert a 'C' into the Unicode string '&N~AB'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS16),
  OVERLAY(UTF8_VAR, 'C', 1, CODEUNITS32),
  OVERLAY(UTF8_VAR, 'C', 1, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'C?N~AB', 'CN~AB', and 'CbbbN~AB', respectively, where '?' represents X'EDB49E', which corresponds to the X'DD1E' in the intermediate UTF-16 form, and 'bbb' replaces the UTF-8 incomplete characters X'9D849E'.

```
SELECT OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS16),
  OVERLAY(UTF8_VAR, 'C', 5, CODEUNITS32),
  OVERLAY(UTF8_VAR, 'C', 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB', '&N~AC', and '&N~AB', respectively.

```
SELECT OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS16),
  OVERLAY(UTF16_VAR, 'C', 1, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'C?N~AB' and 'CN~AB', respectively, where '?' represents the unmatched low surrogate U+DD1E.

```
SELECT OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS16),
  OVERLAY(UTF16_VAR, 'C', 5, CODEUNITS32)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~CB' and '&N~AC', respectively.

PARAMETER

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.

►►──PARAMETER──(──*integer-constant*──)──────────────────────────►◄

The schema is SYSIBM.

integer-constant
> An integer constant that specifies a position index of a value in the arguments of db2-fn:sqlquery. The value must be between 1 and the total number of the arguments specified in the db2-fn:sqlquery SQL statement (SQLSTATE 42815).

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function. The argument of the PARAMETER function determines which value is substituted for the PARAMETER function when the db2-fn:sqlquery function is executed. The value supplied by the PARAMETER function can be referenced multiple times within the same SQL statement.

This function can only be used in a fullselect contained in the string literal argument of the db2-fn:sqlquery function in an XQuery expression (SQLSTATE 42887).
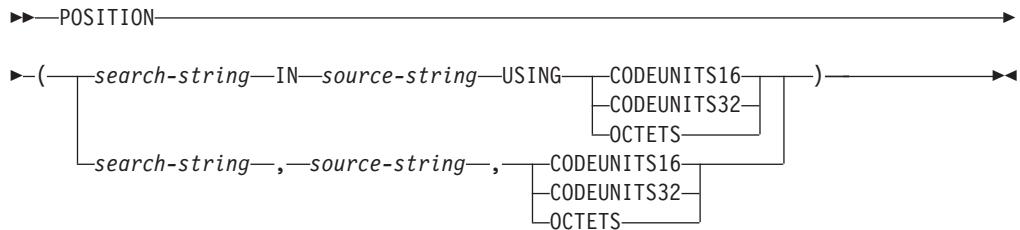
Example

In the following example, the db2-fn:sqlquery function call uses one PARAMETER function call and the XQuery expression $po/@OrderDate, the order date attribute. The PARAMETER function call returns the value of order date attribute:

```
xquery
declare default element namespace "http://posample.org";
for $po in db2-fn:xmlcolumn('PURCHASEORDER.PORDER')/PurchaseOrder,
  $item in $po/item/partid
for $p in db2-fn:sqlquery(
  "select description from product where promostart < PARAMETER(1)",
    $po/@OrderDate )
where $p//@pid = $item
return
<RESULT>
  <PoNum>{data($po/@PoNum)}</PoNum>
  <PartID>{data($item)} </PartID>
  <PoDate>{data($po/@OrderDate)}</PoDate>
</RESULT>
```

The example returns the purchase ID, part ID, and the purchase date for all the parts sold after the promotional start date.

POSITION

The POSITION function returns the starting position of the first occurrence of one string within another string.

```
►►─POSITION──────────────────────────────────────────────────────────────►

►─(─┬─search-string──IN──source-string──USING──┬─CODEUNITS16─┬──┬─)─────────►◄
    │                                           ├─CODEUNITS32─┤
    │                                           └─OCTETS──────┘
    └─search-string──,──source-string──,──┬─CODEUNITS16─┬─────┘
                                          ├─CODEUNITS32─┤
                                          └─OCTETS──────┘
```

The schema is SYSIBM.

The string the POSITION function searches for is called the *search-string*. The string it searches in is called the *source-string*. The POSITION function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). If *search-string* is not found and neither argument is null, the result is zero. If the *search-string* is found, the result is a number from 1 to the actual length of *source-string*, expressed in the string unit that is explicitly specified. The search is done using the collation of the database, unless *search-string* or *source-string* is defined as 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 CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The expression cannot be a BLOB file reference variable. The expression can be specified by any of the following elements:
- A constant
- A special register
- A host variable
- A scalar function whose operands are any of the previously listed items
- An expression that concatenates (using CONCAT or ||) any of the previously listed items
- An SQL procedure parameter

These rules are similar to those that are described for *pattern-expression* for the LIKE predicate.

source-string
The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. The expression can be specified by any of the following elements:
- 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 previously listed items

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 about compatibility, see "Rules for string conversions". In a Unicode database, if one string argument is character (not FOR BIT DATA) and the other string argument is graphic, then the *search-string* is converted to the data type of the *source-string* for processing. If one argument is character FOR BIT DATA, the other argument must not be graphic (SQLSTATE 42846).

The result of the function is a large integer. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Select the RECEIVED column, the SUBJECT column, and the starting position of the string 'GOOD BEER' within the NOTE_TEXT column for all rows in the IN_TRAY table that contain that string.

  ```
  SELECT RECEIVED, SUBJECT, POSITION('GOOD BEER', NOTE_TEXT, OCTETS)
  FROM IN_TRAY
  WHERE POSITION('GOOD BEER', NOTE_TEXT, OCTETS) <> 0
  ```

- *Example 2*: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in CODEUNITS32 units, within the string.

  ```
  SET :LOCATION = POSITION(
    'ß', 'Jürgen lives on Hegelstraße', CODEUNITS32
  )
  ```

  The value of host variable LOCATION is set to 26.

- *Example 3*: Find the position of the character 'ß' in the string 'Jürgen lives on Hegelstraße', and set the host variable LOCATION with the position, as measured in OCTETS, within the string.

  ```
  SET :LOCATION = POSITION(
    'ß', 'Jürgen lives on Hegelstraße', OCTETS
  )
  ```

The value of host variable LOCATION is set to 27.

- *Example 4*: The following examples work with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the non-spacing combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variable UTF8_VAR contains the UTF-8 representation of the string.

```
SELECT POSITION('N', UTF8_VAR, CODEUNITS16),
  POSITION('N', UTF8_VAR, CODEUNITS32),
  POSITION('N', UTF8_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 3, 2, and 5, respectively.

Assume that the variable UTF16_VAR contains the UTF-16BE representation of the string.

```
SELECT POSITION('B', UTF16_VAR, CODEUNITS16),
    POSITION('B', UTF16_VAR, CODEUNITS32),
    POSITION('B', UTF16_VAR, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 6, 5, and 11, respectively.

- *Example 5*: In a Unicode database created with the case insensitive collation CLDR181_LEN_S1, find the position of the word 'Brown' in the phrase 'The quick brown fox'.

```
SET :LOCATION = POSITION('Brown', 'The quick brown fox', CODEUNITS16)
```

The value of the host variable LOCATION is set to 11.

POSSTR

The POSSTR function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*).

►►──POSSTR──(──*source-string*──,──*search-string*──)──────────────────────►◄

The schema is SYSIBM.

Numbers for the *search-string* position start at 1 (not 0).

The result of the function is a large integer. If either of the arguments can be null, the result can be null; if either of the arguments is null, the result is the null value.

source-string
> An expression that specifies the source string in which the search is to take place.
>
> The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. 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 that concatenates (using CONCAT or ||) any of the previously listed items

search-string
> An expression that specifies the string that is to be searched for.
>
> The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, BLOB, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, VARGRAPHIC or BLOB data type, it is implicitly cast to VARCHAR before evaluating the function. The actual length must not be greater than maximum length of a VARCHAR. The expression cannot be a BLOB file reference variable. 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 previously listed items
> - An expression that concatenates (using CONCAT or ||) any of the previously listed items
> - An SQL procedure parameter
>
> The following examples are invalid string expressions or strings:
> - SQL user-defined function parameters
> - Trigger transition variables

- Local variables in compound SQL (inlined) 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 double byte. POSSTR operates on a strict byte-count basis, without awareness of either the database collation or changes between single and multi-byte characters. The POSITION, LOCATE, or LOCATE_IN_STRING functions can be used to operate with awareness of the database collation and the string units.

The following rules apply:
- The data types of *source-string* and *search-string* must be compatible, otherwise an error is raised (SQLSTATE 42884).
    - If *source-string* is a character string, then *search-string* must be a character string, but not a CLOB, with an actual length of 32 672 bytes or less.
    - If *source-string* is a graphic string, then *search-string* must be a graphic string, but not a DBCLOB, with an actual length of 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

The POWER function returns the result of raising the first argument to the power of the second argument.

►►─POWER─(─*expression1*─,─*expression2*─)──────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the POWER function continues to be available.)

expression1
    An expression that returns a value of any built-in numeric data type.

expression2
    An expression that returns a value of any built-in numeric data type.

    If the value of expression1 is equal to zero, then expression2 must be greater than or equal to zero. If both arguments are 0, the result is 1. If the value of expression1 is less than zero, then expression2 must be an integer value.

The result of the function is:
- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DECFLOAT(34) if one of the arguments is decimal floating-point. If either argument is a DECFLOAT and one of the following statements is true, the result is NAN and an invalid operation condition:
    - Both arguments are zero
    - The second argument has a nonzero fractional part
    - The second argument has more than 9 digits
    - The second argument is INFINITY
- DOUBLE otherwise

If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" on page 248 in "Expressions" on page 239.

The result can be null; if any argument is null, the result is the null value.

Example

Assume that the host variable HPOWER is an integer with a value of 3.

    VALUES POWER(2,:HPOWER)

Returns the value 8.

QUANTIZE

The QUANTIZE function returns a decimal floating-point value that is equal in value (except for any rounding) and sign to *numeric-expression* and that has an exponent equal to the exponent of *exp-expression*.

The number of digits (16 or 34) is the same as the number of digits in *numeric-expression*.

►►──QUANTIZE──(──*numeric-expression*──,──*exp-expression*──)──────────────────────►◄

The schema is SYSIBM.

numeric-expression
> An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

exp-expression
> An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing. The *exp-expression* is used as an example pattern for rescaling *numeric-expression*. The sign and coefficient of *exp-expression* are ignored.

The coefficient of the result is derived from that of *numeric-expression*. It is rounded, if necessary (if the exponent is being increased), multiplied by a power of ten (if the exponent is being decreased), or remains unchanged (if the exponent is already equal to that of *exp-expression*).

The CURRENT DECFLOAT ROUNDING MODE special register determines the rounding mode.

Unlike other arithmetic operations on the decimal floating-point data type, if the length of the coefficient after the quantize operation is greater than the precision specified by *exp-expression*, the result is NaN and a warning is returned (SQLSTATE 0168D). This ensures that, unless there is a warning condition, the exponent of the result of QUANTIZE is always equal to that of *exp-expression*.
- if either argument is NaN, NaN is returned
- if either argument is sNaN, NaN is returned and a warning is returned (SQLSTATE 01565)
- if both arguments are infinity (positive or negative), infinity with the same sign as the first argument is returned
- if one argument is infinity (positive or negative) and the other argument is not infinity, NaN is returned and a warning is returned (SQLSTATE 0168D)

The result of the function is a DECFLOAT(16) value if both arguments are DECFLOAT(16). Otherwise, the result of the function is a DECFLOAT(34) value. The result can be null; if any argument is null, the result is the null value.

Examples
- *Example 1*: The following examples show the values that are returned by the QUANTIZE function given a variety of input decimal floating-point values and assuming a rounding mode of ROUND_HALF_UP:

```
QUANTIZE(2.17, DECFLOAT(0.001)) = 2.170
QUANTIZE(2.17, DECFLOAT(0.01)) = 2.17
QUANTIZE(2.17, DECFLOAT(0.1)) = 2.2
QUANTIZE(2.17, DECFLOAT('1E+0')) = 2
QUANTIZE(2.17, DECFLOAT('1E+1')) = 0E+1
QUANTIZE(2, DECFLOAT(INFINITY)) = NaN     -- warning
QUANTIZE(0, DECFLOAT('1E+5')) = 0E+5
QUANTIZE(217, DECFLOAT('1E-1')) = 217.0
QUANTIZE(217, DECFLOAT('1E+0')) = 217
QUANTIZE(217, DECFLOAT('1E+1')) = 2.2E+2
QUANTIZE(217, DECFLOAT('1E+2')) = 2E+2
```

- *Example 2*: In the following example the value -0 is returned for the QUANTIZE function. The CHAR function is used to avoid the potential removal of the minus sign by a client program:

```
CHAR(QUANTIZE(-0.1, DECFLOAT(1))) = -0
```

QUARTER

Returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.

►►──QUARTER──(──*expression*──)──►◄

The schema is SYSFUN.

expression
   An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string (except DBCLOB), it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

RADIANS

The RADIANS function returns the number of radians for an argument that is expressed in degrees.

►►──RADIANS──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the RADIANS function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

Example

Assume that host variable HDEG is an INTEGER with a value of 180. The following statement:

```
VALUES RADIANS(:HDEG)
```

Returns the value +3.14159265358979E+000.

RAISE_ERROR

The RAISE_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE, SQLCODE -438, and *diagnostic-string*.

►►──RAISE_ERROR──(──*sqlstate*──,──*diagnostic-string*──)──────────────────►◄

The schema is SYSIBM.

The RAISE_ERROR function always returns the null value 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' though 'Z', then the subclass (last three characters) can be any of '0' through '9' or 'A' through 'Z'.

    If the SQLSTATE does not conform to these rules an error occurs (SQLSTATE 428B3).

diagnostic-string
    An expression of type CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

To use this function in a context where the data type cannot be determined, a cast specification must be used to give the null returned value a data type. A CASE expression is where the RAISE_ERROR function will be most useful.

Example

List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

```
SELECT EMPNO,
  CASE WHEN EDUCLVL < 16 THEN 'Diploma'
       WHEN EDUCLVL < 18 THEN 'Graduate'
       WHEN EDUCLVL < 21 THEN 'Post Graduate'
       ELSE RAISE_ERROR('70001',
         'EDUCLVL has a value greater than 20')
  END
FROM EMPLOYEE
```

RAND

The RAND function returns a floating point value between 0 and 1. RAND is a non-deterministic function.

```
►►──RAND──(──┬──────────────┬──)────────────────────────────►◄
             └─expression──┘
```

The schema is SYSFUN.

expression
> The expression the returns a value of built-in SMALLINT or INTEGER data type. The value must be between 0 and 2 147 483 647. If an expression is specified, it is used as the seed value

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 produces the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. The seed value is used only for the first invocation of an instance of the RAND function within a statement. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

The RAND scalar function does not guarantee the uniqueness of the random numbers. Use the GENERATE_UNIQUE scalar function to generate a series unique numbers.

The RAND function relies on the random number facilities of the host operating system. The random number facility of each host might vary in factors such as the number of potential distinct values and the quality of the randomness. For these reasons, the output of this function is not suitable as a source of randomness in a cryptographic system.

REAL

The REAL function returns a single-precision floating-point representation of either a number or a string representation of a number.

Numeric to real

►►──REAL──(──*numeric-expression*──)────────────────────────────────►◄

String to real

►►──REAL──(──*string-expression*──)─────────────────────────────────►◄

The schema is SYSIBM.

Numeric to real

numeric-expression
> An expression that returns a value of any built-in numeric data type.
>
> The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable. If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

String to real

string-expression
> An expression that returns a value that is character-string or Unicode graphic-string representation of a number. The data type of string-expression must not be a CLOB or a DBCLOB (SQLSTATE 42884).
>
> The result is the same number that would result from CAST(string-expression AS REAL). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming a valid numeric constant (SQLSTATE 22018). If the numeric value of the argument is not within the range of single-precision floating-point, an error is returned (SQLSTATE 22003).

The result of the function is a single-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Notes

- The CAST specification should be used to increase the portability of applications.

Example

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. The required result is in single-precision floating point. Therefore, REAL is applied to SALARY so that the division is carried out in floating point (actually double-precision) and then REAL is applied to the complete expression to return the result in single-precision floating point.

```
SELECT EMPNO, REAL(REAL(SALARY)/COMM)
  FROM EMPLOYEE
  WHERE COMM > 0
```

REC2XML

The REC2XML function returns a string formatted with XML tags, containing column names and column data.

►►─REC2XML─(─*decimal-constant*─,─*format-string*─,─*row-tag-string*─────────►

►─┬─,─*column-name*─┬─)──►◄

The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

decimal-constant
> The expansion factor for replacing column data characters. The decimal value must be greater than 0.0 and less than or equal to 6.0. (SQLSTATE 42820).
>
> The *decimal-constant* value is used to calculate the result length of the function. For every column with a character data type, the length attribute of the column is multiplied by this expansion factor before it is added in to the result length.
>
> To specify no expansion, use a value of 1.0. Specifying a value less than 1.0 reduces the calculated result length. If the actual length of the result string is greater than the calculated result length of the function, then an error is raised (SQLSTATE 22001).
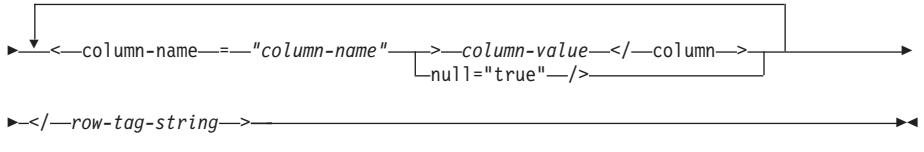
format-string
> The string constant that specifies which format the function is to use during execution.
>
> The *format-string* is case-sensitive, so the following values must be specified in uppercase to be recognized.
>
> **COLATTVAL or COLATTVAL_XML**
>> These formats return a string with columns as attribute values.
>>
>> ►►─<─*row-tag-string*─>───────────────────────────────►
>>
>> ►─┬─<─column-name─=─"*column-name*"─┬─>─*column-value*─</─column─>─┬─►
>> └─null="true"─/>─────────────────────────────┘
>>
>> ►─</─*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 previously listed data types

The same column name cannot be specified more than once (SQLSTATE 42734).

The result of the function is VARCHAR in OCTETS, regardless of the string units of the environment or the specified columns. The maximum length is 32 672 bytes (SQLSTATE 54006).

Consider the following invocation:

```
REC2XML (dc, fs, rt, c₁, c₂, ..., cₙ)
```

If the value of "fs" is either "COLATTVAL" or "COLATTVAL_XML", then the result is the same as this expression:

```
'<' CONCAT rt CONCAT '>' CONCAT y₁ CONCAT y₂
CONCAT ... CONCAT yₙ CONCAT '</' CONCAT rt CONCAT '>'
```

where y_n is equivalent to:

```
'<column name="' CONCAT xvcₙ CONCAT vn
```

and vn is equivalent to:

```
'">' CONCAT rₙ CONCAT '</column>'
```

if the column is not null, and

```
'" null="true"/>'
```

if the column value is null.

xvc_n is equivalent to a string representation of the column name of c_n, where any characters appearing in Table 68 on page 587 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 67 on page 587

Table 67. Column Values String Result

Data type of c_n	r_n
CHAR, VARCHAR	The value is a string. If the *format-string* does not end in the characters "_XML", then each character in c_n is replaced with the corresponding replacement representation from Table 68, as indicated. The length attribute is: dc * the length attribute of c_n.
SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE	The value is LTRIM(RTRIM(CHAR(c_n))). The length attribute is the result length of CHAR(c_n). The decimal character is always the period ('.') character.
DATE	The value is CHAR(c_n,ISO). The length attribute is the result length of CHAR(c_n,ISO).
TIME	The value is CHAR(c_n,JIS). The length attribute is the result length of CHAR(c_n,JIS)
TIMESTAMP	The value is CHAR(c_n). The length attribute is the result length of CHAR(c_n).

Character replacement:

Depending on the value specified for the *format-string*, certain characters in column names and column values will be replaced to ensure that the column names form valid XML attribute values and the column values form valid XML element values.

Table 68. Character Replacements for XML Attribute Values and Element Values

Character	Replacement
<	<
>	>
"	"
&	&
'	'

Examples

Note: REC2XML does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Using the DEPARTMENT table in the sample database, format the department table row, except the DEPTNAME and LOCATION columns, for department 'D01' into an XML string. Since the data does not contain any of the characters which require replacement, the expansion factor will be 1.0 (no expansion). Also note that the MGRNO value is null for this row.

```
SELECT REC2XML (1.0, 'COLATTVAL', '', DEPTNO, MGRNO, ADMRDEPT)
   FROM DEPARTMENT
   WHERE DEPTNO = 'D01'
```

This example returns the following VARCHAR(117) string:

```
<row>
<column name="DEPTNO">D01</column>
<column name="MGRNO" null="true"/>
<column name="ADMRDEPT">A00</column>
</row>
```

- *Example 2*: A 5-day university schedule introduces a class named '&43<FIE' to a table called CL_SCHED, with a new format for the CLASS_CODE column. Using the REC2XML function, this example formats an XML string with this new class data, except for the class end time.

  The length attribute for the REC2XML call with an expansion factor of 1.0 would be 128 (11 for the '<row>' and '</row>' overhead, 21 for the column names, 75 for the '<column name=', '>', '</column>' and double quotation marks, 7 for the CLASS_CODE data, 6 for the DAY data, and 8 for the STARTING data). Since the '&' and '<' characters will be replaced, an expansion factor of 1.0 will not be sufficient. The length attribute of the function will need to support an increase from 7 to 14 bytes for the new format CLASS_CODE data.

  However, since it is known that the DAY value will never be more than 1 digit long, an unused extra 5 units of length are added to the total. Therefore, the expansion only needs to handle an increase of 2. Since CLASS_CODE is the only character string column in the argument list, this is the only column data to which the expansion factor applies. To get an increase of 2 for the length, an expansion factor of 9/7 (approximately 1.2857) would be needed. An expansion factor of 1.3 will be used.

  ```
  SELECT REC2XML (1.3, 'COLATTVAL', 'record', CLASS_CODE, DAY, STARTING)
    FROM CL_SCHED
    WHERE CLASS_CODE = '&43<FIE'
  ```

  This example returns the following VARCHAR(167) string:

  ```
  <record>
  <column name="CLASS_CODE">&amp;43&lt;FIE</column>
  <column name="DAY">5</column>
  <column name="STARTING">06:45:00</column>
  </record>
  ```

- *Example 3*: Assume that new rows have been added to the EMP_RESUME table in the sample database. The new rows store the resumes as strings of valid XML. The COLATTVAL_XML *format-string* is used so character replacement will not be carried out. None of the resumes are more than 3500 bytes in length. The following query is used to select the XML version of the resumes from the EMP_RESUME table and format it into an XML document fragment.

  ```
  SELECT REC2XML (1.0, 'COLATTVAL_XML', 'row', EMPNO, RESUME_XML)
    FROM (SELECT EMPNO, CAST(RESUME AS VARCHAR(3500)) AS RESUME_XML
      FROM EMP_RESUME
      WHERE RESUME_FORMAT = 'XML')
    AS EMP_RESUME_XML
  ```

  This example returns a row for each employee who has a resume in XML format. Each returned row will be a string with the following format:

  ```
  <row>
  <column name="EMPNO">{employee number}</column>
  <column name="RESUME_XML">{resume in XML}</column>
  </row>
  ```

  Where "{employee number}" is the actual EMPNO value for the column and "{resume in XML}" is the actual XML fragment string value that is the resume.

REPEAT

Returns a character string composed of the *expression1* repeated the number of times specified by *expression2*.

►►—REPEAT—(—*expression1*—,—*expression2*—)———————————————————►◄

The schema is SYSFUN.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

expression1
> An expression that returns a value of built-in character string or binary string data 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.

expression2
> An expression that returns a value of built-in SMALLINT or INTEGER data type.

The result of the function is:
- VARCHAR(4000 OCTETS) if the first argument is VARCHAR (not exceeding 4000 bytes) or CHAR
- CLOB(1M OCTETS) if the first argument is CLOB.
- 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 returns the following output:
```
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

Replaces all occurrences of *search-string* in *source-string* with *replace-string*.

```
►►─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.

If *search-string* is not found in *source-string*, *search-string* is returned unchanged. A linguistically correct search is done if the Unicode database is defined with a locale-sensitive UCA-based collation and none of the *source-string*, *search-string* or *replace-string* arguments are defined as FOR BIT DATA. Otherwise, the search is done using a binary comparison with no special consideration for multi-byte characters.

source-string
> An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

search-string
> An expression that specifies the string to be removed from the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

replace-string
> An expression that specifies the replacement string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If the expression is an empty string or is not specified, 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 string units of the result is the same as the string units of *source-string*. The length attribute of the result depends on the arguments:
* The length attribute of the result is the length attribute of the source-string (with the same string units) in the following cases:
  – The *replace-string* argument is not specified or is specified as an empty string constant.

- – The search-string is a constant and the number of bytes in the search-string constant is greater than or equal to:
  - - The number of bytes of a constant *replace-string*.
  - - The length attribute of a non-constant character string type *replace-string* in OCTETS.
  - - The length attribute times 2 of a non-constant graphic string type *replace-string* in double bytes or CODEUNITS16.
  - - The length attribute times 4 of a non-constant *replace-string* in CODEUNITS32.
- Otherwise, the length attribute of the result is determined by the following calculation depending on the data type of the result (that allows for the smallest possible *search-string* to be replaced by the largest possible *replace-string*):
  - – For VARCHAR with string units of OCTETS:
    - - If L1 < = 4000, the length attribute of the result is MIN(4000, (L3*(L1/L2)) + MOD(L1,L2))
    - - Otherwise, the length attribute of the result is MIN(32672, (L3*(L1/L2)) + MOD(L1,L2))
  - – For VARCHAR with string units of CODEUNITS32 the length attribute of the result is MIN(8168, (L3*(L1/L2)) + MOD(L1,L2)).
  - – For VARGRAPHIC with string units of double bytes or CODEUNITS16:
    - - If L1 < = 2000, the length attribute of the result is MIN(2000, (L3*(L1/L2)) + MOD(L1,L2))
    - - Otherwise, the length attribute of the result is MIN(16336, (L3*(L1/L2)) + MOD(L1,L2))
  - – For VARGRAPHIC with string units of CODEUNITS32 the length attribute of the result is MIN(8168, (L3*(L1/L2)) + MOD(L1,L2)).

where the length values for L1, L2, and L3 are given in the following table based on the string units of the result:

REPLACE

Table 69. Length values of L1, L2, and L3

String units of the result	L1	L2	L3
OCTETS (a character string)	Length attribute of *source-string*	One (1) if not a constant, otherwise the actual number of bytes in the constant expressed as a character string type.	• If *replace-string* is a constant, the actual number of bytes in the constant expressed as a character string type. • If *replace-string* is a character string in OCTETS then the length attribute of *replace-string*. • If *replace-string* is a character string or a graphic string in CODEUNITS32, then 4 times the length attribute of *replace-string*. • If *replace-string* is a graphic string in double bytes or CODEUNITS16, then 3 times the length attribute of *replace-string*.
CODEUNITS16 or double bytes (a graphic string)	Length attribute of *source-string*	One (1) if not a constant, otherwise the actual number of double bytes or CODEUNITS16 string units in the constant expressed as a graphic string type.	• If *replace-string* is a constant, the actual number of double bytes or CODEUNITS16 string units in the constant expressed as a graphic string type. • If *replace-string* is a character string in OCTETS or a graphic string in double bytes or CODEUNITS16 then the length attribute of *replace-string*. • If *replace-string* is a character string or graphic string in CODEUNITS32 then the 2 times the length attribute of *replace-string*.

Table 69. Length values of L1, L2, and L3 (continued)

String units of the result	L1	L2	L3
CODEUNITS32 (a character string or a graphic string)	Length attribute of *source-string*	One (1) if not a constant, otherwise length attribute of *search-string*.	Length attribute of *replace-string*

If the result is a character string, the length attribute of the result must not exceed the maximum length of the VARCHAR data type in the string units of the result. If the result is a graphic string, the length attribute of the result must not exceed the maximum length of the VARGRAPHIC data type in the string units of the result.

The actual length of the result is the actual length of *source-string* plus the number of occurrences of *search-string* that exist in *source-string* multiplied by the actual length of *replace-string* minus the actual length of *search-string*.

If the actual length of the *replace-string* exceeds the maximum for the return data type, an error is returned. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Replace all occurrences of the letter 'N' in the word 'DINING' with 'VID'.

      VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)

  The result is the string 'DIVIDIVIDG'.

- *Example 2*: In a Unicode database with case-insensitive collation CLDR181_LEN_S1, replace the word 'QUICK' with the word 'LARGE'.

      VALUES REPLACE ('The quick brown fox', 'QUICK', 'LARGE')

  The result is the string 'The LARGE brown fox'.

REPLACE (SYSFUN schema)

Replaces all occurrences of *expression2* in *expression1* with *expression3*.

►►──REPLACE──(──*expression1*──,──*expression2*──,──*expression3*──)────────────►◄

The schema is SYSFUN.

The search is done using a binary comparison with no special consideration for multi-byte characters.

expression1 **or** *expression2* **or** *expression3*
> The data type for the arguments can be of any built-in character string or binary string type.
>
> 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 or a binary string, the maximum length is 1 048 576 bytes. CHAR is converted to VARCHAR and LONG VARCHAR is converted to CLOB(1M).

The result can be null; if any argument is null, the result is the null value.

Example

Replace all occurrence of the letter 'N' in the word 'DINING' with 'VID'.

```
VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)
```
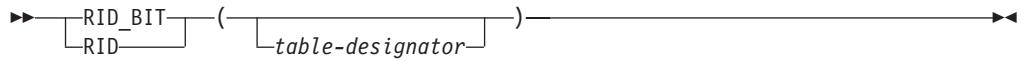
This example returns the following output:

```
1
----------
DIVIDIVIDG
```

The output of the REPLACE function is VARCHAR(4000). For this example, the CHAR function has been used to limit the output of REPLACE to 10 bytes.

RID_BIT and RID

The RID_BIT and RID functions return the row identifier (RID) of a row in different formats.

```
►►──┬─RID_BIT─┬──(─┬──────────────────┬─)──────────────────►◄
    └─RID─────┘    └─table-designator─┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

The RID is used to uniquely identify a row. Each function might return different values when it is invoked multiple times against a row. For example, after the reorg utility is run against a table, the RID_BIT and RID functions might return different values for a row than would have been returned before running the utility. The RID_BIT and RID functions are non-deterministic. The RID_BIT function result, unlike the RID function, contains table information to protect from inadvertently using it with a different table. The RID function is not supported in a partitioned database environment.

table-designator
> Uniquely identifies a base table, view, or nested table expression (SQLSTATE 42703). If *table-designator* specifies a view or nested table expression, the RID_BIT and RID functions return the RID of the base table of the view or nested table expression. The specified view or nested table expression must contain only one base table in its outer subselect (SQLSTATE 42703). The *table-designator* must be deletable (SQLSTATE 42703). For information about deletable views, see the "Notes" section of "CREATE VIEW".
>
> If *table-designator* is not specified, the FROM clause must contain only one element which can be derived to be the table designator (SQL STATE 42703).

The result of the RID_BIT function is VARCHAR (16) FOR BIT DATA. The result can be null. The result of the RID function is BIGINT. The result can be null.

Notes

- To implement optimistic locking in an application, use the values returned by the ROW CHANGE TOKEN expression as arguments to the RID_BIT scalar function.
- The RID_BIT and RID scalar functions are not supported with column-organized tables.
- *Syntax alternatives*: The following syntax alternatives are supported for compatibility with previous versions of DB2 and with other database products. These alternatives are non-standard and should not be used.
  - The pseudocolumn "ROWID" can be used to refer to the RID. An unqualified ROWID reference is equivalent to RID_BIT() and a qualified ROWID such as EMPLOYEE.ROWID is equivalent to RID_BIT(EMPLOYEE).

Examples

- *Example 1*: Return the RID and last name of employees in department 20 from the EMPLOYEE table.

```
SELECT RID_BIT (EMPLOYEE), ROW CHANGE TOKEN FOR EMPLOYEE, LASTNAME
   FROM EMPLOYEE
   WHERE DEPTNO = '20'
```

- *Example 2*: Given table EMP1, which is defined as follows:

```
CREATE TABLE EMP1 (
  EMPNO   CHAR(6),
  NAME    CHAR(30),
  SALARY  DECIMAL(9,2),
  PICTURE BLOB(250K),
  RESUME  CLOB(32K)
)
```

Set host variable HV_EMP_RID to the value of the RID_BIT built-in scalar
function, and HV_EMP_RCT to the value of the ROW CHANGE TOKEN
expression for the row corresponding to employee number 3500.
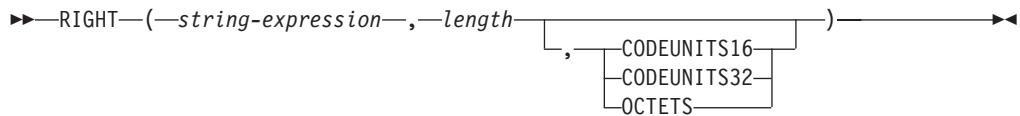
```
SELECT RID_BIT(EMP1), ROW CHANGE TOKEN FOR EMP1
  INTO :HV_EMP_RID, :HV_EMP_RCT FROM EMP1
  WHERE EMPNO = '3500'
```

Using that RID value to identify the employee, and user-defined function
UPDATE_RESUME, increase the employee's salary by $1000 and update the
employee's resume.

```
UPDATE EMP1 SET
  SALARY = SALARY + 1000,
  RESUME = UPDATE_RESUME(:HV_RESUME)
WHERE RID_BIT(EMP1) = :HV_EMP_RID
AND ROW CHANGE TOKEN FOR EMP1 = :HV_EMP_RCT
```

RIGHT

The RIGHT function returns the rightmost string of *string-expression* of length *length*, expressed in the specified string unit.

```
►►──RIGHT──(──string-expression──,──length──────────────────────)──────────►◄
                                          └─,─┬──CODEUNITS16──┬─┘
                                              ├──CODEUNITS32──┤
                                              └──OCTETS───────┘
```

The schema is SYSIBM. The SYSFUN version of the RIGHT function continues to be available.

If *string-expression* is a character string, the result is a character string. If *string-expression* is a graphic string, the result is a graphic string

string-expression
An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string-expression* is zero or more contiguous code points of *string-expression*.

length
An expression that specifies the length of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be between 0 and the length of *string-expression*, expressed in units that are either implicitly or explicitly specified (SQLSTATE 22011) with one exception. If the value is specified as a constant without explicitly specifying the string unit, the value can exceed the length attribute of *string-expression* in the implicit string unit. 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 string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The *string-expression* is padded on the right with the necessary number of padding characters so that the specified substring of *string-expression* always exists. The character used for padding is the same character that is used to pad the string in contexts where padding would occur. For more information about padding, see "String assignments" in "Assignments and comparisons".

The result of the function is a varying-length string with a length attribute that depends on how *length* and the string unit are specified. If *length* is not specified using a constant or the string unit is explicitly specified, then the length attribute is the same as the length attribute of *string-expression*. If *length* is specified using a constant and the string unit is not specified, then the length attribute is the maximum of *length* and the length attribute of *string-expression* up to the maximum length for the result data type in the string units of *string-expression*.. The data type of the result 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
- BLOB if *string-expression* is BLOB

The string unit of the result is the string unit of *string-expression*. The actual length of the result (in string units) is *length*.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that variable ALPHA has a value of 'ABCDEF'. The following statement:

```
SELECT RIGHT(ALPHA,3)
  FROM SYSIBM.SYSDUMMY1
```

  returns 'DEF', which are the three rightmost characters in ALPHA.

- *Example 2*: Assume that variable NAME, which is defined as VARCHAR(50), has a value of 'KATIE AUSTIN', and that the integer variable LASTNAME_LEN has a value of 6. The following statement:

```
SELECT RIGHT(NAME,LASTNAME_LEN)
  FROM SYSIBM.SYSDUMMY1
```

  returns the value 'AUSTIN'.

- *Example 3*: The following statement returns a zero-length string.

```
SELECT RIGHT('ABCABC',0)
  FROM SYSIBM.SYSDUMMY1
```

- *Example 4*: The FIRSTNME column in the EMPLOYEE table is defined as VARCHAR(12). Find the first name of an employee whose last name is 'BROWN' and return the first name in a 10-byte string.

```
SELECT RIGHT(FIRSTNME, 10)
  FROM EMPLOYEE
  WHERE LASTNAME = 'BROWN'
```

  returns a VARCHAR(12) string that has the value 'DAVID' followed by five blank characters.

- *Example 5*: In a Unicode database, FIRSTNAME is a VARCHAR(12) column. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function...                        Returns...

RIGHT(FIRSTNAME,5,CODEUNITS32)     'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNAME,5,CODEUNITS16)     'ürgen' -- x'C3BC7267656E'
RIGHT(FIRSTNAME,5,OCTETS)          'rgen'  -- x'207267656E', a truncated string
```

- *Example 6*: The following example works with the Unicode string '&N~AB', where '&' is the musical symbol G clef character, and '~' is the combining tilde character. This string is shown in different Unicode encoding forms in the following example:

	'&'	'N'	'~'	'A'	'B'
UTF-8	X'F09D849E'	X'4E'	X'CC83'	X'41'	X'42'
UTF-16BE	X'D834DD1E'	X'004E'	X'0303'	X'0041'	X'0042'

Assume that the variable UTF8_VAR, with a length attribute of 20 bytes, contains the UTF-8 representation of the string.

```
SELECT RIGHT(UTF8_VAR, 2, CODEUNITS16),
  RIGHT(UTF8_VAR, 2, CODEUNITS32),
  RIGHT(UTF8_VAR, 2, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values 'AB', 'AB', and 'AB', respectively.

```
SELECT RIGHT(UTF8_VAR, 5, CODEUNITS16),
  RIGHT(UTF8_VAR, 5, CODEUNITS32),
  RIGHT(UTF8_VAR, 5, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '?N~AB', '&N~AB', and 'N~AB', respectively, where '?' is X'EDB49E'.

```
SELECT RIGHT(UTF8_VAR, 10, CODEUNITS16),
  RIGHT(UTF8_VAR, 10, CODEUNITS32),
  RIGHT(UTF8_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbbb', '&N~ABbbbbb', and '&N~ABb', respectively, where 'b' represents the blank character.

Assume that the variable UTF16_VAR, with a length attribute of 20 code units, contains the UTF-16BE representation of the string.

```
SELECT RIGHT(UTF16_VAR, 2, CODEUNITS16),
  RIGHT(UTF16_VAR, 2, CODEUNITS32),
  RIGHT(UTF16_VAR, 2, OCTETS))
FROM SYSIBM.SYSDUMMY1
```

returns the values 'AB', 'AB', and 'B', respectively.

```
SELECT RIGHT(UTF16_VAR, 5, CODEUNITS16),
  RIGHT(UTF16_VAR, 5, CODEUNITS32),
  RIGHT(UTF16_VAR, 6, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '?N~AB', '&N~AB', and '~AB', respectively, where '?' is the stand-alone low surrogate X'DD1E'.

```
SELECT RIGHT(UTF16_VAR, 10, CODEUNITS16),
  RIGHT(UTF16_VAR, 10, CODEUNITS32),
  RIGHT(UTF16_VAR, 10, OCTETS)
FROM SYSIBM.SYSDUMMY1
```

returns the values '&N~ABbbbb', '&N~ABbbbbb', and '?N~AB', respectively, where 'b' represents the blank character and '?' is X'DD1E'.

ROUND

The ROUND function returns a rounded value of a number or a datetime value.

ROUND numeric

►►──ROUND──(──*numeric-expression1*──┬─,──0──────────────────────┬──)──────────────►◄
                                    └─,──*numeric-expression2*──┘

ROUND datetime

►►──ROUND──(──*datetime-expression*──┬─,──'DD'─────────────────────┬──)──────────►◄
                                   └─,──*format-string*──┬──────────────┬┘
                                                    └─*locale-name*─┘

The schema is SYSIBM. The SYSFUN version of the ROUND numeric function continues to be available.

The return value depends on the first argument:
- If the result of the first argument is a numeric value, the ROUND functions returns a number, rounded to the specified number of places to the right or left of the decimal point.
- If the first argument is a DATE, TIME, or TIMESTAMP the ROUND functions returns a datetime value, rounded to the unit specified by *format-string*.

ROUND numeric

    If *numeric-expression1* is positive, a digit value of 5 or greater is an indication to round to the next higher positive number. For example, ROUND(3.5,0) = 4. If *numeric-expression1* is negative, a digit value of 5 or greater is an indication to round to the next lower negative number. For example, ROUND(-3.5,0) = -4.

    numeric-expression1
        An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to DECFLOAT(34) before evaluating the function.

        If the expression is a decimal floating-point data type, the DECFLOAT rounding mode will not be used. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If a different rounding behavior is wanted, use the QUANTIZE function.

    numeric-expression2
        An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

        If *numeric-expression2* is not negative, *numeric-expression1* is rounded to the absolute value of *numeric-expression2* number of places to the right of the decimal point.

        If *numeric-expression2* is negative, *numeric-expression1* is rounded to the absolute value of *numeric-expression2*+1 number of places to the left of the decimal point.

If the absolute value of a negative *numeric-expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, ROUND(748.58,-4) = 0. If *numeric-expression1* is positive, a digit value of 5 is rounded to the next higher positive number. If *numeric-expression1* is negative, a digit value of 5 is rounded to the next lower negative number.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that the precision is increased by one if the *numeric-expression1* is DECIMAL and the precision is less than 31. For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2). The scale is the same as the scale of the first argument.

If either argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES, the result can be null. If either argument is null, the result is the null value.

This function is not affected by the setting of the CURRENT DECFLOAT ROUNDING MODE special register, even for decimal floating-point arguments. The rounding behavior of ROUND corresponds to a value of ROUND_HALF_UP. If you want behavior for a decimal floating-point value that conforms to the rounding mode specified by the CURRENT DECFLOAT ROUNDING MODE special register, use the QUANTIZE function instead.

ROUND datetime

If *datetime-expression* has a datetime data type, the ROUND function returns *datetime-expression* rounded to the unit specified by the *format-string*. If *format-string* is not specified, *datetime-expression* is rounded to the nearest day, as if 'DD' is specified for *format-string*.

datetime-expression
An expression that must return a value that is a date, a time, or a timestamp. String representations of these data types are not supported and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use with this function; alternatively, you can use the ROUND_TIMESTAMP function for a string representation of a date or timestamp.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in *format-string* specifies how *datetime-expression* should be rounded. For example, if *format-string* is 'DD', a timestamp that is represented by *datetime-expression* is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for the type of *datetime-expression* (SQLSTATE 22007). The default is 'DD', which cannot be used if the data type of *datetime-expression* is TIME.

Allowable values for *format-string* are listed in the table of format elements listed in the Table 1.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format element values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale

(SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same DATE type as *datetime-expression*. The result can be null; if any argument is null, the result is the null value.

The following format elements are used to identify the rounding or truncation unit of the datetime value in the ROUND, ROUND_TIMESTAMP, TRUNCATE and TRUNC_TIMESTAMP functions.

Table 70. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
CC SCC	Century Rounds up to the start of the next century after the 50th year of the century (for example on 1951-01-01-00.00.00). Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1901-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1801-01-01-00.00.00.000000
SYYYY YYYY YEAR SYEAR YYY YY Y	Year Rounds up on July 1st to January 1st of the next year. Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-01-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-01-00.00.00.000000
IYYY IYY IY I	ISO Year Rounds up on July 1st to the first day of the next ISO year. The first day of the ISO year is defined as the Monday of the first ISO week. Not valid for TIME argument.	Input Value: 1897-12-04-12.22.22.000000 Result: 1898-01-03-00.00.00.000000	Input Value: 1897-12-04-12.22.22.000000 Result: 1897-01-04-00.00.00.000000
Q	Quarter Rounds up on the 16th day of the second month of the quarter. Not valid for TIME argument.	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-04-12.12.30.000000 Result: 1999-04-01-00.00.00.000000

Table 70. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
MONTH MON MM RM	Month Rounds up on the 16th day of the month. Not valid for TIME argument.	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-07-01-00.00.00.000000	Input Value: 1999-06-18-12.12.30.000000 Result: 1999-06-01-00.00.00.000000
WW	Same day of the week as the first day of the year. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the year. Not valid for TIME argument.	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-06-00.00.00.000000	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-04-29-00.00.00.000000
IW	Same day of the week as the first day of the ISO year. See "WEEK_ISO scalar function" for details. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the ISO year. Not valid for TIME argument.	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-08-00.00.00.000000	Input Value: 2000-05-05-12.12.30.000000 Result: 2000-05-01-00.00.00.000000
W	Same day of the week as the first day of the month. Rounds up on the 12th hour of the 4th day of the week, with respect to the first day of the month. Not valid for TIME argument.	Input Value: 2000-06-21-12.12.30.000000 Result: 2000-06-22-00.00.00.000000	Input Value: 2000-06-21-12.12.30.000000 Result: 2000-06-15-00.00.00.000000
DDD DD J	Day Rounds up on the 12th hour of the day. Not valid for TIME argument.	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-18-00.00.00.000000	Input Value: 2000-05-17-12.59.59.000000 Result: 2000-05-17-00.00.00.000000

Table 70. Format elements for ROUND, ROUND_TIMESTAMP, TRUNCATE, and TRUNC_TIMESTAMP (continued)

Format element	Rounding or truncating unit	ROUND example	TRUNCATE example
DAY DY D	Starting day of the week. Rounds up with respect to the 12th hour of the 4th day of the week. The first day of the week is based on the locale (see *locale-name*). Not valid for TIME argument.	Input Value: 2000-05-17- 12.59.59.000000 Result: 2000-05-21- 00.00.00.000000	Input Value: 2000-05-17- 12.59.59.000000 Result: 2000-05-14- 00.00.00.000000
HH HH12 HH24	Hour Rounds up at 30 minutes.	Input Value: 2000-05-17- 23.59.59.000000 Result: 2000-05-18- 00.00.00.000000	Input Value: 2000-05-17- 23.59.59.000000 Result: 2000-05-17- 23.00.00.000000
MI	Minute Rounds up at 30 seconds.	Input Value: 2000-05-17- 23.58.45.000000 Result: 2000-05-17- 23.59.00.000000	Input Value: 2000-05-17- 23.58.45.000000 Result: 2000-05-17- 23.58.00.000000
SS	Second Rounds up at half a second.	Input Value: 2000-05-17- 23.58.45.500000 Result: 2000-05-17- 23.58.46.000000	Input Value: 2000-05-17- 23.58.45.500000 Result: 2000-05-17- 23.58.45.000000

Note: The format elements in Table 70 on page 602 must be specified in uppercase.

If a format element that applies to a time part of a value is specified for a date argument, the date argument is returned unchanged. If a format element that is not valid for a time argument is specified for a time argument, an error is returned (SQLSTATE 22007).

Notes

- **Determinism:** ROUND is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a datetime value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Calculate the value of 873.726, rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places, respectively.

```
VALUES (
  ROUND(873.726, 2),
  ROUND(873.726, 1),
  ROUND(873.726, 0),
  ROUND(873.726,-1),
  ROUND(873.726,-2),
  ROUND(873.726,-3),
  ROUND(873.726,-4) )
```

  This example returns:

```
1         2         3         4         5         6         7
--------- --------- --------- --------- --------- --------- ---------
  873.730   873.700   874.000   870.000   900.000  1000.000     0.000
```

- *Example 2*: Calculate using both positive and negative numbers.

```
VALUES (
  ROUND(3.5, 0),
  ROUND(3.1, 0),
  ROUND(-3.1, 0),
  ROUND(-3.5,0) )
```

  This example returns:

```
1    2    3    4
---- ---- ---- ----
 4.0  3.0 -3.0 -4.0
```

- *Example 3*: Calculate the decimal floating-point number 3.12350 rounded to three decimal places.

```
VALUES (
  ROUND(DECFLOAT('3.12350'), 3))
```

  This example returns:

```
1
-----
3.12400
```

- *Example 4*: Set the host variable RND_DT with the input date rounded to the nearest month value.

```
SET :RND_DATE = ROUND(DATE('2000-08-16'), 'MONTH');
```

  The value set is 2000-09-01.

- *Example 5*: Set the host variable RND_TMSTMP with the input timestamp rounded to the nearest year value.

```
SET :RND_TMSTMP = ROUND(TIMESTAMP('2000-08-14-17.30.00'),
                        'YEAR');
```
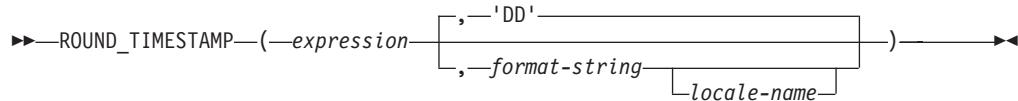
  The value set is 2001-01-01-00.00.00.000000.

ROUND_TIMESTAMP

The ROUND_TIMESTAMP scalar function returns a TIMESTAMP based on a provided argument (*expression*), rounded to the unit specified in another argument (*format-string*).

```
►►─ROUND_TIMESTAMP─(─expression─┬─────────────────────┬─)─────►◄
                                │    ┌─,─'DD'─────┐    │
                                └─,─format-string─┴────┘
                                        └─locale-name─┘
```

The schema is SYSIBM.

If *format-string* is not specified, *expression* is rounded to the nearest day, as if 'DD' is specified for *format-string*.

expression
    An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

format-string
    An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in *format-string* specifies how *expression* should be rounded. For example, if format-string is 'DD', the timestamp that is represented by *expression* is rounded to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is 'DD'.

    Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
    A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a timestamp precision of:
- *p* when the data type of *expression* is TIMESTAMP(*p*)
- 0 when the data type of *expression* is DATE
- 6 otherwise.

The result can be null; if any argument is null, the result is the null value.

Notes
- **Determinism:** ROUND_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Round of a date or timestamp value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Set the host variable *RND_TMSTMP* with the input timestamp rounded to the nearest year value.

```
SET :RND_TMSTMP = ROUND_TIMESTAMP('2000-08-14-17.30.00', 'YEAR');
```

The value set is 2001-01-01-00.00.00.000000.

RPAD

The RPAD function returns a string composed of *string-expression* padded on the right, with *pad* or blanks.

```
►►──RPAD──(──string-expression──,──integer─────────────)────────────────────►◄
                                            └─, pad─┘
```

The schema is SYSIBM.

The RPAD function treats leading or trailing blanks in *string-expression* as significant. Padding will only occur if the actual length of *string-expression* is less than *integer*, and *pad* is not an empty string.

string-expression
    An expression that specifies the source string. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

integer
    An integer expression that specifies the length of the result in the string units of *string-expression*. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be zero or a positive integer that is less than or equal to the maximum length for the result data type in the string units of *string-expression*.

pad
    An expression that specifies the string with which to pad. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

    If *pad* is not specified, the pad character is determined as follows:
- SBCS blank character if *string-expression* is a character string.
- Ideographic blank character, if *string-expression* is a graphic string. For graphic string in an EUC database, X'3000' is used. For graphic string in a Unicode database, X'0020' is used.

The result of the function is a varying length string that has the same string unit and code page as *string-expression*. The value of *string-expression* and the value of *pad* must have compatible data types. If the *string-expression* and *pad* have different code pages, then *pad* is converted to the code page of *string-expression*. If either *string-expression* or *pad* is FOR BIT DATA, no character conversion occurs.

The length attribute of the result depends on whether the value for *integer* is available when the SQL statement containing the function invocation is compiled (for example, if it is specified as a constant or a constant expression) or available only when the function is executed (for example, if it is specified as the result of invoking a function). When the value is available when the SQL statement containing the function invocation is compiled, if *integer* is greater than zero, the length attribute of the result is *integer*. If *integer* is 0, the length attribute of the

result is 1. When the value is available only when the function is executed, the length attribute of the result is determined according to the following table:

Table 71. Determining the result length when integer is available only when the function is executed

Data type of *string-expression*	Result data type length
CHAR(*n*) or VARCHAR(*n*)	Minimum of *n*+100 and 32 672
GRAPHIC(*n*) or VARGRAPHIC(*n*)	Minimum of *n*+100 and 16 336
CHAR(*n*) or VARCHAR(*n*) or GRAPHIC(*n*) or VARGRAPHIC(*n*) with string units of CODEUNITS32 (Unicode database only)	Minimum of *n*+100 and 8 168

The actual length of the result is determined from *integer*. If *integer* is 0 the actual length is 0, and the result is the empty result string. If *integer* is less than the actual length of *string-expression*, the actual length is *integer* and the result is truncated.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples

- *Example 1*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with periods:

  SELECT RPAD(NAME,15,'.') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  ---------------
  Chris..........
  Meg............
  Jeff...........
  ```

- *Example 2*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will completely pad out a value on the right with *pad* (note that in some cases there is a partial instance of the padding specification):

  SELECT RPAD(NAME,15,'123') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  ---------------
  Chris1231231231
  Meg123123123123
  Jeff12312312312
  ```

- *Example 3*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". The following query will only pad each value to a length of 5:

  SELECT RPAD(NAME,5,'.') **AS** NAME **FROM** T1;

  returns:

  ```
  NAME
  -----
  Chris
  Meg..
  Jeff.
  ```

- *Example 4*: Assume that NAME is a CHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that the result of RTRIM is a varying length string with the blanks removed:

  ```
  SELECT RPAD(RTRIM(NAME),15,'.' ) AS NAME FROM T1;
  ```

  returns:

  ```
  NAME
  ---------------
  Chris..........
  Meg............
  Jeff...........
  ```

- *Example 5*: Assume that NAME is a VARCHAR(15) column containing the values "Chris", "Meg", and "Jeff". Note that "Chris" is truncated, "Meg" is padded, and "Jeff" is unchanged:

  ```
  SELECT RPAD(NAME,4,'.' ) AS NAME FROM T1;
  ```

  returns:

  ```
  NAME
  ----
  Chri
  Meg.
  Jeff
  ```

RTRIM

The RTRIM function removes any of the specified characters from the end of a string.

The RTRIM function removes any of the characters contained in a *trim-expression* from the end of a *string-expression*. The character search compares the binary representation of each character (consisting of one or more bytes) in the *trim-expression* to the binary representation of each character (consisting of one or more bytes) at the end of the *string-expression*. The database collation does not affect the search. If the *string-expression* is defined as FOR BIT DATA, the search compares each byte in the *trim-expression* to the byte at the end of the *string-expression*.

```
►►─RTRIM─(─string-expression─┬──────────────────────┬─)─────────────►◄
                             └─,─trim-expression─┘
```

The schema is SYSIBM. (The SYSFUN version of this function that uses a single parameter continues to be available with support for CLOB arguments.)

string-expression

An expression that specifies the source string.

- If only one argument is specified, the expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the data type of the *string-expression* value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, the value is implicitly cast to VARCHAR before evaluating the function.

- If both arguments are specified, the expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. If the data type of the *string-expression* value is not a character string or graphic string data type, the value is implicitly cast to VARCHAR before evaluating the function. The actual length of a CLOB value is limited to the maximum size of a VARCHAR data type (SQLSTATE 22001). The actual length of a DBCLOB value is limited to the maximum size of a VARGRAPHIC data type (SQLSTATE 22001).

trim-expression

An expression that specifies the characters that are being removed from the end of a *string-expression*. The expression must be a value that is a built-in character string, graphic string, numeric, or datetime data type.

- If the data type of the *trim-expression* is not a character string or graphic string, then the value is implicitly cast to VARCHAR before evaluating the function.

- If the data type of the *trim-expression* is a CLOB, then the actual length of the value is limited to the maximum size of a VARCHAR (SQLSTATE 22001).

- If the data type of the *trim-expression* is a DBCLOB, then the actual length of the value is limited to the maximum size of a VARGRAPHIC (SQLSTATE 22001).

- If the *string-expression* is not defined as FOR BIT DATA, then the *trim-expression* cannot be defined as FOR BIT DATA (SQLSTATE 42815).

When a *trim-expression* is not specified, the data type of the *string-expression* determines the default value used:

- A double byte blank if the *string-expression* is a graphic string in a DBCS or EUC database
- A UCS-2 blank if the *string-expression* is a graphic string in a Unicode database
- A value of X'20' if the *string-expression* is a FOR BIT DATA string
- A single-byte blank for all other cases.

If one function argument is character FOR BIT DATA, then the other argument cannot be a graphic (SQLSTATE 42846). A combination of character string and graphic string arguments can be used only in a Unicode database (SQLSTATE 42815).

The data type of the result depends on the data type of the *string-expression*.
- VARCHAR if the data type is VARCHAR or CHAR
- CLOB if the data type is CLOB
- VARGRAPHIC if the data type is VARGRAPHIC or GRAPHIC
- DBCLOB if the data type is DBCLOB

The length attribute of the result data type is the same as the length attribute of the *string-expression* data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of string units removed. The actual length of the result for graphic strings is the length of *string-expression* minus the number of string units removed. If all of the characters are removed, the result is an empty string with a length of zero.

If any argument can be null, the result can be null; if any argument is null, the result is the null value.

Examples
- *Example 1*: Use the RTRIM function when the host variable HELLO is defined as CHAR(6) and has a value of 'Hello '.

      VALUES RTRIM(:HELLO**)**

  The result is 'Hello'. When a *trim-expression* is not specified only blanks are removed. The host variable is declared as CHAR(9) and is blank-padded up to 9 bytes.
- *Example 2*: Use the RTRIM function to remove the characters specified in the *trim-expression* from the end of the *string-expression*.

      VALUES RTRIM('...VAR...', '$.'**)**

  The result is '**...$VAR**'.
- *Example 3*: Use the RTRIM function to remove the characters specified in the *trim-expression* from the end of the *string-expression*.

      VALUES RTRIM('((-78.0))', '-0. ()'**)**

  The result is '**((-78**'. When removing characters and blanks, you must include a blank in the *trim-expression*.

RTRIM (SYSFUN schema)

Returns the characters of the argument with trailing blanks removed.

►►──RTRIM──(──*expression*──)──►◄

The schema is SYSFUN.

expression
> The *expression* can be of any built-in character string data type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes.

The result of the function is:
- VARCHAR(4000) if *expression* is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if *expression* is CLOB or LONG VARCHAR

The result can be null; if *expression* is null, the result is the null.

SECLABEL

The SECLABEL function returns an unnamed security label with a data type of DB2SECURITYLABEL. Use the SECLABEL function to insert a security label with given component values without having to create a named security label.

►►──SECLABEL──(──*security-policy-name*──,──*security-label-string*──)──────────────►◄

The schema is SYSIBM.

security-policy-name
> A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-string
> An expression that returns a valid representation of a security label for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples

- *Example 1*: The following statement inserts a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. The security label for the row to be inserted is given by the SECLABEL function. The security policy CONTRIBUTIONS has two components. The security label given has the element LIFE MEMBER for first component, the elements BLUE and YELLOW for the second component.

```
INSERT INTO REGIONS
  VALUES (SECLABEL('CONTRIBUTIONS', 'LIFE MEMBER:(BLUE,YELLOW)'),
    1, 'Northeast')
```

- *Example 2*: The following statement inserts a row in table CASE_IDS which is protected by the security policy named TS_SECPOLICY, which has three components. The security label is provided by the SECLABEL function. The security label inserted has the element HIGH PROFILE for the first component, the empty value for the second component and the element G19 for the third component.

```
INSERT INTO CASE_IDS
  VALUES (SECLABEL('TS_SECPOLICY', 'HIGH PROFILE:():G19') , 3, 'KLB')
```

SECLABEL_BY_NAME

The SECLABEL_BY_NAME function returns the specified security label. The security label returned has a data type of DB2SECURITYLABEL. Use this function to insert a named security label.

►►——SECLABEL_BY_NAME——(——*security-policy-name*——,——*security-label-name*——)————————►◄

The schema is SYSIBM.

security-policy-name
> A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label-name
> An expression that returns the name of a security label that exists at the current server for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.

Examples

In the following examples, Tina is trying to insert a row in table REGIONS which is protected by the security policy named CONTRIBUTIONS. Tina wants the row to be protected by the security label named EMPLOYEESECLABEL.

- *Example 1*: This statement fails because CONTRIBUTIONS.EMPLOYEESECLABEL is an unknown identifier:

  ```
  INSERT INTO REGIONS
  VALUES (CONTRIBUTIONS.EMPLOYEESECLABEL, 1, 'Southwest')   -- incorrect
  ```

- *Example 2*: This statement fails because the first value is a string, it does not have a data type of DB2SECURITYLABEL:

  ```
  INSERT INTO REGIONS
  VALUES ('CONTRIBUTIONS.EMPLOYEESECLABEL', 1, 'Southwest') -- incorrect
  ```

- *Example 2*: This statement succeeds because the SECLABEL_BY_NAME function returns a security label that has a data type of DB2SECURITYLABEL:

  ```
  INSERT INTO REGIONS
  VALUES (SECLABEL_BY_NAME('CONTRIBUTIONS', 'EMPLOYEESECLABEL'),
    1, 'Southwest')                                    -- correct
  ```

SECLABEL_TO_CHAR

The SECLABEL_TO_CHAR function accepts a security label and returns a string that contains all elements in the security label. The string is in the security label string format.

►►──SECLABEL_TO_CHAR──(──*security-policy-name*──,──*security-label*──)────────────►◄

The schema is SYSIBM.

security-policy-name
> A string that specifies a security policy that exists at the current server (SQLSTATE 42704). The string must be a character or graphic string constant or host variable.

security-label
> An expression that returns a security label value that is valid for the security policy named by *security-policy-name* (SQLSTATE 4274I). The expression must return a value that is a built-in SYSPROC.DB2SECURITYLABEL distinct type.

The result of the function is VARCHAR(32672 OCTETS). The result can be null; if the second argument is null, the result is the null value.
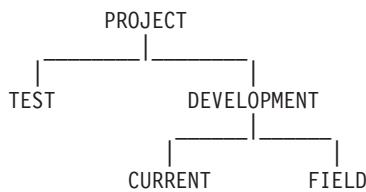
Notes
- If the authorization ID of the statement executes this function on a security label being read from a column with a data type of DB2SECURITYLABEL then that authorization ID's LBAC credentials might affect the output of the function. In such a case an element is not included in the output if the authorization ID does not have read access to that element. An authorization ID has read access to an element if its LBAC credentials would allow it to read data that was protected by a security label containing only that element, and no others.

  For the rule set DB2LBACRULES only components of type TREE can contain elements that you do not have read access to. For other types of component, if any one of the elements block read access then you will not be able to read the row at all. So only components of type tree will have elements excluded in this way.

Example

The EMP table has two columns, RECORDNUM and LABEL; RECORDNUM has data type INTEGER, and LABEL has type DB2SECURITYLABEL. Table EMP is protected by security policy DATA_ACCESSPOLICY, which uses the DB2LBACRULES rule set and has only one component (GROUPS, of type TREE). GROUPS has five elements: PROJECT, TEST, DEVELOPMENT, CURRENT, AND FIELD. The following diagram shows the relationship of these elements to one another:

```
                PROJECT
        |————————|————————|
        |                 |
      TEST           DEVELOPMENT
                   |————————|————————|
                   |                 |
               CURRENT             FIELD
```

The EMP table contains the following data:

```
RECORDNUM  LABEL
---------  ----------------
        1  PROJECT
        2  (TEST, FIELD)
        3  (CURRENT, FIELD)
```

The user whose ID is Djavan holds a security label for reading that contains only the DEVELOPMENT element. This means that Djavan has read access to the DEVELOPMENT, CURRENT, and FIELD elements:

```
SELECT RECORDNUM, SECLABEL_TO_CHAR('DATA_ACCESSPOLICY', LABEL) FROM EMP
```

returns:

```
RECORDNUM  LABEL
---------  ----------------
        2  FIELD
        3  (CURRENT, FIELD)
```

The row with a RECORDNUM value of 1 is not included in the output, because Djavan's LBAC credentials do not allow him to read that row. In the row with a RECORDNUM value of 2, element TEST is not included in the output, because Djavan does not have read access to that element; Djavan would not have been able to access the row at all if TEST were the only element in the security label. Because Djavan has read access to elements CURRENT and FIELD, both elements appear in the output.

Now Djavan is granted an exemption to the DB2LBACREADTREE rule. This means that no element of a TREE type component will block read access. The same query returns:

```
RECORDNUM  LABEL
---------  ----------------
        1  PROJECT
        2  (TEST, FIELD)
        3  (CURRENT, FIELD)
```

This time the output includes all rows and all elements, because the exemption gives Djavan read access to all of the elements.

SECOND

The SECOND function returns the seconds part of a value with optional fractional seconds.

▶▶──SECOND──(──*expression*─────────────────────)────────────────▶◀
 └─,──*integer-constant*─┘

The schema is SYSIBM.

expression
 An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, time duration, timestamp duration, or a valid character string representation of a date, time, or timestamp that is not a CLOB. If *expression* is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). If expression is a valid string representation of a timestamp, it is first converted to a TIMESTAMP(12) value. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

integer-constant
 An integer constant representing the scale for the fractional seconds. The value must be in the range 0 through 12.

The result of the function with a single argument is a large integer. The result of the function with two arguments is DECIMAL(2+*s*,*s*) where *s* is the value of the *integer-constant*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The other rules depend on the data type of the first argument and the number of arguments:
- If the first argument is a DATE, TIME, TIMESTAMP, or valid string representation of a date, time, or timestamp:
  - If only one argument is specified, the result is the seconds part of the value (0 and 59).
  - If both arguments are specified, the result is the seconds part of the value (0 to 59) and *integer-constant* digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value, then zeros are returned.
- If the first argument is a time duration or timestamp duration:
  - If only one argument is specified, the result is the seconds part of the value (-99 to 99).
  - If both arguments are specified, the result is the seconds part of the value (-99 to 99) and *integer-constant* digits of the fractional seconds part of the value where applicable. If there are no fractional seconds in the value then zeros are returned. A nonzero result has the same sign as the argument.

Examples
- *Example 1*: Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.
  ```
  SELECT SECOND(:TIME_DUR)
    FROM SYSIBM.SYSDUMMY1
  ```
  Returns the value 45.

- *Example 2*: Assume that the column RECEIVED (whose data type is TIMESTAMP) has an internal value equivalent to 1988-12-25-17.12.30.000000.

  ```
  SELECT SECOND(RECEIVED)
     FROM IN_TRAY
  ```

  Returns the value 30.

- *Example 3*: Get the seconds with fractional seconds from a current timestamp with milliseconds.

  ```
  SELECT SECOND (CURRENTTIMESTAMP(3), 3)
     FROM SYSIBM.SYSDUMMY1
  ```

  Returns a DECIMAL(5,3) value based on the current timestamp that could be something like 54.321.

SIGN

Returns an indicator of the sign of the argument.

►►──SIGN──(──*expression*──)────────────────────────────────►◄

The schema is SYSIBM. (The SYSFUN version of the SIGN function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type. DECIMAL and REAL values are converted to double-precision floating-point numbers for processing by the function.

If the argument is less than zero, -1 is returned. If the argument is the decimal floating-point value of -0, the decimal floating-point value of -0 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The result of the function is:
- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DECFLOAT(n) if the argument is DECFLOAT(n)
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

Example

Assume that host variable PROFIT is a large integer with a value of 50000.
```
VALUES SIGN(:PROFIT)
```

Returns the value 1.

SIN

Returns the sine of the argument, where the argument is an angle expressed in radians.

>>──SIN──(──*expression*──)────────────────────────────────────><

The schema is SYSIBM. (The SYSFUN version of the SIN function continues to be available.)

expression
  An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

SINH

Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

►►──SINH──(──*expression*──)──►◄

The schema is SYSIBM.

expression
>   An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

SMALLINT

The SMALLINT function returns a small integer representation of either a number or a string representation of a number.

Numeric to Smallint:

►►──SMALLINT──(──*numeric-expression*──)──────────────────────────────────►◄

String to Smallint:

►►──SMALLINT──(──*string-expression*──)───────────────────────────────────►◄

The schema is SYSIBM.

Numeric to Smallint:

numeric-expression
> An expression that returns a value of any built-in numeric data type.

> The result is the same number that would occur if the argument were assigned to a small integer column or variable. The fractional part of the argument is truncated. If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003).

String to Smallint:

string-expression
> An expression that returns a value that is a character-string or Unicode graphic-string representation of a number with a length not greater than the maximum length of a character constant.

> The result is the same number that would result from CAST(*string-expresssion* AS SMALLINT). Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an integer, decimal, floating-point, or decimal floating-point constant (SQLSTATE 22018). If the whole part of the argument is not within the range of small integers, an error is returned (SQLSTATE 22003). The data type of *string-expression* must not be CLOB or DBCLOB (SQLSTATE 42884).

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.

Note: The CAST specification should be used to increase the portability of applications. For more information, see "CAST specification".

Example

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and the employee number (EMPNO).

```
SELECT SMALLINT(SALARY / EDLEVEL), SALARY, ESDLEVEL, EMPNO
   FROM EMPLOYEE
```

SOUNDEX

Returns a 4-character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.

►►—SOUNDEX—(—*expression*—)————————————————————————►◄

The schema is SYSFUN.

expression
> An expression that returns a value of CHAR or VARCHAR data type. The length of the value must not exceed 4 000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. The function interprets data that is passed to it as if it were ASCII characters, even if it is encoded in UTF-8.

The result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function.

Example

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

```
SELECT EMPNO, LASTNAME FROM EMPLOYEE
  WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following output:

```
EMPNO  LASTNAME
------ ---------------
000110 LUCCHESSI
```

SPACE

Returns a character string consisting of blanks with length specified by the argument.

►►──SPACE──(──*expression*──)──►◄

The schema is SYSFUN.

expression
> An expression that returns a value of built-in SMALLINT or INTEGER data type.

The result of the function is VARCHAR(4000 OCTETS). The result can be null; if the argument is null, the result is the null value.

SQRT

The SQRT function returns the square root of a number.

►►──SQRT──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the SQRT function continues to be available.)

expression
> An expression that returns a value of any built-in numeric data type. If the argument is decimal floating-point, the operation is performed in decimal floating-point; otherwise, the argument is converted to double-precision floating-point for processing by the function.

If the argument is DECFLOAT(n), the result is DECFLOAT(n); otherwise, the result is a double-precision floating-point number.

The result can be null; if the argument is null, the result is the null value.

Notes

- *Results involving DECFLOAT special values:* If the argument is a special decimal floating-point value, the rules for general arithmetic operations for decimal floating-point apply. See "General arithmetic operation rules for decimal floating-point" on page 248 in "Expressions" on page 239.

Example
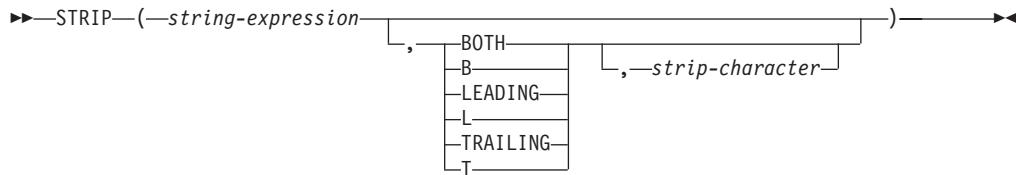
Assume that SQUARE is a DECIMAL(2,1) host variable with a value of 9.0.

```
VALUES SQRT(:SQUARE)
```

Returns the approximate value 3.00.

STRIP

The STRIP function removes blanks or occurrences of another specified character from the end or the beginning of a string expression.

```
►►─STRIP─(─string-expression─┬─────────────────────────────────────┬─)─────►◄
                             │  ┌─BOTH─────┐                        │
                             └─,─┼─B────────┼─┬──────────────────┬─┘
                                 ├─LEADING──┤ └─,─strip-character─┘
                                 ├─L────────┤
                                 ├─TRAILING─┤
                                 └─T────────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The STRIP function is identical to the TRIM scalar function.

string-expression
> An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

BOTH, LEADING, or TRAILING
> Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

strip-character
> A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.
>
> If *strip-character* is not specified and:
> - If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
> - If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
> - 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 string units that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Example

Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.
```
SELECT STRIP(:BALANCE, LEADING, '0'),
FROM SYSIBM.SYSDUMMY1
```

returns the value '345.50'.

SUBSTR

The SUBSTR function returns a substring of a string.

```
►►──SUBSTR──(──string──,──start──┬──────────────┬──)──────────────────►◄
                                 └──,──length──┘
```

The schema is SYSIBM.

string
> An expression that specifies the string from which the result is derived.
>
> The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous string units of *string*.

start
> An expression that specifies the position of the first string unit of the result. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The integer value must be 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 string units 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, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer must be in the range of 0 to n, where n equals (the length attribute of *string* in string units) - *start* + 1 (SQLSTATE 22011, if out of range).
>
> If *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of blank characters (single-byte for character strings; double-byte for graphic strings) or hexadecimal zero characters (for BLOB strings) so that the specified substring of *string* always exists. The default for *length* is the number of string units from the string unit specified by the *start* to the last string unit of *string*. However, if *string* is a varying-length string with a length less than *start*, the default is zero and the result is the empty string. It must be specified as number of string units in the context of the database code page and not the application code page. (For example, the column NAME with a data type of VARCHAR(18) and a value of 'MCKNIGHT' will yield an empty string with SUBSTR(NAME,10)).

If *string* is a character string, the result of the function is a character string represented in the code page and string units 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 and string units of its first argument. If the first argument is a host variable that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the database code page. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

Table 72 on page 629 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

Table 72. Data Type and Length of SUBSTR Result

String Argument Data Type	Length Argument	Result Data Type
CHAR(A)	constant (*l*<*n*) where *n* is 255 if the string units of *string* is OCTETS or 64 if the string units of *string* is CODEUNITS32	CHAR(*l*)
CHAR(A)	not specified but *start* argument is a constant	CHAR(A-*start*+1)
CHAR(A)	not a constant	VARCHAR(A)
VARCHAR(A)	constant (*l*<*n*) where *n* is 255 if the string units of *string* is OCTETS or 64 if the string units of *string* is CODEUNITS32	CHAR(*l*)
VARCHAR(A)	constant (*m*<*l*<*n*) where *m* is 254 and *n* is 32673 if the string units of *string* is OCTETS or *m* is 63 and *n* is 8169 if the string units of *string* is CODEUNITS32	VARCHAR(*l*)
VARCHAR(A)	not a constant or not specified	VARCHAR(A)
CLOB(A)	constant (*l*)	CLOB(*l*)
CLOB(A)	not a constant or not specified	CLOB(A)
GRAPHIC(A)	constant (*l*<*n*) where *n* is 128 if the string units of *string* is double-bytes or CODEUNITS16; or 64 if the string units of *string* is CODEUNITS32	GRAPHIC(*l*)
GRAPHIC(A)	not specified but *start* argument is a constant	GRAPHIC(A-*start*+1)
GRAPHIC(A)	not a constant	VARGRAPHIC(A)
VARGRAPHIC(A)	constant (*l*<*n*) where *n* is 128 if the string units of *string* is double-bytes or CODEUNITS16; or 64 if the string units of *string* is CODEUNITS32	GRAPHIC(*l*)
VARGRAPHIC(A)	constant (*m*<*l*<*n*) where *m* is 127 and *n* is 16337 if the string units of *string* is double-bytes or CODEUNITS16; or *m* is 63 and *n* is 8169 if the string units of *string* is CODEUNITS32	VARGRAPHIC(*l*)
VARGRAPHIC(A)	not a constant	VARGRAPHIC(A)
DBCLOB(A)	constant (*l*)	DBCLOB(*l*)
DBCLOB(A)	not a constant or not specified	DBCLOB(A)
BLOB(A)	constant (*l*)	BLOB(*l*)
BLOB(A)	not a constant or not specified	BLOB(A)

Note: The LONG VARCHAR and LONG VARGRAPHIC data types continue to be supported but are deprecated and not recommended.

If *string* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string*) - *start* + 1. If *string* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*string*) - *start* + 1, whichever is greater.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.

- Though not explicitly stated in the result definitions mentioned previously, the semantics imply that if *string* is a mixed single- and multi-byte character string, the result might contain fragments of multi-byte characters, depending upon the values of *start* and *length*. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTR function does not detect such fragments, nor provide any special processing should they occur.

Examples

- *Example 1*: Assume 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'.

- *Example 2*: Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION'.

      SELECT * **FROM** PROJECT
        WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '

  The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

SUBSTR2

The SUBSTR2 function returns a substring from a string. The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 16-bit UTF-16 string units (CODEUNITS16).

```
►►──SUBSTR2──(──string──,──start──────┬──────────┬──)──────────────────────►◄
                                       └─,length─┘
```

The schema is SYSIBM.

string
> An expression that specifies the string from which the resulting substring is derived. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. A character string cannot have the FOR BIT DATA attribute (SQLSTATE 428GC). If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous bytes of *string*.

start
> An expression that specifies the starting position in *string* for the beginning of the result substring in 16-bit UTF-16 string units. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If *start* is positive, then the starting position is calculated from the beginning of the string. If *start* is greater than the length of *string* in 16-bit UTF-16 string units, then a zero length string is returned.
>
> If *start* is negative, then the starting position is calculated from the end of the string by counting backwards. If the absolute value of *start* is greater than the length of *string* in 16-bit UTF-16 string units, then a zero length string is returned.
>
> If *start* is 0, then a starting position of 1 is used.

length
> An expression that specifies the length of the resulting substring in 16-bit UTF-16 string units. If *length* is specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If the value for *length* is greater than the number of 16-bit UTF-16 string units from the starting position to the end of the string, then the length of the resulting substring is the length of the first argument in 16-bit UTF-16 string units minus the starting position plus one.
>
> If the value for *length* is less than or equal to zero, the result is a zero length string.
>
> The default value for *length* is the number of CODEUNITS16 from the position specified by *start* to the last byte of *string*.

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If *string* is a CLOB, the result of the function is a CLOB. If *string* is a GRAPHIC or VARGRAPHIC data type, the result of the function is a VARGRAPHIC data type. If *string* is a DBCLOB, the result of the function is a

DBCLOB. If the first argument is a host variable, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is based on the first applicable row in the following table.

Table 73. Length Attribute of SUBSTR2 Result when Arguments Include Constants

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	any valid argument	constant value L<=0	0
character type with length attribute A	constant value S and $|(S)|>A$	not specified or any valid argument	0
character type with length attribute A	not a constant	constant value L>0	$MIN(A, L×4)$
character type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
character type with length attribute A	constant value S<0	not specified or not a constant	$MIN(A, |(S)×4|)$
character type with length attribute A	constant value S>0	constant value L>0	$MIN(A-S+1, L×4)$
character type with length attribute A	constant value S<0	constant value L>0	$MIN(A, |(S)×4|, L×4)$
graphic type with length attribute A	any valid argument	constant value L<=0	0
graphic type with length attribute A	constant value S and $|(S)|>A$	not specified or any valid argument	0
graphic type with length attribute A	not a constant	constant value L>0	$MIN(A, L)$
graphic type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
graphic type with length attribute A	constant value S<0	not specified or not a constant	$|(S)|$
graphic type with length attribute A	constant value S>0	constant value L>0	$MIN(A-S+1, L)$
graphic type with length attribute A	constant value S<0	constant value L>0	$MIN(|(S)|, L)|$

Table 73. Length Attribute of SUBSTR2 Result when Arguments Include Constants (continued)

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]

Notes:

[1] If a start argument value of 0 is specified, then use a value of 1 for S when referencing this table.

[2] The length attribute of the result for some of the character result types involves a constant that is multiplied by a factor of 4. This multiplier covers the worst case expansion derived from the following factors:
- Multiplying by 2 to switch from counting in 16-bit UTF-16 string units to counting in bytes used for the length attributes of a character data type.
- Multiplying by 2 again because a 2-byte character in UTF-16 can be represented by up to 4 bytes in a character string.

If any argument of the SUBSTR2 function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If an untyped parameter marker is used for string, the operand will be nullable and if the database supports graphic data types the data type of the operand will be VARGRAPHIC(16336). Otherwise, the data type will be VARCHAR(32672).
- If *string* is a mixed single-byte and multi-byte character string, the result might contain fragments of multi-byte characters depending on the values of *start* and *length*. For example, the result might begin with the third byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTR2 function detects these partial characters and replaces each byte of an incomplete character with a single blank character.
- SUBSTR2 is similar to the SUBSTR function, with the following exceptions:
  - SUBSTR2 supports a negative start value, which indicates that processing is to start from the end of the string.
  - SUBSTR2 supports a *length* value that is greater than the calculated result length. In such cases, a shorter string is returned, rather than an error.
  - SUBSTR2 returns a result data type of VARCHAR if the input data type is CHAR. VARGRAPHIC is the result data type returned if the input type is GRAPHIC.
  - The length attribute of the result for SUBSTR2 is either the same as the length attribute of the first argument, or it is derived based on the start or length attributes, if either of these are constants.
  - SUSTR2 returns a result with a length attribute that is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is derived based on the start or length attributes (see the preceding table).

Examples

- *Example 1*: Given the following host variables:
  - NAME (VARGRAPHIC(50) with a value of 'Roméo Jürgen'

      – SURNAME POS (INTEGER) with a value of 7

```
SUBSTR2(:NAME, :SURNAME_POS)
```

returns the value Jürgen

```
SUBSTR2(:NAME, :SURNAME_POS,2)
```

returns the value Jü

- *Example 2*: Select all rows from the PROJECT table which end in 'ING'

```
SELECT * FROM PROJECT
   WHERE SUBSTR2(PROJNAME,-3) = 'ING'
```

SUBSTR4

The SUBSTR4 function returns a substring from a string.

The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 32-bit UTF-32 string units (CODEUNITS32).

```
►►──SUBSTR4──(──string──,──start──┬──────────────┬──)──────────────────────────►◄
                                   └──,──length──┘
```

The schema is SYSIBM.

string
> An expression that specifies the string from which the resulting substring is derived. The expression must return a value that is a built-in character string, graphic string, numeric, or datetime data type. A character string cannot have the FOR BIT DATA attribute (SQLSTATE 428GC). If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous string units of *string*.

start
> An expression that specifies the starting position in *string* for the beginning of the result substring in 32-bit UTF-32 string units. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If *start* is positive, then the starting position is calculated from the beginning of the string. If *start* is greater than the length of *string* in 32-bit UTF-32 string units, then a zero length string is returned.
>
> If *start* is negative, then the starting position is calculated from the end of the string by counting backwards. If the absolute value of *start* is greater than the length of *string* in 32-bit UTF-32 string units, then a zero length string is returned.
>
> If *start* is 0, then a starting position of 1 is used.

length
> An expression that specifies the length of the resulting substring in 32-bit UTF-32 string units. If *length* is specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If the value for *length* is greater than the number of 32-bit UTF-32 string units from the starting position to the end of the string, then the length of the resulting substring is the length of the first argument in 32-bit UTF-32 string units minus the starting position plus one.
>
> If the value for *length* is less than or equal to zero, the result is a zero length string.
>
> The default value for *length* is the number of CODEUNITS32 from the position specified by *start* to the last string unit of *string*.

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If *string* is a CLOB, the result of the function is a CLOB. If *string* is a GRAPHIC or VARGRAPHIC data type, the result of the function is a

VARGRAPHIC data type. If *string* is a DBCLOB, the result of the function is a DBCLOB. If the first argument is a host variable, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is based on the first applicable row in the following table. The string unit of the result is the same as the string unit of the first argument.

Table 74. Length attribute of SUBSTR4 result when arguments include constants

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
character type with length attribute A	any valid argument	constant value $L<=0$	0
character type with length attribute A	constant value S and $|(S)|>A$	not specified or any valid argument	0
character type with length attribute A	not a constant	constant value $L>0$	MIN(A, L×4) if the string units of *string* is OCTETS MIN(A, L) if the string units of *string* is CODEUNITS32
character type with length attribute A	constant value $S>0$	not specified or not a constant	A-S+1
character type with length attribute A	constant value $S<0$	not specified or not a constant	MIN(A, $|(S)×4|$) if the string units of *string* is OCTETS MIN(A, $|S|$) if the string units of *string* is CODEUNITS32
character type with length attribute A	constant value $S>0$	constant value $L>0$	MIN(A-S+1, L×4) if the string units of *string* is OCTETS MIN(A-S+1, L) if the string units of *string* is CODEUNITS32
character type with length attribute A	constant value $S<0$	constant value $L>0$	MIN(A,$|(S)×4|$, L×4) if the string units of *string* is OCTETS MIN(A, $|S|$, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	any valid argument	constant value $L<=0$	0

Table 74. Length attribute of SUBSTR4 result when arguments include constants (continued)

String Argument	Start Argument[1]	Length Argument	Length Attribute of Result[2]
graphic type with length attribute A	constant value S and $\lvert(S)\rvert>A$	not specified or any valid argument	0
graphic type with length attribute A	not a constant	constant value L>0	MIN(A, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S>0	not specified or not a constant	A-S+1
graphic type with length attribute A	constant value S<0	not specified or not a constant	MIN(A, $\lvert(S) * 2\rvert$) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, $\lvert S\rvert$) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S>0	constant value L>0	MIN(A-S+1, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A-S+1, L) if the string units of *string* is CODEUNITS32
graphic type with length attribute A	constant value S<0	constant value L>0	MIN(A, $\lvert(S)*2\rvert$, L*2) if the string units of *string* is double-bytes or CODEUNITS16 MIN(A, $\lvert S\rvert$, L) if the string units of *string* is CODEUNITS32

Notes:

[1] If a start argument value of 0 is specified, then use a value of 1 for S when referencing this table.

[2] The length attribute of the result for some of the character result types involves a constant that is multiplied by a factor of 4. This multiplier covers the worst case expansion derived from multiplying by 4 to switch from counting in 32-bit UTF-32 string units to counting in bytes used for the length attributes of a character data type with string units of OCTETS. The length attribute of the result for some of the graphic result types involves a constant that is multiplied by a factor of 2. This multiplier covers the worst case expansion derived from multiplying by 2 because a 4-byte character in UTF-32 could be represented by up to 2 double-byte characters in a graphic string.

If any argument of the SUBSTR4 function can be null, the result can be null. If any argument is null, the result is the null value.

Notes

- If *string* contains combining characters, the result might contain base characters without their combining characters or combining characters without their base characters depending on the values of start and length.
- SUBSTR4 is similar to the SUBSTR function, with the following exceptions:
  - SUBSTR4 supports a negative start value, which indicates that processing is to start from the end of the string.
  - SUBSTR4 supports a *length* value that is greater than the calculated result length. In such cases, a shorter string is returned, rather than an error.
  - SUBSTR4 returns a result data type of VARCHAR if the input data type is CHAR. VARGRAPHIC is the result data type returned if the input type is GRAPHIC.
  - The length attribute of the result for SUBSTR4 is either the same as the length attribute of the first argument, or it is derived based on the start or length attributes, if either of these are constants.
  - SUBSTR4 returns a result with a length attribute that is the same as the length attribute of the first argument, unless the start or length arguments are specified as constants. When a constant is specified, the length attribute of the result is derived based on the start or length attributes (see the preceding table).

SUBSTRB

The SUBSTRB function returns a substring of a string, beginning at a specified position in the string. Lengths are calculated in bytes.

```
►►──SUBSTRB──(──string──,──start─────────────)──────────────────────►◄
                                  └─,──length─┘
```

The schema is SYSIBM.

The SUBSTRB function is available starting with version 9.7 Fix Pack 1.

string
> An expression that specifies the string from which the result is derived.
>
> The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type it is implicitly cast to VARCHAR before evaluating the function. A substring of *string* is zero or more contiguous bytes of *string*. In a Unicode database, if the value is a graphic data type, it is implicitly cast to a character string data type before evaluating the function.

start
> An expression that specifies the start position in *string* of the beginning of the result substring. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If *start* is positive, then the start position is calculated from the beginning of the string. If *start* is greater than the length of *string*, then a zero length string is returned.
>
> If *start* is negative, then the start position is calculated from the end of the string and by counting backwards. If the absolute value of *start* is greater than the length of *string*, then a zero length string is returned.
>
> If *start* is 0, then a start position of 1 is used.

length
> An expression that specifies the length of the result in bytes. If specified, the expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.
>
> If the value of *length* is greater than the number of bytes from the start position to the end of the string, the result length is the length of the first argument minus the start position plus one.
>
> If the value of *length* is less than or equal to zero, the result of SUBSTRB is a zero length string.
>
> The default for *length* is the number of bytes from the position specified by *start* to the last byte of *string*.

If *string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. If it is a CLOB, the result of the function is a CLOB. If it is a BLOB, the result of the function is a BLOB. If the first argument is a host variable

that is not a binary string and not a FOR BIT DATA character string, the code page of the result is the section code page; otherwise, it is the code page of the first argument.

The length attribute of the result is the same as the length attribute of the first argument unless both *start* and *length* arguments are specified and defined as constants. In this case, the length attribute of the result is determined as follows:

- If *length* is a constant which is less than or equal to zero, the length attribute of the result is zero.
- If *start* is not a constant, but *length* is a constant, the length attribute of the result is the minimum of the length attribute of the first argument and *length*.
- If *start* is a constant, but *length* is not a constant or not specified, the length attribute of the result is the length attribute of the first argument minus the start position, plus one.
- If *start* and *length* are constants, the length attribute of the result is the minimum of the following values:
  - *length*
  - The length attribute of the first argument minus the start position plus one

If any argument of the SUBSTRB function can be null, the result can be null; if any argument is null, the result is the null value.

Notes

- In dynamic SQL, *string*, *start*, and *length* can be represented by a parameter marker. If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.
- Though not explicitly stated in the previously mentioned result definitions, the semantics imply that if *string* is a mixed single-byte and multi-byte character string, the result might contain fragments of multi-byte characters, depending on the values of *start* and *length*. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTRB function will detect these partial characters and will replace each byte of an incomplete character with a single blank character.
- SUBSTRB is similar to the existing SUBSTR function, with the following exceptions:
  - SUBSTRB supports a negative *start* value, which indicates the processing should start from the end of the string.
  - SUBSTRB allows *length* to be greater than the calculated result length. In this case, a shorter string will be returned, rather than returning an error.
  - Graphic input data is not natively supported for the first argument of SUBSTRB. In a Unicode database, graphic data is supported, but it is first converted to character data before evaluating the function, and lengths are calculated in bytes.
  - The result data type of SUBSTRB is VARCHAR if the input data type is CHAR.
  - The length attribute of the result for SUBSTRB is either the same as the length attribute of the first argument, or it is derived based on the *start* or *length* attributes, if either of these are constants.

Examples

- *Example 1*: Assume the host variable NAME (VARCHAR(50)) has a value of 'BLUE JAY' and the host variable SURNAME_POS (INTEGER) has a value of 6.

```
SUBSTRB(:NAME, :SURNAME_POS)
```

Returns the value 'JAY'.

```
SUBSTRB(:NAME, :SURNAME_POS,1)
```
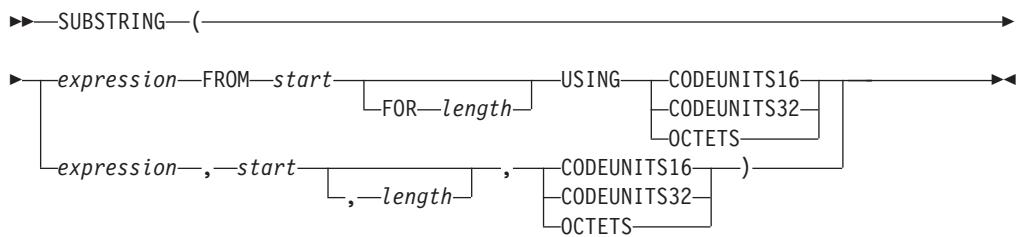
Returns the value 'J'.

- *Example 2*: Select all rows from the PROJECT table which end in 'ING'.

```
SELECT * FROM PROJECT
   WHERE SUBSTRB(PROJNAME,-3) = 'ING'
```

SUBSTRING

The SUBSTRING function returns a substring of a string.

```
►►─SUBSTRING─(─────────────────────────────────────────────────────►

►─┬─expression─FROM─start─┬────────────┬──USING─┬─CODEUNITS16─┬─────┬──►◄
  │                       └─FOR─length─┘        ├─CODEUNITS32─┤     │
  │                                             └─OCTETS──────┘     │
  └─expression──,──start─┬──────────┬──,──┬─CODEUNITS16─┬──)────────┘
                         └─,─length─┘     ├─CODEUNITS32─┤
                                          └─OCTETS──────┘
```

The schema is SYSIBM.

expression
> An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to VARCHAR before evaluating the function. 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. The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value of the integer 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, the value of the integer must be odd; otherwise, an error is returned (SQLSTATE 428GC).

length
> The expression must return a value that is a built-in numeric, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

> If *expression* is a fixed-length string, omission of *length* is an implicit specification of CHARACTER_LENGTH(*expression* USING *string-unit*) - *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 CHARACTER_LENGTH(*expression* USING *string-unit*) - *start* + 1, whichever is greater. If the required length is zero, the result is the empty string.

> If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function. The value must be greater than or equal to zero. If a value greater than *n* is specified, where *n* is the (length attribute of *expression*) - *start* + 1, then *n* is used as the length of the resulting substring. The value is expressed in the units that are explicitly specified. If OCTETS is specified, and *expression* is graphic data, the value 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 in the Examples section.)

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

- *Example 1*: FIRSTNAME is a VARCHAR(12) column in table T1. One of its values is the 6-character string 'Jürgen'. When FIRSTNAME has this value:

```
Function ...                       Returns ...
---------------------------------  ----------------------------------
SUBSTRING(FIRSTNAME,1,2,CODEUNITS32)  'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,CODEUNITS16)  'Jü' -- x'4AC3BC'
SUBSTRING(FIRSTNAME,1,2,OCTETS)       'J ' -- x'4A20' (a truncated string)
SUBSTRING(FIRSTNAME,8,CODEUNITS16)    a zero-length string
SUBSTRING(FIRSTNAME,8,4,OCTETS)       a zero-length string
```

- *Example 2*: The following example illustrates how SUBSTRING replaces the bytes of leading or trailing partial multi-byte code points with blanks when the string length unit is OCTETS. Assume that UTF8_VAR contains the UTF-8 representation of the Unicode string '&N~AB', where '&' is the musical symbol G clef and '~' is the combining tilde character.

```
SUBSTRING(UTF8_VAR, 2, 5, OCTETS)
```

Three blank bytes precede the 'N', and one blank byte follows the 'N'.

TABLE_NAME

The TABLE_NAME function returns an unqualified name of the object found after any alias chains have been resolved.

►►—TABLE_NAME—(—*object-name*———————————)——————————————►◄
                            └—,—*object-schema*—┘

The schema is SYSIBM.

The specified *object-name* (and *object-schema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

object-name
> A character expression representing the unqualified name (usually of an existing alias) to be resolved. *object-name* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

object-schema
> A character expression representing the schema used to qualify the supplied *object-name* value before resolution. *object-schema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.
>
> If *object-schema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128 OCTETS). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value. If *object-schema* is the null value, the default schema name is used. The result is the character string representing an unqualified name. The result name could represent one of the following objects:

table The value for *object-name* was either a table name (the input value is returned) or an alias name that resolved to the table whose name is returned.

view The value for *object-name* was either a view name (the input value is returned) or an alias name that resolved to the view whose name is returned.

undefined object

> The value for *object-name* was either an undefined object (the input value is returned) or an alias name that resolved to the undefined object whose name is returned.

Therefore, if a non-null value is given to this function, a value is always returned, even if no object with the result name exists.

Notes
- To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.

TABLE_SCHEMA

The TABLE_SCHEMA function returns the schema name of the object found after any alias chains have been resolved.

►►──TABLE_SCHEMA──(──*object-name*─┬──────────────────────┬──)──────────────────►◄
                                   └─,──*object-schema*──┘

The schema is SYSIBM.

The specified *object-name* (and *object-schema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

object-name
> A character expression representing the unqualified name (usually of an existing alias) to be resolved. *object-name* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

object-schema
> A character expression representing the schema used to qualify the supplied *object-name* value before resolution. *object-schema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

> If *object-schema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128 OCTETS). If *object-name* can be null, the result can be null; if *object-name* is null, the result is the null value. If *object-schema* is the null value, the default schema name is used. The result is the character string representing a schema name. The result schema could represent the schema name for one of the following objects:

table　The value for *object-name* was either a table name (the input or default value of *object-schema* is returned) or an alias name that resolved to a table for which the schema name is returned.

view　The value for *object-name* was either a view name (the input or default value of *object-schema* is returned) or an alias name that resolved to a view for which the schema name is returned.

undefined object
> The value for *object-name* was either an undefined object (the input or default value of *object-schema* is returned) or an alias name that resolved to an undefined object for which the schema name is returned.

Therefore, if a non-null *object-name* value is given to this function, a value is always returned, even if the object name with the result schema name does not exist. For example, TABLE_SCHEMA('DEPT', 'PEOPLE') returns 'PEOPLE ' if the catalog entry is not found.

Notes

- To improve performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions, the BASE_TABLE table function can be used instead.

Examples

- *Example 1*: PBIRD tries to select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A1 defined on the table HEDGES.T1.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('A1')
    AND TABSCHEMA = TABLE_SCHEMA ('A1')
  ```

  The requested statistics for HEDGES.T1 are retrieved from the catalog.

- *Example 2*: Select the statistics for an object called HEDGES.X1 from SYSCAT.TABLES using HEDGES.X1. Use TABLE_NAME and TABLE_SCHEMA since it is not known whether HEDGES.X1 is an alias or a table.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('X1','HEDGES')
    AND TABSCHEMA = TABLE_SCHEMA ('X1','HEDGES')
  ```

  Assuming that HEDGES.X1 is a table, the requested statistics for HEDGES.X1 are retrieved from the catalog.

- *Example 3*: Select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A2 defined on HEDGES.T2 where HEDGES.T2 does not exist.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('A2','PBIRD')
    AND TABSCHEMA = TABLE_SCHEMA ('A2',PBIRD')
  ```

  The statement returns 0 records as no matching entry is found in SYSCAT.TABLES where TABNAME = 'T2' and TABSCHEMA = 'HEDGES'.

- *Example 4*: Select the qualified name of each entry in SYSCAT.TABLES along with the final referenced name for any alias entry.

  ```
  SELECT TABSCHEMA AS SCHEMA,  TABNAME AS NAME,
    TABLE_SCHEMA (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_SCHEMA,
    TABLE_NAME (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_NAME
    FROM SYSCAT.TABLES
  ```

  The statement returns the qualified name for each object in the catalog and the final referenced name (after alias has been resolved) for any alias entries. For all non-alias entries, BASE_TABNAME and BASE_TABSCHEMA are null so the REAL_SCHEMA and REAL_NAME columns will contain nulls.

TAN

Returns the tangent of the argument, where the argument is an angle expressed in radians.

►►──TAN──(──*expression*──)──►◄

The schema is SYSIBM. (The SYSFUN version of the TAN function continues to be available.)

expression
An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

TANH

Returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.

►►──TANH──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of any built-in numeric data type except for DECFLOAT. The value is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

TIME

The TIME function returns a time from a value.

```
►►──TIME──(──expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIME, TIMESTAMP, or a valid character string representation of a date, time, or timestamp that is not a CLOB. In a Unicode database, if an expression returns a value of a graphic string data type, the value is first converted to a character string before the function is executed.

The result of the function is a TIME. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a DATE or string representation of a date:
  - The result is midnight.
- If the argument is a TIME:
  - The result is that time.
- If the argument is a TIMESTAMP:
  - The result is the time part of the timestamp.
- If the argument is a string representation of time or timestamp:
  - The result is the time represented by the string.

Example

Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT * FROM IN_TRAY
  WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP

The TIMESTAMP function returns a timestamp from a value or a pair of values.

```
>>--TIMESTAMP--(--expression1----------------------)----------------------------------><
                            |_,expression2_|
```

The schema is SYSIBM.

Only Unicode databases support an argument that is a graphic string representation of a date, a time, or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

expression1 **and** *expression2*

The rules for the arguments depend on whether **expression2** is specified and the data type of **expression2**.

- If only one argument is specified it must be an expression that returns a value of one of the following built-in data types: a DATE, a TIMESTAMP, or a character string that is not a CLOB. If **expression1** is a character string, it must be one of the following:
  - A valid character string representation of a date or a timestamp. For the valid formats of string representations of date or timestamp values, see "String representations of datetime values" in "Datetime values".
  - A character string with an actual length of 13 that is assumed to be a result from the GENERATE_UNIQUE function.
  - A string of length 14 that is a string of digits that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If **expression1** and **expression2** are specified:
  - If the data type of **expression2** is not an integer:
    - **expression1** must be a DATE or a valid string representation of a date and the **expression2** must be a TIME or a valid string representation of a time.
  - If the data type of **expression2** is an integer:
    - **expression1** must be a DATE, TIMESTAMP, or a valid string representation of a timestamp or date. **expression2** must be an integer constant in the range 0 to 12 representing the timestamp precision.

The result of the function is a TIMESTAMP.

The timestamp precision and other rules depend on whether the second argument is specified:

- If both arguments are specified and the second argument is not an integer:
  - The result is a TIMESTAMP(6) with the date specified by the first argument and the time specified by the second argument. The fractional seconds part of the timestamp is zero.
- If both arguments are specified and the second argument is an integer:
  - The result is a TIMESTAMP with the precision specified in the second argument.
- If only one argument is specified and it is a TIMESTAMP(*p*):

- – The result is that TIMESTAMP(*p*).
- If only one argument is specified and it is a DATE:
  - – The result is that date with an assumed time of midnight cast to TIMESTAMP(0).
- If only one argument is specified and it is a string:
  - – The result is the TIMESTAMP(6) value represented by that string extended as described earlier with any missing time information. If the argument is a string of length 14, the TIMESTAMP has a fractional seconds part of zero.

If the arguments include only date information, the time information in the result value is all zeros. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

- *Example 1*: Assume the column START_DATE (whose data type is DATE) has a value equivalent to 1988-12-25, and the column START_TIME (whose data type is TIME) has a value equivalent to 17.12.30.

  TIMESTAMP(START_DATE, START_TIME)

  Returns the value '1988-12-25-17.12.30.000000'.
- *Example 2*: Convert a timestamp string with 7 digits of fractional seconds to a TIMESTAMP(9) value.
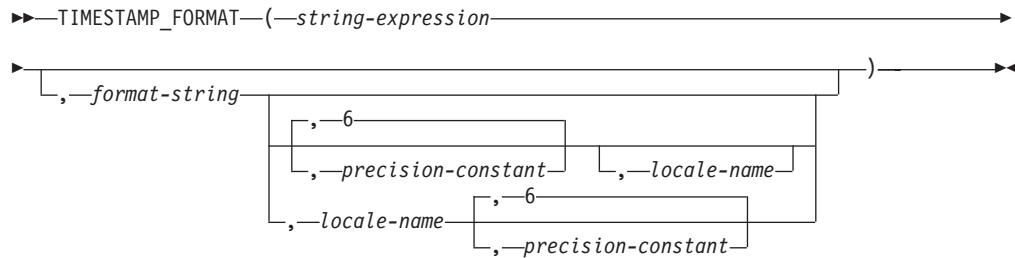
  TIMESTAMP('2007-09-24-15.53.37.2162474',9)

  Returns the value '2007-09-24-15.53.37.216247400'.

TIMESTAMP_FORMAT

The TIMESTAMP_FORMAT function returns a timestamp that is based on the interpretation of the input string using the specified format.

▶▶──TIMESTAMP_FORMAT──(──*string-expression*──▶

▶───┬──┬──)───▶◀
　　 └─,──*format-string*──┬──┬─┘
　　　　　　　　　　　　　 　├─┬─────,──6────────┬───┬──────────────────────┬─┤
　　　　　　　　　　　　　 　│ └─,──*precision-constant*─┘ └─,──*locale-name*─┘ │
　　　　　　　　　　　　　 　└─,──*locale-name*──┬────,──6────────────┬───────────┘
　　　　　　　　　　　　　　　　　　　　　　　　　 　└─,──*precision-constant*─┘

The schema is SYSIBM.

string-expression
> The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The *string-expression* must contain the components of a timestamp that correspond to the format specified by *format-string*.

format-string
> The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22007). The value is a template for how *string-expression* is interpreted and then converted to a timestamp value.

> A valid *format-string* must contain at least one format element, must not contain multiple specifications for any component of a timestamp, and can contain any combination of the format elements, unless otherwise noted in Table 75 on page 653 (SQLSTATE 22007). For example, *format-string* cannot contain both YY and YYYY, because they are both used to interpret the year component of *string-expression*. Refer to the table to determine which format elements cannot be specified together. Two format elements can optionally be separated by one or more of the following separator characters:

> - minus sign (-)
> - period (.)
> - slash (/)
> - comma (,)
> - apostrophe (')
> - semi-colon (;)
> - colon (:)
> - blank ( )

> Separator characters can also be specified at the start or end of *format-string*. These separator characters can be used in any combination in the format string, for example 'YYYY/MM-DD HH:MM.SS'. Separator characters specified in a *string-expression* are used to separate components and are not required to match the separator characters specified in the *format-string*.

Table 75. Format elements for the TIMESTAMP_FORMAT function

Format element	Related components of a timestamp	Description
AM or PM	hour	Meridian indicator (morning or evening) without periods. This format element is dependent on *locale-name*, if specified. Otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
A.M. or P.M.	hour	Meridian indicator (morning or evening) with periods. This format element uses the exact strings "A.M." or "P.M." and is independent of the locale name in effect.
DAY, Day, or day	none	Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DY, Dy, or dy	none	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
D	none	Day of the week (1-7). The first day of the week is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DD	day	Day of month (01-31).
DDD	month, day	Day of year (001-366).

TIMESTAMP_FORMAT

Table 75. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related components of a timestamp	Description
FF or FF*n*	fractional seconds	Fractional seconds (0-999999999999). The number *n* is used to specify the number of digits expected in the *string-expression*. Valid values for *n* are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. When the component in *string-expression* corresponding to the FF format element is followed by a separator character or is the last component, the number of digits for the fractional seconds can be less than what is specified by the format element. In this case zero digits are padded onto the right of the specified digits.
HH	hour	HH behaves the same as HH12.
HH12	hour	Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.
HH24	hour	Hour of the day (00-24) in 24-hour format.
J	year, month, and day	Julian day (number of days since January 1, 4713 BC).
MI	minute	Minute (00-59).
MM	month	Month (01-12).
MONTH, Month, or month	month	Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
MON, Mon, or mon	month	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.

Table 75. Format elements for the TIMESTAMP_FORMAT function (continued)

Format element	Related components of a timestamp	Description
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).
SSSSS	hours, minutes, and seconds	Seconds since previous midnight (00000-86400).
Y	year	Last digit of the year (0-9). First three digits of the current year are used to determine the full 4-digit year.
YY	year	Last two digits of the year (00-99). First two digits of the current year are used to determine the full 4-digit year.
YYY	year	Last three digits of the year (000-999). First digit of the current year is used to determine the full 4-digit year.
YYYY	year	4-digit year (0000-9999).

Note: The format elements in Table 75 on page 653 are not case sensitive, except for the following:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The DAY, Day, day, DY, Dy, dy, and D format elements do not contribute to any components of the resulting timestamp. However, a specified value for any of these format elements must be correct for the combination of the year, month, and day components of the resulting timestamp (SQLSTATE 22007). For example, assuming a value of 'en_US' is used for *locale-name*, a value of 'Monday 2008-10-06' for *string-expression* is valid for a value of 'Day YYYY-MM-DD'. However, value of 'Tuesday 2008-10-06' for *string-expression* would result in error for the same *format-string*.

TIMESTAMP_FORMAT

The RR and RRRR format elements can be used to alter how a specification for a year is to be interpreted by adjusting the value to produce a 2-digit value or a 4-digit value depending on the leftmost two digits of the current year according to the following table.

Last two digits of the current year	Two-digit year in *string-expression*	First two digits of the year component of timestamp
00-50	00-49	First two digits of the current year
51-99	00-49	First two digits of the current year + 1
00-50	50-99	First two digits of the current year - 1
51-99	50-99	First two digits of the current year

For example, if the current year is 2007, '86' with format 'RR' means 1986, but if the current year is 2052, it means 2086.

The following defaults are used when a *format-string* does not include a format element for one of the following components of a timestamp:

Timestamp component	Default
year	current year, as 4 digits
month	current month, as 2 digits
day	01 (first day of the month)
hour	00
minute	00
second	00
fractional seconds	a number of zeros matching the timestamp precision of the result

Leading zeros can be specified for any component of the timestamp value (that is, month, day, hour, minutes, seconds) that does not have the maximum number of significant digits for the corresponding format element in the *format-string*.

A substring of the *string-expression* representing a component of a timestamp (such as year, month, day, hour, minutes, seconds) can include less than the maximum number of digits for that component of the timestamp indicated by the corresponding format element. Any missing digits default to zero. For example, with a *format-string* of 'YYYY-MM-DD HH24:MI:SS', an input value of '999-3-9 5:7:2' would produce the same result as '0999-03-09 05:07:02'.

If *format-string* is not specified, *string-expression* will be interpreted using a default format based on the value of the special register CURRENT LOCALE LC_TIME.

precision-constant
An integer constant that specifies the timestamp precision of the result. The value must be in the range 0 to 12. If not specified, the timestamp precision defaults to 6.

locale-name
A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery" in the *Globalization Guide* . If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function is a TIMESTAMP with a precision based on *precision-constant*. If either of the first two arguments can be null, the result can be null; if either of the first two arguments is null, the result is the null value.

Notes

- *Julian and Gregorian calendar*: The transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.
- *Determinism:* TIMESTAMP_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of either the special register CURRENT LOCALE LC_TIME or CURRENT TIMESTAMP.
  - When *format-string* is not explicitly specified, or when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive
    - *format-string* is a constant and does not include a format element that fully defines the year (that is, J or YYYY) and so uses the value of the current year
    - *format-string* is a constant and does not include a format element that fully defines the month (for example, J, MM, MONTH, or MON) and so uses the value of the current month

  These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).
- *Syntax alternatives:* TO_DATE and TO_TIMESTAMP are synonyms for TIMESTAMP_FORMAT.

Examples

- *Example 1*: Insert a row into the IN_TRAY table with a receiving timestamp that is equal to one second before the beginning of the year 2000 (December 31, 1999 at 23:59:59).
  ```
  INSERT INTO IN_TRAY (RECEIVED)
    VALUES (TIMESTAMP_FORMAT('1999-12-31 23:59:59',
      'YYYY-MM-DD HH24:MI:SS'))
  ```
- *Example 2*: An application receives strings of date information into a variable called INDATEVAR. This value is not strictly formatted and might include two or four digits for years, and one or two digits for months and days. Date components might be separated with minus sign (-) or slash (/) characters and are expected to be in day, month, and year order. Time information consists of

hours (in 24-hour format) and minutes, and is usually separated by a colon. Sample values include '15/12/98 13:48' and '9-3-2004 8:02'. Insert such values into the IN_TRAY table.

```
INSERT INTO IN_TRAY (RECEIVED)
  VALUES (TIMESTAMP_FORMAT(:INDATEVAR,
    'DD/MM/RRRR HH24:MI'))
```

The use of RRRR in the format allows for 2- and 4-digit year values and assigns missing first two digits based on the current year. If YYYY were used, input values with a 2-digit year would have leading zeros. The slash separator also allows the minus sign character. Assuming a current year of 2007, resulting timestamps from the sample values are:

```
'15/12/98 13:48' --> 1998-12-15-13.48.00.000000
'9-3-2004 8:02'  --> 2004-03-09-08.02.00.000000
```

TIMESTAMP_ISO

Returns a timestamp value based on a date, time, or timestamp argument.

►►──TIMESTAMP_ISO──(──*expression*──)──────────────────────────────────────►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: CHAR, VARCHAR, DATE, TIME, or TIMESTAMP data type. In a Unicode database, if a supplied argument has a GRAPHIC or VARGRAPHIC data type, it is first converted to a character string before evaluating the function. A string expression must return a valid character string representation of a date or timestamp.

If the argument is a date value, TIMESTAMP_ISO inserts zero for all the time elements. If the argument is a time value, TIMESTAMP_ISO inserts the value of the CURRENT DATE special register for the date elements, and zero for the fractional seconds element. The result of the function is a TIMESTAMP(6). The result can be null; if the argument is null, the result is the null value.

The TIMESTAMP_ISO function is generally defined as deterministic. If the first argument has the TIME data type, then the function is not deterministic because the CURRENT DATE is used for the date portion of the timestamp value.

The result of the function is a TIMESTAMP. The result can be null; if the argument is null, the result is the null value.

TIMESTAMPDIFF

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

►►──TIMESTAMPDIFF──(──*numeric-expression*──,──*string-expression*──)────────────────►◄

The schema is SYSIBM. The SYSFUN version of the TIMESTAMPDIFF function continues to be available.

numeric-expression
An expression that returns a value of built-in INTEGER or SMALLINT data type. Valid values represent an interval as defined in the following table.

Table 76. Valid interval values

Value	Interval
1	Microseconds
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

string-expression
An expression that returns a value of built-in CHAR or VARCHAR data type. The value is expected to be the result of subtracting two timestamps and converting the result to CHAR. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed

If a positive or negative sign is present, it is the first character of the string. The following table describes the elements of the character string duration.

Table 77. TIMESTAMPDIFF string elements

String Elements	Valid Values	Character position from the decimal point (negative is left)
Years	1-9998 or blank	-14 to -11
Months	0-11 or blank	-10 to -9
Days	0-30 or blank	-8 to -7
Hours	0-24 or blank	-6 to -5
Minutes	0-59 or blank	-4 to -3
Seconds	0-59	-2 to -1
Decimal Points	Period	0
Microseconds	000000-999999	1 to 6

The result of the function is INTEGER with the same sign as the second argument. The result can be null; if the argument is null, the result is the null value.

The returned value is determined for each interval as indicated by the following table:

Table 78. TIMESTAMPDIFF computations

Result interval	Computation using duration elements
Years	Years
Quarters	integer value of (months+(years*12))/3
Months	months + (years*12)
Weeks	integer value of ((days+(months*30))/7)+(years*52)
Days	days + (months*30)+(years*365)
Hours	hours + ((days+(months*30)+(years*365))*24)
Minutes (the absolute value of the duration must not exceed 40850913020759.999999)	minutes + (hours+((days+(months*30)+(years*365))*24))*60
Seconds (the absolute value of the duration must be less than 680105031408.000000)	seconds + (minutes+(hours+((days+(months*30)+(years*365))*24))*60)*60
Microseconds (the absolute value of the duration must be less than 3547.483648)	microseconds + (seconds+(minutes*60))*1000000

The following assumptions may be used in estimating a difference:
- There are 365 days in a year.
- There are 30 days in a month.
- There are 24 hours in a day.
- There are 60 minutes in an hour.
- There are 60 seconds in a minute.

These assumptions are used when converting the information in the second argument, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for the difference between '1997-03-01-00.00.00' and '1997-02-01-00.00.00', the result is 30. This is because the difference between the timestamps is 1 month, and the assumption of 30 days in a month applies.

Example

The following example returns 4277, the number of minutes between two timestamps:

```
TIMESTAMPDIFF(4,CHAR(TIMESTAMP('2001-09-29-11.25.42.483219') -
  TIMESTAMP('2001-09-26-12.07.58.065497')))
```

TO_CHAR

The TO_CHAR function returns a character representation of an input expression that has been formatted using a character template.

Character to varchar

►►──TO_CHAR──(──*character-expression*──)──────────────────────────►◄

Timestamp to varchar

►►──TO_CHAR──(──*timestamp-expression*──────────────────────────────────)──────►◄
                                        └─,──*format-string*──────────────┘
                                                         └─,──*locale-name*─┘

Decimal floating-point to varchar

►►──TO_CHAR──(──*decimal-floating-point-expression*──────────────────)──────►◄
                                                    └─*format-string*─┘

The schema is SYSIBM.

The TO_CHAR scalar function is a synonym for the VARCHAR_FORMAT scalar function.

TO_CLOB

The TO_CLOB function returns a CLOB representation of a character string type.

►►──TO_CLOB──(──*character-string-expression*──┬───────────┬──)──────────────►◄
                                                        └─,──*integer*─┘
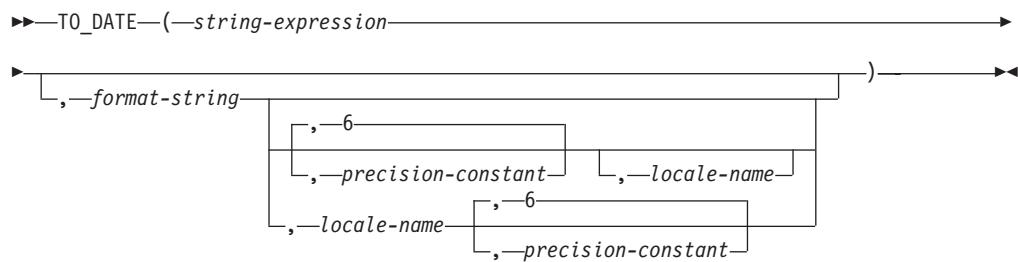
The schema is SYSIBM.

The TO_CLOB scalar function is a synonym for the CLOB scalar function.

TO_DATE

The TO_DATE function returns a timestamp that is based on the interpretation of the input string using the specified format.

```
►►──TO_DATE──(──string-expression────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────────┬──)────►◄
   └─,──format-string─┬──────────────────────────────────┬─┬──────────────┬─┘
                      │         ┌─,──6──────────┐         │ └─,──locale-name─┘
                      ├─,──precision-constant─┘            │
                      └─,──locale-name─┬─,──6──────────────┬─┘
                                       └─,──precision-constant─┘
```

The schema is SYSIBM.

The TO_DATE scalar function is a synonym for the TIMESTAMP_FORMAT scalar function.

TO_NCHAR

The TO_NCHAR function returns a national character representation of an input expression that has been formatted using a character template.

Character to nvarchar

►►──TO_NCHAR──(──*character-expression*──────────────────────────────────►◄

Timestamp to nvarchar

►►──TO_NCHAR──(──*timestamp-expression*──┬───────────────────────┬──)──►◄
                                └,──*format-string*──┬──────────────────┬┘
                                                     └,──*locale-name*──┘

Decimal floating-point to nvarchar

►►──TO_NCHAR──(──*decimal-floating-point-expression*──┬──────────────────┬──)──►◄
                                                      └,──*format-string*──┘

The schema is SYSIBM.

The TO_NCHAR function can be specified only in a Unicode database (SQLSTATE 560AA).

The TO_NCHAR scalar function is equivalent to invoking the TO_CHAR function and casting its result to NVARCHAR.

For more information about TO_NCHAR refer to VARCHAR_FORMAT.

TO_NCLOB

The TO_NCLOB function returns any type of national character string.

►►──TO_NCLOB──(──*character-string-expression*────────────────────────►◄

The schema is SYSIBM.

The TO_NCLOB scalar function is a synonym for the NCLOB scalar function.

The TO_NCLOB function can be specified only in a Unicode database (SQLSTATE 560AA).

TO_NUMBER

The TO_NUMBER function returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

```
►►──TO_NUMBER──(──string-expression──┬──────────────────┬──)──────────────►◄
                                      └─,──format-string─┘
```

The schema is SYSIBM.

The TO_NUMBER scalar function is a synonym for the DECFLOAT_FORMAT scalar function.

TO_SINGLE_BYTE

The TO_SINGLE_BYTE function returns a string in which multi-byte characters are converted to the equivalent single-byte character where an equivalent character exists.

►►──TO_SINGLE_BYTE──(──*string-expression*──)──────────────────────────────►◄

The schema is SYSIBM.

Only characters that have an equivalent to the single-byte characters represented by the characters in the UTF-8 code point range U+0020 to U+007E will be converted. If a multi-byte character does not have a single-byte equivalent, then it remains unchanged.

This function is only supported for database created in unicode and IBM-943 code page (SQLCODE 42997).

string-expression
An expression that specifies the string which gets converted. The expression must return a value that is a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The expression cannot be a character string defined as FOR BIT DATA (SQLSTATE 42815).

The data type, code page and length attribute of the result is the same as the data type, code page and length attribute of the argument. If the argument can be null, the result can be null. If the argument is null, the result is the null value.

Example
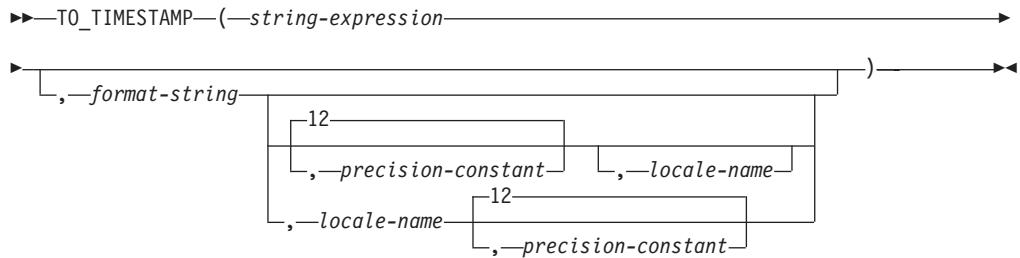
Convert the full width UTF-8 characters 'ABC' (x'efbca1efbca2efbca3' in hex format) to their single byte equivalents.

```
VALUES TO_SINGLE_BYTE(x'efbca1efbca2efbca3')
```

The result is the value 'ABC' (x'414243' in hex format).

TO_TIMESTAMP

The TO_TIMESTAMP function returns a timestamp that is based on the interpretation of the input string using the specified format.

```
►►──TO_TIMESTAMP──(──string-expression───────────────────────────────►

►──┬──────────────────────────────────────────────────────┬──)──►◄
   │                  ┌─12─┐                                │
   └─,──format-string─┴─────────────────────┬──────────────┘
                      ├─,──precision-constant─┘  ┌─,──locale-name─┐
                      │                     ┌─12─┐
                      └─,──locale-name──┬─────────────────┬─
                                        └─,──precision-constant─┘
```

The schema is SYSIBM.

The TO_TIMESTAMP scalar function is a synonym for the TIMESTAMP_FORMAT scalar function except that the default value for *precision-constant* is 12.

TOTALORDER

The TOTALORDER function returns a SMALLINT value of -1, 0, or 1 that indicates the comparison order of two arguments.

►►──TOTALORDER──(──*decfloat-expression1*──,──*decfloat-expression2*──)──────────────►◄

The schema is SYSIBM.

decfloat-expression1
> An expression that returns a value of any built-in numeric data type. If the argument is not DECFLOAT(34), it is logically converted to DECFLOAT(34) for processing.

decfloat-expression2
> An expression that returns a value of any built-in numeric data type. If the argument is not a decimal floating-point value, it is converted to DECFLOAT(34) for processing.

Numeric comparison is exact, and the result is determined for finite operands as if range and precision were unlimited. An overflow or underflow condition cannot occur.

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

The semantics of the TOTALORDER function are based on the total order predicate rules of IEEE 754R. TOTALORDER returns the following values:

- -1 if *decfloat-expression1* is lower in order compared to *decfloat-expression2*
- 0 if both *decfloat-expression1* and *decfloat-expression2* have the same order
- 1 if *decfloat-expression1* is higher in order compared to *decfloat-expression2*

The ordering of the special values and finite numbers is as follows:
```
-NAN<-SNAN<-INFINITY<-0.10<-0.100<-0<0<0.100<0.10<INFINITY<SNAN<NAN
```

The result of the function is a SMALLINT value. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

Examples

The following examples show the use of the TOTALORDER function to compare decimal floating point values:
```
TOTALORDER(-INFINITY, -INFINITY) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.0)) = 0
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-1.00)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(-0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), DECFLOAT(0.5)) = -1
TOTALORDER(DECFLOAT(-1.0), INFINITY) = -1
TOTALORDER(DECFLOAT(-1.0), SNAN) = -1
TOTALORDER(DECFLOAT(-1.0), NAN) = -1
TOTALORDER(NAN, DECFLOAT(-1.0)) = 1
TOTALORDER(-NAN, -NAN) = 0
TOTALORDER(-SNAN, -SNAN) = 0
TOTALORDER(NAN, NAN) = 0
TOTALORDER(SNAN, SNAN) = 0
TOTALORDER(-1.0, -1.0) = 0
TOTALORDER(-1.0, -1.00) = -1
```
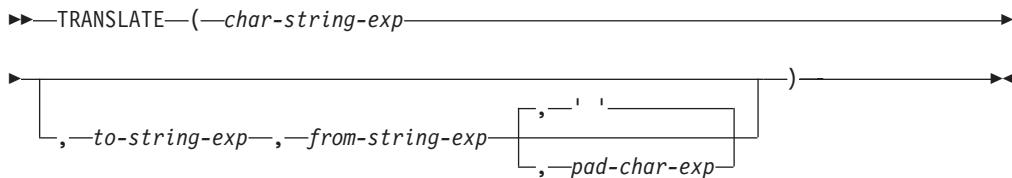
```
TOTALORDER(-1.0, -0.5) = -1
TOTALORDER(-1.0, 0.5) = -1
TOTALORDER(-1.0, INFINITY) = -1
TOTALORDER(-1.0, SNAN) = -1
TOTALORDER(-1.0, NAN) = -1
```
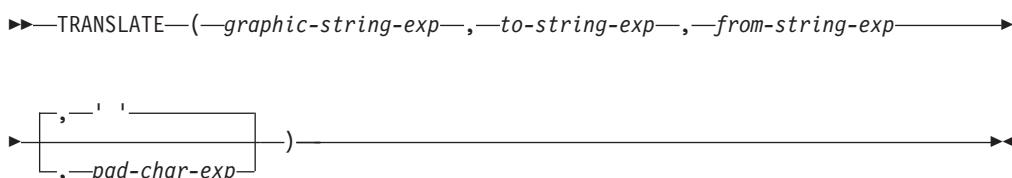
TRANSLATE

The TRANSLATE function returns a value in which one or more characters in a string expression might have been converted to other characters.

character string expression:

►►──TRANSLATE──(──*char-string-exp*──►

►──┬──┬──)──────────►◄
　　└─,──*to-string-exp*──,──*from-string-exp*──┬─────────,──' '─────────┬─┘
　　　　　　　　　　　　　　　　　　　　　　　　　　　　　└─,──*pad-char-exp*─┘

graphic string expression:

►►──TRANSLATE──(──*graphic-string-exp*──,──*to-string-exp*──,──*from-string-exp*────────►

►──┬──────────,──' '─────────┬──)──►◄
　　└─,──*pad-char-exp*─┘

The schema is SYSIBM.

The function converts all the characters in *char-string-exp* or *graphic-string-exp* that also occur in *from-string-exp* to the corresponding characters in *to-string-exp* or, if no corresponding characters exist, to the pad character specified by *pad-char-exp*.

char-string-exp **or** *graphic-string-exp*
> Specifies a string that is to be converted. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

to-string-exp
> Specifies a string of characters to which certain characters in *char-string-exp* will be converted.

> The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If a value for *to-string-exp* is not specified, and the data type is not graphic, all characters in *char-string-exp* will be in monocase; that is, the characters a-z will be converted to the characters A-Z, and other characters will be converted to their uppercase equivalents, if they exist. For example, in code page 850, é maps to É, but ÿ is not mapped, because code page 850 does not include Ÿ. If the code point length of the result character is not the same as the code point length of the source character, the source character is not converted.

from-string-exp
> Specifies a string of characters which, if found in *char-string-exp*, will be converted to the corresponding character in *to-string-exp*.

> The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not

a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. If *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If *to-string-exp* is longer than *from-string-exp*, the surplus characters will be ignored. If *to-string-exp* is specified, *from-string-exp* must also be specified.

pad-char-exp

Specifies a single character that will be used to pad *to-string-exp* if *to-string-exp* is shorter than *from-string-exp*. The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function. The value must have a length attribute of zero or one. If a zero-length string is specified, characters in the *from-string-exp* with no corresponding character in the *to-string-exp* are removed from *char-string-exp* or *graphic-string-exp*. If a value is not specified a single-byte blank character is assumed.

With *graphic-string-exp*, only *pad-char-exp* is optional (if a value is not specified, the double-byte blank character is assumed), and each argument, including the pad character, must be of a graphic data type.

The data type and code page of the result is the same as the data type and code page of the first argument. If the first argument is a host variable, the code page of the result is the database code page. 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 and string units of the result is the same as that of the first argument. If any argument can be null, the result can be null. If any argument is null, the result is the null value.

If the arguments are of data type CHAR or VARCHAR, the corresponding characters in *to-string-exp* and *from-string-exp* must have the same number of bytes (except in the case of a zero-length string). For example, it is not valid to convert a single-byte character to a multi-byte character, or to convert a multi-byte character to a single-byte character. The *pad-char-exp* argument cannot be the first byte of a valid multi-byte character (SQLSTATE 42815).

The characters are matched using a binary comparison. The database collation is not used.

If only *char-string-exp* is specified, single-byte characters will be monocased, and multi-byte characters will remain unchanged.

Examples

For the provided examples, assume that the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

- *Example 1*: The following example returns the value 'HANAUMA BAY'.

    TRANSLATE(:SITE)

- *Example 2*: The following example returns the value 'Hanauma jay'.

    TRANSLATE(:SITE, 'j', 'B')

- *Example 3*: The following example returns the value 'Heneume Bey'.

    TRANSLATE(:SITE, 'ei', 'aa')

- *Example 4*: The following example returns the value 'HAnAumA bA%'.
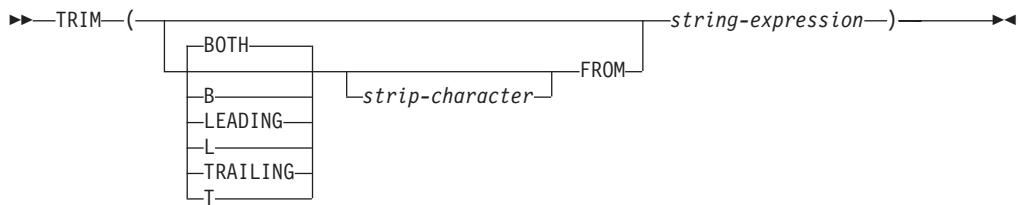
    TRANSLATE(:SITE, 'bA', 'Bay', '%')

- *Example 5*: The following example returns the value 'Hana ma ray'.

    TRANSLATE(:SITE, 'r', 'Bu')

TRIM

The TRIM function removes bytes from the beginning, from the end, or from both the beginning and end of a string expression.

```
►►─TRIM─(─┬──────────────────────────┬──string-expression─)──────────►◄
          │  ┌─BOTH────┐              │
          │  │         │    ┌─────┐   │
          └──┼─B───────┼────┤FROM ├───┘
             ├─LEADING─┤  └─strip-character─┘
             ├─L───────┤
             ├─TRAILING┤
             └─T───────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

BOTH, LEADING, or TRAILING
  Specifies whether characters are removed from the beginning, the end, or from both ends of the string expression. If this argument is not specified, the characters are removed from both the end and the beginning of the string.

strip-character
  A single-character constant that specifies the character that is to be removed. The *strip-character* can be any character whose UTF-32 encoding is a single character or a single digit numeric value. The binary representation of the character is matched.

  If *strip-character* is not specified and:
  - If the *string-expression* is a DBCS graphic string, the default *strip-character* is a DBCS blank, whose code point is dependent on the database code page
  - If the *string-expression* is a UCS-2 graphic string, the default *strip-character* is a UCS-2 blank (X'0020')
  - Otherwise, the default *strip-character* is an SBCS blank (X'20')

FROM *string-expression*
  The expression must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, numeric, or datetime data type. If the value is not a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type, it is implicitly cast to VARCHAR before evaluating the function.

The result is a varying-length string with the same maximum length as the length attribute of the *string-expression*. The actual length of the result is the length of the *string-expression* minus the number of string units that are removed. If all of the characters are removed, the result is an empty varying-length string. The code page of the result is the same as the code page of the *string-expression*.

Examples

- *Example 1*: Assume that the host variable HELLO of type CHAR(9) has a value of ' Hello'.
  ```
  SELECT TRIM(:HELLO),
    TRIM(TRAILING FROM :HELLO)
  FROM SYSIBM.SYSDUMMY1
  ```

  returns the values 'Hello' and ' Hello', respectively.
- *Example 2*: Assume that the host variable BALANCE of type CHAR(9) has a value of '000345.50'.

TRIM

```
SELECT TRIM(L '0' FROM :BALANCE),
FROM SYSIBM.SYSDUMMY1
```

returns the value '345.50'.

TRIM_ARRAY

The TRIM_ARRAY function deletes elements from the end of an array.

```
►►──┬─TRIM_ARRAY─┬──(─array-variable──,─numeric-expression──)─────────────►◄
    └─ARRAY_TRIM─┘
```

The schema is SYSIBM.

array-variable
> An SQL variable, SQL parameter, or global variable of an ordinary array type, or a CAST specification of a parameter marker to an ordinary array type. An associative array data type cannot be specified (SQLSTATE 42884).

numeric-expression
> Specifies the number of elements trimmed from the end of the array. The *numeric-expression* can be of any numeric data type with a value that can be cast to INTEGER. The value of *numeric-expression* must be between 0 and the cardinality of array-variable (SQLSTATE 2202E).

The function returns a value with the same array type as *array-variable* but with the cardinality reduced by the value of INTEGER(*numeric-expression*). The result can be null; if either argument is null, the result is the null value.

Rules

- The TRIM_ARRAY function is not supported for associative arrays (SQLSTATE 42884).
- The TRIM_ARRAY function can only be used on the right side of an assignment statement in contexts where arrays are supported (SQLSTATE 42884).

Examples

- *Example 1*: Remove the last element from the array variable RECENT_CALLS.

    SET RECENT_CALLS **= TRIM_ARRAY(**RECENT_CALLS**,** 1**)**

- *Example 2*: Assign only the first two elements from the array variable SPECIALNUMBERS to the SQL array variable EULER_CONST:
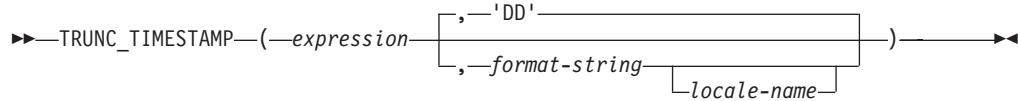
    SET EULER_CONST **= TRIM_ARRAY(**SPECIALNUMBERS**,** 8**)**

    The result is that EULER_CONST will be assigned an array with two elements, the first element value is 2.71828183 and the second element value is the null value.

TRUNC_TIMESTAMP

The TRUNC_TIMESTAMP scalar function returns a TIMESTAMP that is an argument (*expression*) truncated to the unit specified by another argument (*format-string*).

```
►►──TRUNC_TIMESTAMP──(──expression──┬──,──'DD'──────────────────┬──)────►◄
                                    └──,──format-string──┬──────────────┘
                                                         └──locale-name──┘
```

The schema is SYSIBM.

If *format-string* is not specified, *expression* is truncated to the nearest day, as if 'DD' was specified for *format-string*.

expression
> An expression that returns a value of one of the following built-in data types: a DATE or a TIMESTAMP.

format-string
> An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in *format-string* specifies how *expression* should be truncated. For example, if *format-string* is 'DD', a timestamp that is represented by *expression* is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for a timestamp (SQLSTATE 22007). The default is 'DD'.

> Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
> A character constant that specifies the locale used to determine the first day of the week when using format model values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

> The result of the function is a TIMESTAMP with the same timestamp precision as *expression*. The result can be null; if any argument is null, the result is the null value.

The result of the function is a TIMESTAMP with a timestamp precision of:
- *p* when the data type of expression is TIMESTAMP(*p*)
- 0 when the data type of expression is DATE
- 6 otherwise

The result can be null; if any argument is null, the result is the null value.

Notes
- **Determinism:** TRUNC_TIMESTAMP is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a date or timestamp value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant

- *format-string* is a constant and includes format elements that are locale sensitive

Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Example

Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.

```
SET :TRNK_TMSTMP = TRUNC_TIMESTAMP('2000-03-14-17.30.00', 'YEAR');
```

The host variable TRNK_TMSTMP is set with the value 2000-01-01-00.00.00.000000.

TRUNCATE or TRUNC

The TRUNCATE function returns a truncated value of a number or a datetime value.

TRUNCATE numeric:

```
►►─┬─TRUNCATE─┬─(─numeric-expression1─┬─────────,──0──────────┬─)─────►◄
   └─TRUNC────┘                       └─,──numeric-expression2─┘
```

TRUNCATE datetime:

```
►►─┬─TRUNCATE─┬─(─datetime-expression─┬──────────,──'DD'─────────┬─)────►◄
   └─TRUNC────┘                       └─,──format-string──────────┘
                                                    └─locale-name─┘
```

The schema is SYSIBM. The SYSFUN version of the TRUNCATE numeric function continues to be available.

The data type of the return value depends on the first argument:
- If the result of the first argument is a numeric value, a number is returned, truncated to the specified number of places to the right or left of the decimal point.
- If the first argument is a DATE, TIME, or TIMESTAMP, a datetime value, truncated to the unit specified by *format-string*.

TRUNCATE numeric

> If *numeric-expression1* has a numeric data type, the TRUNCATE function returns *numeric-expression1* truncated to *numeric-expression2* places to the right of the decimal point if *numeric-expression2* is positive, or to the left of the decimal point if *numeric-expression2* is zero or negative. If *numeric-expression2* is not specified, *numeric-expression1* is truncated to zero places left of the decimal point.

> *numeric-expression1*
>> An expression that must return a value that is a built-in CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, or numeric data type. If the value is not a numeric data type, it is implicitly cast to DECFLOAT(34) before evaluating the function.

>> If the expression is a decimal floating-point data type, the DECFLOAT rounding mode will not be used. The rounding behavior of TRUNCATE corresponds to a value of ROUND_DOWN. If a different rounding behavior is wanted, use the QUANTIZE function.

> *numeric-expression2*
>> An expression that returns a value that is a built-in numeric data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before evaluating the function.

>> If *numeric-expression2* is not negative, *numeric-expression1* is truncated to the absolute value of *numeric-expression2* number of places to the right of the decimal point.

>> If *numeric-expression2* is negative, *numeric-expression1* is truncated to the absolute value of *numeric-expression2* + 1 number of places to the left of

the decimal point. If the absolute value of a negative *numeric-expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example:

```
TRUNCATE(748.58,-4) = 0
```

The data type, length, and scale attributes of the result are the same as the data type, length, and scale attributes of the first argument.

If either argument can be null, the result can be null. If either argument is null, the result is the null value.

TRUNCATE datetime

If *datetime-expression* has a datetime data type, the TRUNCATE function returns *datetime-expression* rounded to the unit specified by the *format-string*. If *format-string* is not specified, *datetime-expression* is truncated to the nearest day, as if 'DD' is specified for *format-string*.

datetime-expression
An expression that must return a value that is a DATE, a TIME, or a TIMESTAMP. String representations of these data types are not supported and must be explicitly cast to a DATE, TIME, or TIMESTAMP for use with this function; alternatively, you can use the TRUNC_TIMESTAMP function for a string representation of a date or timestamp.

format-string
An expression that returns a built-in character string data type with an actual length that is not greater than 254 bytes. The format element in *format-string* specifies how *datetime-expression* should be truncated. For example, if *format-string* is 'DD', a timestamp that is represented by *datetime-expression* is truncated to the nearest day. Leading and trailing blanks are removed from the string, and the resulting substring must be a valid format element for the type of *datetime-expression* (SQLSTATE 22007). The default is 'DD', which cannot be used if the data type of *datetime-expression* is TIME.

Allowable values for *format-string* are listed in the table of format elements found in the description of the ROUND function.

locale-name
A character constant that specifies the locale used to determine the first day of the week when using format element values DAY, DY, or D. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery". If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result of the function has the same date type as *datetime-expression*. The result can be null; if any argument is null, the result is the null value.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

The result can be null if the argument can be null or if the argument is not a decimal floating-point number and the database is configured with **dft_sqlmathwarn** set to YES; the result is the null value if the argument is null.

Notes

- **Determinism:** TRUNCATE is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Truncate of a datetime value when *locale-name* is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

  Invocations of the function that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).

Examples

- *Example 1*: Using the EMPLOYEE table, calculate the average monthly salary for the highest paid employee. Truncate the result two places to the right of the decimal point.

  ```
  SELECT TRUNCATE(MAX(SALARY)/12,2)
    FROM EMPLOYEE;
  ```

  Assuming the highest paid employee earns $52750.00 per year, the example returns 4395.83.

- *Example 2*: Display the number 873.726 truncated 2, 1, 0, -1, and -2 decimal places, respectively.

  ```
  VALUES (
    TRUNCATE(873.726,2),
    TRUNCATE(873.726,1),
    TRUNCATE(873.726,0),
    TRUNCATE(873.726,-1),
    TRUNCATE(873.726,-2),
    TRUNCATE(873.726,-3) );
  ```

  This example returns 873.720, 873.700, 873.000, 870.000, 800.000, and 0.000.

- *Example 3*: Display the decimal-floating point number 873.726 truncated 0 decimal places.

  ```
  VALUES(TRUNCATE(DECFLOAT(873.726),0))
  ```

  Returns the value 873.

- *Example 4*: Set the variable vTRNK_DT with the input date rounded to the nearest month value.

  ```
  SET vTRNK_DT = TRUNC(DATE('2000-08-16'), 'MONTH');
  ```

  The value set is 2000-08-01.

- *Example 5*: Set the host variable TRNK_TMSTMP with the current year rounded to the nearest year value.

  ```
  SET :TRNK_TMSTMP = TRUNCATE('2000-03-14-17.30.00'), 'YEAR');
  ```

  The value set is 2000-01-01-00.00.00.000000.

TYPE_ID

The TYPE_ID function returns the internal type identifier of the dynamic data type of the *expression*.

►►—TYPE_ID—(—*expression*—)————————————————————————————►◄

The schema is SYSIBM.

expression
   An expression that returns a value of a user-defined structured data type.

The data type of the result of the function is INTEGER. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

The value returned by the TYPE_ID function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the TYPE_SCHEMA and TYPE_NAME functions to determine the type schema and type name.

Notes
- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

Example

A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the internal type identifier of the row that corresponds to the reference.

```
SELECT TASK, WHO_RESPONSIBLE->NAME,
    TYPE_ID(DEREF(WHO_RESPONSIBLE))
  FROM ACTIVITIES
```

The DEREF function is used to return the object corresponding to the row.

TYPE_NAME

The TYPE_NAME function returns the unqualified name of the dynamic data type of the *expression*.

►►──TYPE_NAME──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of a user-defined structured data type.

The data type of the result of the function is VARCHAR(128 OCTETS). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_SCHEMA function to determine the schema name of the type name returned by TYPE_NAME.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

Example

A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the type of the row that corresponds to the reference.

```
   SELECT TASK, WHO_RESPONSIBLE->NAME,
      TYPE_NAME(DEREF(WHO_RESPONSIBLE)),
      TYPE_SCHEMA(DEREF(WHO_RESPONSIBLE))
    FROM ACTIVITIES
```

The DEREF function is used to return the object corresponding to the row.

TYPE_SCHEMA

The TYPE_SCHEMA function returns the schema name of the dynamic data type of the *expression*.

►►──TYPE_SCHEMA──(──*expression*──)──────────────────────────────────►◄

The schema is SYSIBM.

expression
   An expression that returns a value of a user-defined structured data type.

The data type of the result of the function is VARCHAR(128 OCTETS). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_NAME function to determine the type name associated with the schema name returned by TYPE_SCHEMA.

Notes

- This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

UCASE

The UCASE function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.

>>—UCASE—(—*expression*—)——><

The schema is SYSIBM.

UCASE is a synonym for UPPER.

UCASE (locale sensitive)

The UCASE function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

```
►►──UCASE──(──string-expression──,──locale-name──┬─────────────────┬──┬──────────────────────┬──)───────►◄
                                                  └─,──code-units──┘  └─,──┬─CODEUNITS16─┬──┘
                                                                           ├─CODEUNITS32─┤
                                                                           └─OCTETS──────┘
```

The schema is SYSIBM.

UCASE is a synonym for UPPER.

UPPER

The UPPER function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.

►►──UPPER──(──*expression*──)──►◄
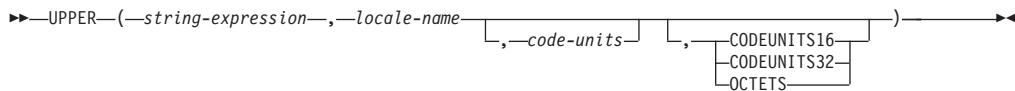
The schema is SYSIBM. (The SYSFUN version of this function continues to be available for upward compatibility.)

expression
An expression that returns a value of built-in CHAR or VARCHAR data type that is not FOR BIT DATA. 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.

UPPER (locale sensitive)

The UPPER function returns a string in which all characters have been converted to uppercase characters using the rules associated with the specified locale.

```
►►──UPPER──(──string-expression──,──locale-name─────────────────────────────────────────)────►◄
                                              └─,──code-units─┘  └─,──┬─CODEUNITS16─┬─┘
                                                                      ├─CODEUNITS32─┤
                                                                      └─OCTETS──────┘
```

The schema is SYSIBM.

string-expression
> An expression that returns a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC string. If *string-expression* is CHAR or VARCHAR, the expression must not be FOR BIT DATA (SQLSTATE 42815).

locale-name
> A character constant that specifies the locale that defines the rules for conversion to uppercase characters. The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming, see "Locale names for SQL and XQuery".

code-units
> An integer constant that specifies the number of code units in the result. If specified, *code-units* must be an integer between:
> * 1 and 32 672 if the string unit of the result is OCTETS
> * 1 and 16 336 if the string unit of the result is double bytes or CODEUNITS16
> * 1 and 8 168 if the string unit of the result is CODEUNITS32
>
> otherwise an error (SQLSTATE 42815). If *code-units* is not explicitly specified, it is implicitly the length attribute of *string-expression*. If OCTETS is specified and the result is graphic data, the value of *code-units* must be even, otherwise an error is returned (SQLSTATE 428GC). If OCTETS is specified and the string units of the result is CODEUNIT32, the value of *code-units* must be a multiple of 4 (SQLSTATE 428GC). If CODEUNITS16 is specified and the string units of the result is CODEUNITS32, the value of *code-units* must be a multiple of 2 (SQLSTATE 428GC).

CODEUNITS16, CODEUNITS32, or OCTETS
> Specifies the string unit of *code-units*.
>
> CODEUNITS16 specifies that *code-units* is expressed in 16-bit UTF-16 code units. CODEUNITS32 specifies that *code-units* is expressed in 32-bit UTF-32 code units. OCTETS specifies that *code-units* is expressed in bytes.
>
> If a string unit is not explicitly specified, the string unit of *string-expression* determines the unit that is used. For more information about CODEUNITS16, CODEUNITS32, and OCTETS, see "String units in built-in functions" in "Character strings".

The result of the function is VARCHAR if *string-expression* is CHAR or VARCHAR, and VARGRAPHIC if *string-expression* is GRAPHIC or VARGRAPHIC. The string units of the result is the same as the string units of *string-expression*

The length attribute of the result is determined by the implicit or explicit value of *code-units*, the implicit or explicit string unit, the result data type, and the result string units, as shown in the following table:

UPPER (locale sensitive)

Table 79. Length attribute of the result of UPPER as a function of string unit and result type

Data type and string units of result	Length attribute for *code-units* in **CODEUNITS16**	Length attribute for *code-units* in **CODEUNITS32**	Length attribute for *code-units* in **OCTETS**
VARCHAR in OCTETS	MIN(*code-units* * 3, 32672)	MIN(*code-units* * 4, 32672)	*code-units*
VARCHAR in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)
VARGRAPHIC in CODEUNITS16 or double bytes	*code-units*	MIN(*code-units* * 2, 16336)	MIN(*code-units* / 2, 16336)
VARGRAPHIC in CODEUNITS32	MIN(*code-units* / 2, 8168)	MIN(*code-units*, 8168)	MIN(*code-units* / 4, 8168)

The actual length of the result might be greater than the length of *string-expression*. If the actual length of the result is greater than the length attribute of the result, an error is returned (SQLSTATE 42815). If the number of code units in the result exceeds the value of *code-units*, an error is returned (SQLSTATE 42815).

If *string-expression* is not in UTF-16, this function performs code page conversion of *string-expression* to UTF-16, and of the result from UTF-16 to the code page of *string-expression*. If either code page conversion results in at least one substitution character, the result includes the substitution character, a warning is returned (SQLSTATE 01517), and the warning flag SQLWARN8 in the SQLCA is set to 'W'.

If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in uppercase characters.

  ```
  SELECT UPPER(JOB, 'en_US')
    FROM EMPLOYEE
    WHERE EMPNO = '000020'
  ```

  The result is the value 'MANAGER'.

- *Example 2*: Find the uppercase characters for all the 'I' characters in a Turkish string.

  ```
  VALUES UPPER('I△ıi', 'tr_TR', CODEUNITS16)
  ```

  The result is the string 'I△I△'.

- *Example 3*: Find the uppercase form of the German 'ß' (sharp S).

  ```
  VALUES UPPER('ß', 'de', 2, CODEUNITS16)
  ```

  The result is the string 'SS'. Note that *code-units* must be specified in this example, because there are more code units in the result than in *string-expression*.

VALUE

The VALUE function returns the first argument that is not null.

```
►►──VALUE──(──expression──┬──,expression──┬──)─────────────────────────►◄
```

The schema is SYSIBM.

VALUE is a synonym for COALESCE.

VARCHAR

The VARCHAR function returns a varying-length character string representation of
a variety of data types.

Integer to varchar

▶▶──VARCHAR──(──*integer-expression*──)────────────────────────────────▶◀

Decimal to varchar

▶▶──VARCHAR──(──*decimal-expression*─┬──────────────────────┬──)──────────▶◀
                                     └─,──*decimal-character*─┘

Floating-point to varchar

▶▶──VARCHAR──(──*floating-point-expression*─┬──────────────────────┬──)────▶◀
                                            └─,──*decimal-character*─┘

Decimal floating-point to varchar

▶▶──VARCHAR──(──*decimal-floating-point-expression*─┬──────────────────────┬──)──▶◀
                                                    └─,──*decimal-character*─┘

Character to varchar

▶▶──VARCHAR──(──*character-expression*─┬─────────────┬──)─────────────────▶◀
                                       └─,──*integer*─┘

Graphic to varchar

▶▶──VARCHAR──(──*graphic-expression*─┬─────────────┬──)───────────────────▶◀
                                     └─,──*integer*─┘

Datetime to varchar

▶▶──VARCHAR──(──*datetime-expression*─┬──────────────┬──)─────────────────▶◀
                                      └─,─┬──ISO───┬──┘
                                          ├──USA───┤
                                          ├──EUR───┤
                                          ├──JIS───┤
                                          └──LOCAL─┘

The schema is SYSIBM. The function name cannot be specified as a qualified name
when keywords are used in the function signature.

The VARCHAR function returns a varying-length character string representation
of:
- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number

- A double-precision floating-point number, if the first argument is a DOUBLE or REAL
- A decimal floating-point number, if the first argument is a (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the first argument is a DATE, TIME, or TIMESTAMP

In a non-Unicode database, the string units of the result is OCTETS. Otherwise, the string units of the result is determined by the data type of the first argument.
- OCTETS, if the first argument is character string or a graphic string with string units of OCTETS, CODEUNITS16, or double bytes.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:
- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors, because they might change in a future release.

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.

Integer to varchar

> *integer-expression*
> An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).
>
> The result is the varying-length string representation of *integer-expression* in the form of an SQL integer constant. The length attribute of the result depends on whether *integer-expression* is a small, large or big integer as follows:
> - If the first argument is a small integer, the maximum length of the result is 6.
> - If the first argument is a large integer, the maximum length of the result is 11.
> - If the first argument is a big integer, the maximum length of the result is 20.
>
> The actual length of the result is the smallest number of characters that can be used to represent the value of the argument. Leading zeros are not included. If the argument is negative, the first character of the result is a minus sign. Otherwise, the first character is a digit.
>
> The code page of the result is the code page of the section.

Decimal to varchar

decimal-expression
> An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

decimal-character
> Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *decimal-expression* in the form of an SQL decimal constant. The length attribute of the result is 2+p, where p is the precision of *decimal-expression*. The actual length of the result is the smallest number of characters that can be used to represent the result, except that trailing zeros are included. Leading zeros are not included. If decimal-expression is negative, the first character of the result is a minus sign; otherwise, the first character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned.

The code page of the result is the code page of the section.

Floating-point to varchar

floating-point-expression
> An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
> Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *floating-point-expression* in the form of an SQL floating-point constant.

The maximum length of the result is 24. The actual length of the result is the smallest number of characters that can represent the value of floating-point-expression such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If *floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *floating-point-expression* is zero, the result is 0E0.

The code page of the result is the code page of the section.

Decimal floating-point to varchar

decimal-floating-point-expression
> An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
> Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length character string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The maximum length of the result is 42. The actual length of the result is the smallest number of characters that can represent the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings "INFINITY", "SNAN", and "NAN", respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string.

The code page of the result is the code page of the section.

Character to varchar

character-expression
> An expression that returns a value that is a built-in character string data type.

integer
> An integer constant that specifies the length attribute for the resulting varying-length character string. The value must be between 0 and the maximum length for the VARCHAR data type in the string units of the result.
>
> If the second argument is not specified:
> * If the *character-expression* is the empty string constant, the length attribute of the result is 0.
> * Otherwise, the length attribute of the result is the lower of the following values:
>   – The maximum length for the VARCHAR data type in the string units of the result
>   – The length attribute of the first argument

The result is a varying-length character string. The length attribute of the result is determined by the value of *integer*. If *character-expression* is the FOR BIT DATA subtype, the result is FOR BIT DATA.

If the length of *character-expression* is greater than the length attribute of the result, several scenarios exist:
* If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
* If *integer* is specified, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). When part of a multi-byte character is truncated, that partial character is replaced with the blank character. Do not rely on this behavior because it might change in a future release.
* If *integer* is not specified, an error is returned (SQLSTATE 22001).

Graphic to varchar

graphic-expression
> An expression that returns a value that is a built-in graphic string data
> type.

integer
> An integer constant that specifies the length attribute for the resulting
> varying-length character string. The value must be between 0 and the
> maximum length for the VARCHAR data type in the string units of the
> result.
>
> If the second argument is not specified:
>
> - If the *graphic-expression* is the empty string constant, the length
>   attribute of the result is 0.
> - If the string units of *graphic-expression* is CODEUNITS32, the length
>   attribute of the result is the lower of the following values:
>   – The maximum length for the VARCHAR data type in the string
>   units of the result
>   – The length attribute of the first argument
> - Otherwise, the length attribute of the result is the lower of the
>   following values:
>   – The maximum length for the VARCHAR data type in the string
>   units of the result
>   – 3 * length attribute of the first argument

The result is a varying-length character string that is converted from
graphic-expression. The length attribute of the result is determined by the
value of *integer*.

If the length of *graphic-expression* that is converted to a character string is
greater than the length attribute of the result, several scenarios exist:

- If the string unit of the result is CODEUNITS32, truncation is performed.
  If only blank characters are truncated and *graphic-expression* is GRAPHIC
  or VARGRAPHIC, no warning is returned. Otherwise, a warning is
  returned (SQLSTATE 01004).
- If *integer* is specified and *graphic-expression* is a GRAPHIC or
  VARGRAPHIC, truncation is performed with no warning returned.
- If *integer* is specified and *graphic-expression* is a DBCLOB, truncation is
  performed with a warning returned (SQLSTATE 01004).
- If *integer* is not specified and *graphic-expression* is a GRAPHIC or
  VARGRAPHIC, truncation is performed with no warning returned.
- If *integer* is not specified and *graphic-expression* is a DBCLOB, an error is
  returned (SQLSTATE 22001).

Datetime to varchar

datetime-expression
> An expression that is of one of the following data types:
>
> **DATE** The result is the character string representation of the date in
> the format specified by the second argument. The length of the
> result is 10. An error is returned if the second argument is
> specified and is not a valid value (SQLSTATE 42703).
>
> **TIME** The result is the character string representation of the time in
> the format specified by the second argument. The length of the

result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP

The result is the character string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the code page of the section.

Notes

- The CAST specification should be used to increase the portability of applications when the first argument is numeric, or the first argument is a string and the length argument is specified.
- A binary string is allowed as the first argument to the function, and the resulting varying-length string is a FOR BIT DATA character string.

Examples

- *Example 1:* Make EMPNO varying-length with a length of 10.

  ```
  SELECT VARCHAR(EMPNO,10)
    INTO :VARHV
    FROM EMPLOYEE
  ```

- *Example 2:* Set the host variable JOB_DESC, defined as VARCHAR(8), to the VARCHAR equivalent of the job description (which is the value of the JOB column), defined as CHAR(8), for employee Dolores Quintana.

  ```
  SELECT VARCHAR(JOB)
    INTO :JOB_DESC
    FROM EMPLOYEE
    WHERE LASTNAME = 'QUINTANA'
  ```

- *Example 3:* The EDLEVEL column is defined as SMALLINT. The following returns the value as a varying-length character string.

  ```
  SELECT VARCHAR(EDLEVEL)
    FROM EMPLOYEE
    WHERE LASTNAME = 'HAAS'
  ```

  Results in the value '18'.

- *Example 4:* The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

  ```
  SELECT VARCHAR(SALARY + COMM, ',')
    FROM EMPLOYEE
    WHERE LASTNAME = 'HAAS'
  ```

  Results in the value '56970,00'.

VARCHAR_BIT_FORMAT

The VARCHAR_BIT_FORMAT function returns a bit string representation of a character string that has been formatted using a character template.

►►──VARCHAR_BIT_FORMAT──(─*character-expression*───────────────)──────────►◄
                                            └─,*format-string*─┘

The schema is SYSIBM.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

character-expression
    An expression that returns a value that is a built-in character string that is not a CLOB (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted. If a *format-string* argument is not specified, the length must be an even number of characters from the ranges '0' to '9', 'a' to 'f', and 'A' to 'F' (SQLSTATE 42815).

format-string
    A character constant that contains a template for how the bytes of *character-expression* are to be interpreted.

    Valid format strings include: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx' and 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX' (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit in the result.

The result of the function is a varying-length character string FOR BIT DATA with the length attribute and actual length based on the format string. For the two valid format strings listed previously, the length attribute of the result is 36 and the actual length is 16. If a *format-string* argument is not specified, the length attribute of the result is half the length attribute of *character-expression* and the actual length is half the length of the actual length of *character-expression*. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

HEXTORAW(*character-expression*) is equivalent to VARCHAR_BIT_FORMAT(*character-expression*).

Examples

- *Example 1*: Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('d83d6360-1818-11db-9804-b622a1ef5492',
                   'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

  Result returned:

```
 x'D83D6360181811DB9804B622A1EF5492'
```

- *Example 2*: Represent a Universal Unique Identifier in its binary form:

```
VARCHAR_BIT_FORMAT('D83D6360-1818-11DB-9804-B622A1EF5492',
                   'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

  Result returned:

```
 x'D83D6360181811DB9804B622A1EF5492'
```

- *Example 3*: Represent a string of hexadecimal characters in binary form.

```
VARCHAR_BIT_FORMAT('ef01abC9')
```

Result returned as a VARCHAR(4) FOR BIT DATA value:

```
x'EF01ABC9'
```

- *Example 4*: Represent a string of hexadecimal characters as a character string in the code page of the database. The result needs to be cast to a VARCHAR data type with the FOR MIXED DATA clause, assuming the database supports graphic types. Otherwise the result needs to be cast to a VARCHAR data type with the FOR SBCS clause. The following example assumes a Unicode database:

VALUES CAST(VARCHAR_BIT_FORMAT(HEX('abcdefg'**)) AS VARCHAR(**10**) FOR MIXED DATA)**

Result returned:

```
abcdefg
```

VARCHAR_FORMAT

The VARCHAR_FORMAT function returns a character string based on applying the specified format string argument, if provided, to the value of the first argument.

Character to varchar

►►──VARCHAR_FORMAT──(──*character-expression*──)────────────────────◄

Timestamp to varchar

►►──VARCHAR_FORMAT──(──*timestamp-expression*─────────────────────────)────◄
                 └─,──*format-string*───────────────┘
                           └─,──*locale-name*─┘

Decimal floating-point to varchar

►►──VARCHAR_FORMAT──(──*decimal-floating-point-expression*────────)────◄
                            └─,──*format-string*─┘

The schema is SYSIBM.

If any argument of the VARCHAR_FORMAT function can be null, the result can be null; if any argument is null, the result is the null value.

The expression must be formatted according to a specified character template.

Character to varchar

> *character-expression*
>> An expression that returns a value that must be a built-in CHAR or VARCHAR data type. In a Unicode database, if a supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function.
>>
>> The result is a VARCHAR with a length attribute that matches the length attribute of the argument. The value of the result is the same as the value of *character-expression*.

Timestamp to varchar

> *timestamp-expression*
>> An expression that returns a value that must be a DATE or TIMESTAMP, or a valid string representation of a date or timestamp that is not a CLOB or DBCLOB. If the argument is a string, the *format-string* argument must also be specified. In a Unicode database, if a supplied argument is a graphic string representation of a data, time, or timestamp, it is first converted to a character string before evaluating the function.
>>
>> If *timestamp-expression* is a DATE or a valid string representation of a date, it is first converted to a TIMESTAMP(0) value, assuming a time of exactly midnight (00.00.00). If *timestamp-expression* is a valid string representation of a timestamp, it is first converted to a TIMESTAMP(12) value.

For the valid formats of string representations of datetime values, see "String representations of datetime values" in "Datetime values".

format-string
: The expression must return a value that is a built-in CHAR, VARCHAR, numeric, or datetime data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if the supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22007). The value is a template for how *timestamp-expression* is to be formatted.

A valid *format-string* must contain a combination of the format elements listed in Table 1 (SQLSTATE 22007). Two format elements can optionally be separated by one or more of the following separator characters:

- minus sign (-)
- period (.)
- slash (/)
- comma (,)
- apostrophe (')
- semi-colon (;)
- colon (:)
- blank ( )

Separator characters can also be specified at the start or end of *format-string*.

Table 80. Format elements for the VARCHAR_FORMAT function

Format element	Description
AM or PM	Meridian indicator (morning or evening) without periods. This format element is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
A.M. or P.M.	Meridian indicator (morning or evening) with periods. This format element uses the exact strings 'A.M.' or 'P.M.' and is independent of the locale name in effect.
CC	Century (01-99). If the last two digits of the four-digit year are zero, the result is the first two digits of the year; otherwise, the result is the first two digits of the year plus one.
DAY, Day, or day	Name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.

VARCHAR_FORMAT

Table 80. Format elements for the VARCHAR_FORMAT function (continued)

Format element	Description
DY, Dy, or dy	Abbreviated name of the day in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
D	Day of the week (1-7). The first day of the week is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
DD	Day of month (01-31).
DDD	Day of year (001-366).
FF or FF*n*	Fractional seconds (0-999999999999). The number *n* is used to specify the number of digits to include in the returned value. Valid values for *n* are 1-12 with no leading zeros. Specifying FF is equivalent to specifying FF6. If the timestamp precision of *timestamp-expression* is less than what is specified by the format, zero digits are padded onto the right of the specified digits.
HH	HH behaves the same as HH12.
HH12	Hour of the day (01-12) in 12-hour format.
HH24	Hour of the day (00-24) in 24-hour format.
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.
IYYY	ISO year (0000-9999). The 4-digit year based on the ISO week that is returned.
J	Julian day (number of days since January 1, 4713 BC).
MI	Minute (00-59).
MM	Month (01-12).
MONTH, Month, or month	Name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.

Table 80. Format elements for the VARCHAR_FORMAT function (continued)

Format element	Description
MON, Mon, or mon	Abbreviated name of the month in uppercase, titlecase, or lowercase format. The language used is dependent on *locale-name*, if specified; otherwise, it is dependent on the value of the special register CURRENT LOCALE LC_TIME.
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).

Note: The format elements in Table 80 on page 701 are not case sensitive, except for the following:

- AM, PM
- A.M., P.M.
- DAY, Day, day
- DY, Dy, dy
- D
- MONTH, Month, month
- MON, Mon, mon

In cases where format elements are ambiguous, the case insensitive format elements will be considered first. For example, DDYYYY would be interpreted as DD followed by YYYY, rather than D followed by DY, followed by YYY.

If *format-string* is not specified, the format used is based on the value of the special register CURRENT LOCALE LC_TIME.

locale-name
A character constant that specifies the locale used for the following format elements:

- AM, PM
- DAY, Day, day
- DY, Dy, dy

VARCHAR_FORMAT

- D
- MONTH, Month, month
- MON, Mon, mon

The value of *locale-name* is not case sensitive and must be a valid locale (SQLSTATE 42815). For information about valid locales and their naming,, see "Locale names for SQL and XQuery" in the *Globalization Guide* . If *locale-name* is not specified, the value of the special register CURRENT LOCALE LC_TIME is used.

The result 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 80 on page 701. 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 254. If the string units of the environment or *format-string* is CODEUNITS32, the string units of the result is CODEUNITS32. Otherwise, the string units of the result is OCTETS. The *format-string* determines the actual length of the result. If the resulting string exceeds the length attribute of the result, the result is truncated.

Decimal floating-point to varchar

decimal-floating-point-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.

format-string
> The expression must return a value that is a built-in CHAR, VARCHAR, numeric, or datetime data type. If the value is not a CHAR or VARCHAR data type, it is implicitly cast to VARCHAR before evaluating the function. In a Unicode database, if the supplied argument is a GRAPHIC or VARGRAPHIC data type, it is first converted to VARCHAR before evaluating the function. The actual length must not be greater than 254 bytes (SQLSTATE 22018). The value is a template for how *decimal-floating-point-expression* is to be formatted. Format elements specified as a prefix can be used only at the beginning of the template. Format elements specified as a suffix can be used only at the end of the template. The format elements are case sensitive. The template must not contain more than one of the MI, S, or PR format elements (SQLSTATE 22018).

Table 81. Format elements for the VARCHAR_FORMAT function

Format element	Description
0	Each 0 represents a significant digit. Leading zeros in a number are displayed as zeros.
9	Each 9 represents a significant digit. Leading zeros in a number are displayed as blanks.

Table 81. Format elements for the VARCHAR_FORMAT function (continued)

Format element	Description
MI	Suffix: If *decimal-floating-point-expression* is a negative number, a trailing minus sign (-) is included in the result. If *decimal-floating-point-expression* is a positive number, a trailing blank is included in the result.
S	Prefix: If *decimal-floating-point-expression* is a negative number, a leading minus sign (-) is included in the result. If *decimal-floating-point-expression* is a positive number, a leading plus sign (+) is included in the result.
PR	Suffix: If *decimal-floating-point-expression* is a negative number, a leading less than character (<) and a trailing greater than character (>) are included in the result. If *decimal-floating-point-expression* is a positive number, a leading space and a trailing space are included in the result.
$	Prefix: A leading dollar sign ($) is included in the result.
,	Specifies that a comma be included in that location in the result. This comma is used as a group separator.
.	Specifies that a period be included in that location in the result. This period is used as a decimal point.

If format-string is not specified, *decimal-floating-point-expression* is formatted in the form of an SQL decimal floating-point constant. If *decimal-floating-point-expression* is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

The result is a varying-length character string representation of *decimal-floating-point-expression*. The length attribute is 254. If the string units of the environment or *format-string* is CODEUNITS32, the string units of the result is CODEUNITS32. Otherwise, the string units of the result is OCTETS. The actual length of the result is determined by the *format-string*, if specified; otherwise, the actual length of the result is the smallest number of characters that can represent the value of *decimal-floating-point-expression*. If the resulting string exceeds the length attribute of the result, the result will be truncated.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings "INFINITY", "SNAN", and "NAN", respectively, are returned. If the special value is negative, the first character of the result is a minus sign. The decimal floating-point special value sNaN does not result in an exception when converted to a string.

If *format-string* does not include any of the format elements MI, S, or PR, and the value of *decimal-floating-point-expression* is negative, then a minus sign (-) will be included in the result; otherwise, a blank will be included in the result.

If the number of digits to the left of the decimal point in *decimal-floating-point-expression* is greater than the number of digits to the left of the decimal point in *format-string*, the result is a string of number sign (#) characters. If the number of digits to the right of the decimal point in *decimal-floating-point-expression* is greater than the number of digits to the right of the decimal point in *format-string*, the result is *decimal-floating-point-expression* rounded to the number of digits to the right of the decimal point in *format-string*. The DECFLOAT rounding mode will not be used. The rounding behavior of VARCHAR_FORMAT corresponds to a value of ROUND_HALF_UP.

VARCHAR_FORMAT

The code page of the result is the code page of the section.

Notes

- *Julian and Gregorian calendar*: For Timestamp to varchar, the transition from the Julian calendar to the Gregorian calendar on 15 October 1582 is taken into account by this function.
- *Determinism:* VARCHAR_FORMAT is a deterministic function. However, the following invocations of the function depend on the value of the special register CURRENT LOCALE LC_TIME.
  - Timestamp to varchar, when format-string is not explicitly specified, or when locale-name is not explicitly specified and one of the following is true:
    - *format-string* is not a constant
    - *format-string* is a constant and includes format elements that are locale sensitive

    These invocations that depend on the value of a special register cannot be used wherever special registers cannot be used (SQLSTATE 42621, 428EC, or 429BX).
- *Syntax alternatives:* TO_CHAR is a synonym for VARCHAR_FORMAT.

Examples

- *Example 1*: Display the 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
```

- *Example 2*: Assume that the variable TMSTAMP is defined as a TIMESTAMP and has the following value: 2007-03-09-14.07.38.123456. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

```
Function invocation                         Result
-------------------                         ------
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMISSFF3') 20070309020738123
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHH24MISS')  20070309140738
VARCHAR_FORMAT(TMSTAMP,'YYYYMMDDHHMI')      200703090207
VARCHAR_FORMAT(TMSTAMP,'DD/MM/YY')          09/03/07
VARCHAR_FORMAT(TMSTAMP,'MM-DD-YYYY')        03-09-2007
VARCHAR_FORMAT(TMSTAMP,'J')                 2454169
VARCHAR_FORMAT(TMSTAMP,'Q')                 1
VARCHAR_FORMAT(TMSTAMP,'W')                 2
VARCHAR_FORMAT(TMSTAMP,'IW')                10
VARCHAR_FORMAT(TMSTAMP,'WW')                10
VARCHAR_FORMAT(TMSTAMP,'Month','en_US')     March
VARCHAR_FORMAT(TMSTAMP,'MONTH','en_US')     MARCH
VARCHAR_FORMAT(TMSTAMP,'MON','en_US')       MAR
VARCHAR_FORMAT(TMSTAMP,'Day','en_US')       Friday
VARCHAR_FORMAT(TMSTAMP,'DAY','en_US')       FRIDAY
VARCHAR_FORMAT(TMSTAMP,'Dy','en_US')        Fri
VARCHAR_FORMAT(TMSTAMP,'Month','de_DE')     März
VARCHAR_FORMAT(TMSTAMP,'MONTH','de_DE')     MÄRZ
```

```
VARCHAR_FORMAT(TMSTAMP,'MON','de_DE')          MRZ
VARCHAR_FORMAT(TMSTAMP,'Day','de_DE')          Freitag
VARCHAR_FORMAT(TMSTAMP,'DAY','de_DE')          FREITAG
VARCHAR_FORMAT(TMSTAMP,'Dy','de_DE')           Fr
```

- *Example 3*: Assume that the variable DTE is defined as a DATE and has the following value: 2007-03-09. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

```
Function invocation                      Result
-------------------                      ------
VARCHAR_FORMAT(DTE,'YYYYMMDD')           20070309
VARCHAR_FORMAT(DTE,'YYYYMMDDHH24MISS')   20070309000000
```

- *Example 4*: Assume that the variables POSNUM and NEGNUM are defined as DECFLOAT(34) and have the following values: 1234.56 and -1234.56, respectively. The following examples show several invocations of the function and the resulting string values. The result data type in each case is VARCHAR(254).

```
Function invocation                      Result
-------------------                      ------
VARCHAR_FORMAT(POSNUM)                    '1234.56'
VARCHAR_FORMAT(NEGNUM)                    '-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99')          '1234.56'
VARCHAR_FORMAT(NEGNUM,'9999.99')          '1234.56'
VARCHAR_FORMAT(POSNUM,'99999.99')         ' 1234.56'
VARCHAR_FORMAT(NEGNUM,'99999.99')         ' 1234.56'
VARCHAR_FORMAT(POSNUM,'00000.00')         '01234.56'
VARCHAR_FORMAT(NEGNUM,'00000.00')         '01234.56'
VARCHAR_FORMAT(POSNUM,'9999.99MI')        '1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99MI')        '1234.56-'
VARCHAR_FORMAT(POSNUM,'S9999.99')         '+1234.56'
VARCHAR_FORMAT(NEGNUM,'S9999.99')         '-1234.56'
VARCHAR_FORMAT(POSNUM,'9999.99PR')        ' 1234.56 '
VARCHAR_FORMAT(NEGNUM,'9999.99PR')        '<1234.56>'
VARCHAR_FORMAT(POSNUM,'S$9,999.99')       '+$1,234.56'
VARCHAR_FORMAT(NEGNUM,'S$9,999.99')       '-$1,234.56'
```

VARCHAR_FORMAT_BIT

The VARCHAR_FORMAT_BIT function returns a character representation of a bit string that has been formatted using a character template.

►►—VARCHAR_FORMAT_BIT—(—*bit-data-expression*—,—*format-string*—)————————►◄

The schema is SYSIBM.

bit-data-expression
> An expression that returns a value that is a built-in character-string FOR BIT DATA data type (SQLSTATE 42815). The required length is determined by the specified format string and how the value is interpreted.

format-string
> A character constant that contains a template for how the result is to be formatted.

> Valid format strings include: `'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'` and `'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX'` (SQLSTATE 42815) where each 'x' or 'X' corresponds to one hexadecimal digit from *bit-data-expression*.

The result of the function is a varying-length character string with the length attribute and actual length based on the format string. If the string units of the environment or *format-string* is CODEUNITS32, the string units of the result is CODEUNITS32. Otherwise, the string units of the result is OCTETS For the two valid format strings listed previously, the length attribute is 36 and the actual length is 36 bytes. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Examples

- *Example 1*: Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492'
                   AS VARCHAR(16) FOR BIT DATA),
              'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx')
```

  Result returned:

  `'d83d6360-1818-11db-9804-b622a1ef5492'`

- *Example 2*: Represent a Universal Unique Identifier in its formatted form:

```
VARCHAR_FORMAT_BIT(CAST(x'd83d6360181811db9804b622a1ef5492'
                   AS CHAR(16) FOR BIT DATA),
              'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

  Result returned:

  `'D83D6360-1818-11DB-9804-B622A1EF5492'`

VARGRAPHIC

The VARGRAPHIC function returns a varying-length graphic string representation of a variety of data types.

Integer to vargraphic:

►►—VARGRAPHIC—(—*integer-expression*—)———————————————►◄

Decimal to vargraphic:

►►—VARGRAPHIC—(—*decimal-expression*————————————)———————►◄
                        └—,—*decimal-character*—┘

Floating-point to vargraphic:

►►—VARGRAPHIC—(—*floating-point-expression*———————————————)►◄
                            └—,—*decimal-character*—)┘

Decimal floating-point to vargraphic:

►►—VARGRAPHIC—(—*decimal-floating-point-expression*———————————)►◄
                                 └—,—*decimal-character*—)┘

Character to vargraphic:

►►—VARGRAPHIC—(—*character-expression*———————————————)►◄
                        └—,—*integer*—)┘

Graphic to vargraphic:

►►—VARGRAPHIC—(—*graphic-expression*—————————)———————————►◄
                      └—,—*integer*—┘

Datetime to vargraphic:

►►—VARGRAPHIC—(—*datetime-expression*————————————)———————►◄
                        └—,—┬—ISO—┬—┘
                             ├—USA—┤
                             ├—EUR—┤
                             ├—JIS—┤
                             └—LOCAL—┘

The schema is SYSIBM. The function name cannot be specified as a qualified name when keywords are used in the function signature.

The VARGRAPHIC function returns a varying-length graphic string representation of:

- An integer number (Unicode database only), if the first argument is a SMALLINT, INTEGER, or BIGINT

- A decimal number (Unicode database only), if the first argument is a decimal number
- A double-precision floating-point number (Unicode database only), if the first argument is a DOUBLE or REAL
- A decimal floating-point number (Unicode database only), if the first argument is a decimal floating-point number (DECFLOAT)
- A character string, if the first argument is any type of character string
- A graphic string, if the first argument is any type of graphic string
- A datetime value (Unicode databases only), if the first argument is a DATE, TIME, or TIMESTAMP

In a non-Unicode database, the string units of the result is double bytes. Otherwise, the string units of the result is determined by the data type of the first argument.
- CODEUNITS16, if the first argument is character string or a graphic string with string units of OCTETS or CODEUNITS16.
- CODEUNITS32, if the first argument is character string or a graphic string with string units of CODEUNITS32.
- Determined by the default string unit of the environment, if the first argument is not a character string or a graphic string.

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.

Integer to vargraphic

> *integer-expression*
> > An expression that returns a value that is of an integer data type (SMALLINT, INTEGER, or BIGINT).
>
> > The result is the varying-length graphic string representation of *integer-expression* in the form of an SQL integer constant. The length attribute of the result depends on whether *integer-expression* is a small, large or big integer as follows:
> > - If the first argument is a small integer, the maximum length of the result is 6.
> > - If the first argument is a large integer, the maximum length of the result is 11.
> > - If the first argument is a big integer, the maximum length of the result is 20.
>
> > The actual length of the result is the smallest number of double-byte characters that can be used to represent the value of the argument. Leading zeros are not included. If the argument is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit.
>
> > The code page of the result is the DBCS code page of the section.

Decimal to vargraphic

> *decimal-expression*
> > An expression that returns a value that is a decimal data type. The DECIMAL scalar function can be used to change the precision and scale.

decimal-character
> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *decimal-expression* in the form of an SQL decimal constant. The length attribute of the result is 2+p, where p is the precision of *decimal-expression*. The actual length of the result is the smallest number of double-byte characters that can be used to represent the result, except that trailing zeros are included. Leading zeros are not included. If *decimal-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit or the decimal character. If the scale of *decimal-expression* is zero, the decimal character is not returned.

The code page of the result is the DBCS code page of the section.

Floating-point to vargraphic

floating-point-expression
> An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *floating-point-expression* in the form of an SQL floating-point constant.

The maximum length of the result is 24. The actual length of the result is the smallest number of double-byte characters that can represent the value of *floating-point-expression* such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If *floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *floating-point-expression* is zero, the result is 0E0.

The code page of the result is the DBCS code page of the section.

Decimal floating-point to vargraphic

decimal-floating-point-expression
> An expression that returns a value that is a decimal floating-point data type (DECFLOAT).

decimal-character
> Specifies the double-byte character constant that is used to delimit the decimal digits in the result graphic string. The double-byte character constant cannot be a digit, the plus sign (+), the minus sign (-), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is a varying-length graphic string representation of *decimal-floating-point-expression* in the form of an SQL decimal floating-point constant. The maximum length of the result is 42. The actual length of the result is the smallest number of double-byte characters that can represent

the value of *decimal-floating-point-expression*. If *decimal-floating-point-expression* is negative, the first double-byte character of the result is a minus sign; otherwise, the first double-byte character is a digit. If *decimal-floating-point-expression* is zero, the result is 0.

If the value of *decimal-floating-point-expression* is the special value Infinity, sNaN, or NaN, the strings G'INFINITY', G'SNAN', and G'NAN', respectively, are returned. If the special value is negative, the first double-byte character of the result is a minus sign. The decimal floating-point special value sNaN does not result in a warning when converted to a string.

The code page of the result is the DBCS code page of the section.

Character to vargraphic

In Unicode databases:

character-expression
> An expression that returns a value that is a built-in character string data type. The expression must not be a FOR BIT DATA subtype (SQLSTATE 42846).

integer
> An integer constant that specifies the length attribute for the resulting varying-length graphic string. The value must be between 0 and the maximum length for the VARGRAPHIC data type in the string units of the result.
>
> If the second argument is not specified:
> - If the *character-expression* is the empty string constant, the length attribute of the result is 0.
> - Otherwise, the length attribute of the result is the lower of the following values:
>   - The maximum length for the VARGRAPHIC data type in the string units of the result.
>   - The length attribute of the first argument.

The result is a varying-length graphic string that is converted from *character-expression*. The length attribute of the result is determined by the value of *integer*.

If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, several scenarios exist:
- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *character-expression* is CHAR or VARCHAR, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).
- If *integer* is specified, truncation is performed with a warning returned (SQLSTATE 01004). When the output string is truncated, such that the last character is a high surrogate, that surrogate is deleted. Do not rely on this behavior because it might change in a future release.
- If *integer* is not specified and *character-expression* is a VARCHAR, truncation is performed with a warning returned (SQLSTATE 01004).
- If *integer* is not specified and character-expression is a CLOB, an error is returned (SQLSTATE 22001).

This function converts *character-expression* from UTF-8 to UTF-16. Every character of *character-expression* is converted.

In non-Unicode databases:

character-expression
> An expression that returns a value that is a built-in CHAR or VARCHAR data type.

The result is a varying-length graphic string that is converted from *character-expression*. The length attribute of the result is the minimum of the length attribute of *character-expression* and the maximum length for the VARGRAPHIC data type.
If the length of *character-expression* that is converted to a graphic string is greater than the length attribute of the result, an error is returned (SQLSTATE 22001).
For databases with a code set that is not Japanese EUC (code page 954) or Traditional Chinese (code page 964), each single-byte character in *character-expression* is converted to its equivalent double-byte representation or to the double-byte substitution character in the result. Each double-byte character in *character-expression* is mapped without extra conversion. If the first byte of a double-byte character appears as the last byte of *character-expression*, it is converted to the double-byte substitution character. The sequential order of the characters in *character-expression* is preserved. No warning or error code is generated if one or more double-byte substitution characters are returned in the result.
For details about the conversion process for databases with a code set that is Japanese EUC (code page 954) or Traditional Chinese (code page 964), refer to the topic "Japanese and traditional-Chinese extended UNIX code (EUC) considerations" in Globalization Guide.

Graphic to vargraphic

graphic-expression
> An expression that returns a value that is a built-in graphic string data type.

integer
> An integer constant that specifies the length attribute for the resulting varying-length graphic string. The value must be between 0 and the maximum length for the VARGRAPHIC data type in the string units of the result. If the second argument is not specified:
> - If the *graphic-expression* is the empty string constant, the length attribute of the result is 0.
> - Otherwise, the length attribute of the result is the lower of the following values:
>   - The maximum length for the VARGRAPHIC data type in the string units of the result
>   - The length attribute of the first argument

The result is a varying-length graphic string. The length attribute of the result is determined by the value of *integer*.

If the length of *graphic-expression* is greater than the length attribute of the result, several scenarios exist:
- If the string unit of the result is CODEUNITS32, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004).

- If integer is specified, truncation is performed. If only blank characters are truncated and *graphic-expression* is GRAPHIC or VARGRAPHIC, no warning is returned. Otherwise, a warning is returned (SQLSTATE 01004). In a Unicode database, when the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.
- If *integer* is not specified, an error is returned (SQLSTATE 22001).

Datetime to vargraphic

datetime-expression
An expression that is one of the following data types:

DATE   The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIME   The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

TIMESTAMP
The result is the graphic string representation of the timestamp. If the data type of *datetime-expression* is TIMESTAMP(0), the length of the result is 19. If the data type of *datetime-expression* is TIMESTAMP(*n*), where *n* is between 1 and 12, the length of the result is 20+*n*. Otherwise, the length of the result is 26. The second argument must not be specified (SQLSTATE 42815).

The code page of the result is the DBCS code page of the section.

Notes

- The CAST specification should be used to increase the portability of applications when the first argument is numeric, or if the first argument is a string and the length argument is specified. For more information, see "CAST specification".

Examples

- *Example 1*: The EDLEVEL column is defined as SMALLINT. The following returns the value as a varying-length graphic string.

```
SELECT VARGRAPHIC(EDLEVEL)
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```
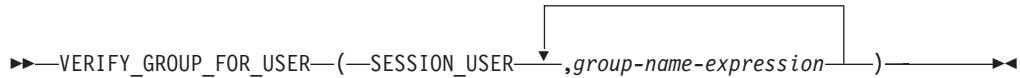
Results in the value G'18'.

- *Example 2*: The SALARY and COMM columns are defined as DECIMAL with a precision of 9 and a scale of 2. Return the total income for employee Haas using the comma decimal character.

```
SELECT VARGRAPHIC(SALARY + COMM, ',')
  FROM EMPLOYEE
  WHERE LASTNAME = 'HAAS'
```

Results in the value G'56970,00'.

VERIFY_GROUP_FOR_USER

The VERIFY_GROUP_FOR_USER function returns a value that indicates whether the groups associated with the authorization ID identified by the SESSION_USER special register are in the group names specified by the list of *group-name-expression* arguments.

```
►►──VERIFY_GROUP_FOR_USER──(──SESSION_USER──┬─,group-name-expression─┬──)──────►◄
                                            └◄─────────────────────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

group-name-expression

An expression that specifies an authorization name (SQLSTATE 42815). The existence of the authorization name at the current server is not verified. *group-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB (SQLSTATE 42815). The content of the string is not folded to uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if any of the groups associated with the authorization ID identified by the SESSION_USER special register are in the list of *group-name-expression* arguments. Otherwise, the result is 0.

Example

The tellers in a bank can only access customers from their own branch. All tellers are members in the group TELLER. A row permission is created by a user with SECADM authority to enforce this rule.
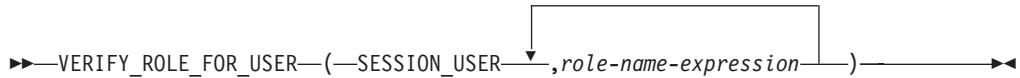
```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE VERIFY_GROUP_FOR_USER(SESSION_USER,'TELLER') = 1 AND
      BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
          WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

VERIFY_ROLE_FOR_USER

The VERIFY_ROLE_FOR_USER function returns a value that indicates whether
any of the roles associated with the authorization ID identified by the
SESSION_USER special register are in (or contain any of) the role names specified
by the list of *role-name-expression* arguments.

```
►►─VERIFY_ROLE_FOR_USER─(─SESSION_USER─┬─,role-name-expression─┬─)───►◄
                                       └──────────◄────────────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

role-name-expression
An expression that specifies a role name (SQLSTATE 42815). The existence of
the role name at the current server is not verified. *role-name-expression* must
return a built-in character string data type or graphic string data type that is
not a LOB (SQLSTATE 42815). The content of the string is not folded to
uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if
any of the roles associated with the authorization ID identified by the
SESSION_USER special register are in (or contain any of) the role names specified
by the list of *role-name-expression* arguments. Otherwise, the result is 0.

Example

The tellers in a bank can only access customers from their own branch. All tellers
are members in the role TELLER. A row permission is created by a user with the
SECADM authority to enforce this rule.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE VERIFY_ROLE_FOR_USER(SESSION_USER,'TELLER') = 1 AND
      BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
          WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER

The VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER function returns a value that indicates whether the authorization ID identified by the SESSION_USER special register has acquired a role under a trusted connection associated with some trusted context and that role is in (or part of) the role names specified by the list of *role-name-expression* arguments.

```
►►─VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER─(─SESSION_USER─┬─,role-name-expression─┬─)──────►◄
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

role-name-expression
An expression that specifies a role name (SQLSTATE 42815). The existence of the role name at the current server is not verified. *role-name-expression* must return a built-in character string data type or graphic string data type that is not a LOB (SQLSTATE 42815). The content of the string is not folded to uppercase and is not left-aligned.

The result of the function is an integer. The result cannot be null. The result is 1 if the authorization ID identified by the SESSION_USER special register has acquired a role under a trusted connection associated with some trusted context and that role is in (or contained in any of) the role names specified by the list of *role-name-expression* arguments. Otherwise, the result is 0.

Example

The tellers in a bank can only access customers from their own branch. All tellers are members in the role TELLER, which can only be acquired through a trusted connection. A row permission is created by a user with SECADM authority to enforce this rule.

```
CREATE PERMISSION TELLER_ROW_ACCESS ON CUSTOMER
  FOR ROWS WHERE
    VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER(SESSION_USER,'TELLER') = 1 AND
        BRANCH = (SELECT HOME_BRANCH FROM INTERNAL_INFO
            WHERE EMP_ID = USER)
ENFORCED FOR ALL ACCESS
ENABLE
```

WEEK

The WEEK scalar function returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.

►►——WEEK——(——*expression*——)————————————————————————————►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

WEEK_ISO

The WEEK_ISO function returns the week of the year of the argument as an integer value in the range 1-53.

►►──WEEK_ISO──(──*expression*──)──►◄

The schema is SYSFUN.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The week starts with Monday and always includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. It is therefore possible to have up to 3 days at the beginning of a year appear in the last week of the previous year. Conversely, up to 3 days at the end of a year may appear in the first week of the next year.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.
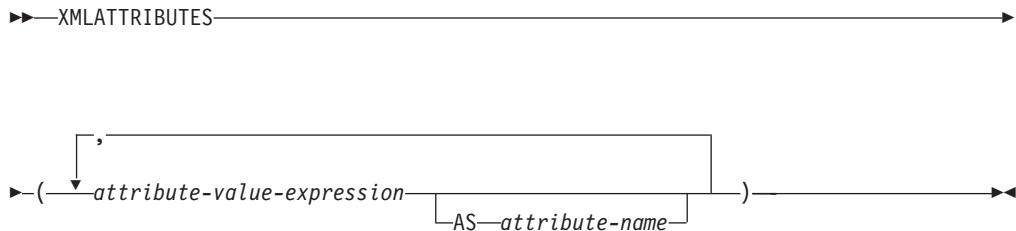
Example

The following list shows examples of the result of WEEK_ISO and DAYOFWEEK_ISO.

```
DATE        WEEK_ISO    DAYOFWEEK_ISO
----------  ----------- -------------
1997-12-28          52              7
1997-12-31           1              3
1998-01-01           1              4
1999-01-01          53              5
1999-01-04           1              1
1999-12-31          52              5
2000-01-01          52              6
2000-01-03           1              1
```

XMLATTRIBUTES

The XMLATTRIBUTES function constructs XML attributes from the arguments.

```
►►─XMLATTRIBUTES──────────────────────────────────────────►

                    ┌─ , ◄──────────────────────────┐
►─(──▼──attribute-value-expression──┬────────────────────────┬─▼──)──►◄
                                    └─AS──attribute-name──────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

This function can only be used as an argument of the XMLELEMENT function. The result is an XML sequence containing an XQuery attribute node for each non-null input value.

attribute-value-expression
> An expression whose result is the attribute value. The data type of *attribute-value-expression* cannot be an XML type or a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an attribute name must be specified.

attribute-name
> Specifies an attribute name. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The attribute name cannot be xmlns or prefixed with xmlns:. A namespace is declared using the function XMLNAMESPACES. Duplicate attribute names, whether implicit or explicit, are not allowed (SQLSTATE 42713).

> If *attribute-name* is not specified, *attribute-value-expression* must be a column name (SQLSTATE 42703). The attribute name is created from the column name using the fully escaped mapping from a column name to an XML attribute name.

The data type of the result is XML. If the result of *attribute-value-expression* can be null, the result can be null; if the result of every *attribute-value-expression* is null, the result is the null value.

Examples

Note: XMLATTRIBUTES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Produce an element with attributes.

```
SELECT E.EMPNO, XMLELEMENT(
  NAME "Emp",
  XMLATTRIBUTES(
    E.EMPNO, E.FIRSTNME ||' '|| E.LASTNAME AS "name"
  )
)
AS "Result"
FROM EMPLOYEE E WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
EMPNO  Result
000290 <Emp EMPNO="000290" name="JOHN PARKER"></Emp>
000310 <Emp EMPNO="000310" name="MAUDE SETRIGHT"></Emp>
200310 <Emp EMPNO="200310" name="MICHELLE SPRINGER"></Emp>
```

- *Example 2*: Produce an element with a namespace declaration that is not used in any QName. The prefix is used in an attribute value.

```
VALUES XMLELEMENT(
  NAME "size",
  XMLNAMESPACES(
    'http://www.w3.org/2001/XMLSchema-instance' AS "xsi",
    'http://www.w3.org/2001/XMLSchema' AS "xsd"
  ),
  XMLATTRIBUTES(
    'xsd:string' AS "xsi:type"
  ), '1'
)
```

This query produces the following result:

```
<size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:type="xsd:string">1</size>
```

XMLCOMMENT

The XMLCOMMENT function returns an XML value with a single XQuery comment node with the input argument as the content.

►►──XMLCOMMENT──(──*string-expression*──)──────────────────────────────────►◄

The schema is SYSIBM. The function name cannot be specified as a qualified name.

string-expression
 An expression whose value has a character string type: CHAR, VARCHAR or CLOB. The result of the *string-expression* is parsed to check for conformance to the requirements for an XML comment, as specified in the XML 1.0 rule. The result of the *string-expression* must conform to the following regular expression:
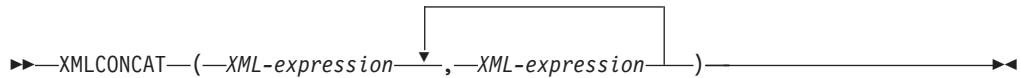
  ```((Char - '-') | ('-' (Char - '-')))*```

 where Char is defined as any Unicode character excluding surrogate blocks X'FFFE' and X'FFFF'. Basically, the XML comment cannot contain two adjacent hyphens, and cannot end with a hyphen (SQLSTATE 2200S).

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value.

XMLCONCAT

The XMLCONCAT function returns a sequence containing the concatenation of a variable number of XML input arguments.

>>—XMLCONCAT—(—*XML-expression*——,—*XML-expression*——)————————><

The schema is SYSIBM. The function name cannot be specified as a qualified name.

XML-expression
    Specifies an expression of data type XML.

The data type of the result is XML. The result is an XML sequence containing the concatenation of the non-null input XML values. Null values in the input are ignored. If the result of any *XML-expression* can be null, the result can be null; if the result of every input value is null, the result is the null value.

Example

Note: XMLCONCAT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

Construct a department element for departments A00 and B01, containing a list of employees sorted by first name. Include an introductory comment immediately preceding the department element.

```
SELECT XMLCONCAT(
  XMLCOMMENT(
    'Confirm these employees are on track for their product schedule'
  ),
  XMLELEMENT(
    NAME "Department",
    XMLATTRIBUTES(
      E.WORKDEPT AS "name"
    ),
    XMLAGG(
      XMLELEMENT(
        NAME "emp", E.FIRSTNME
      )
      ORDER BY E.FIRSTNME
    )
  )
)
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY E.WORKDEPT
```

This query produces the following result:

```
<!--Confirm these employees are on track for their product schedule-->
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>DIAN</emp>
<emp>GREG</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<!--Confirm these employees are on track for their product schedule-->
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

XMLDOCUMENT

The XMLDOCUMENT function returns an XML value with a single XQuery document node with zero or more children nodes.

►►─XMLDOCUMENT─(─*XML-expression*─)──────────────────────────────►◄

The schema is SYSIBM. The function name cannot be specified as a qualified name.

XML-expression
    An expression that returns an XML value. A sequence item in the XML value must not be an attribute node (SQLSTATE 10507).

The data type of the result is XML. If the result of *XML-expression* can be null, the result can be null; if the input value is null, the result is the null value.

The children of the resulting document node are constructed as described in the following steps. The input expression is a sequence of nodes or atomic values, which is referred to in these steps as the content sequence.

1. If the content sequence contains a document node, the document node is replaced in the content sequence by the children of the document node.

2. Each adjacent sequence of one or more atomic values in the content sequence are replaced with a text node containing the result of casting each atomic value to a string with a single blank character inserted between adjacent values.

3. For each node in the content sequence, a new deep copy of the node is constructed. A deep copy of a node is a copy of the whole subtree rooted at that node, including the node itself and its descendants. Each copied node has a new node identity.

4. The nodes in the content sequence become the children of the new document node.

The XMLDOCUMENT function effectively executes the XQuery computed document constructor. The result of

```
XMLQUERY('document {$E}' PASSING BY REF XML-expression AS "E")
```

is equivalent to

```
XMLDOCUMENT( XML-expression )
```

with the exception of the case where *XML-expression* is null and XMLQUERY returns the empty sequence compared to XMLDOCUMENT which returns the null value.

Example
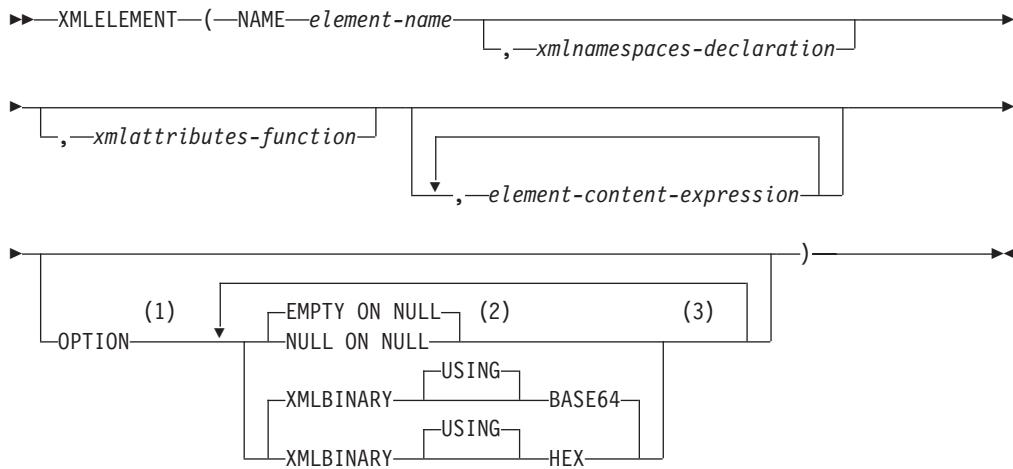
Insert a constructed document into an XML column.

```
INSERT INTO T1 VALUES(
  123, (
    SELECT XMLDOCUMENT(
      XMLELEMENT(
        NAME "Emp", E.FIRSTNME || ' ' || E.LASTNAME, XMLCOMMENT(
          'This is just a simple example'
        )
      )
    )
```

```
      FROM EMPLOYEE E
      WHERE E.EMPNO = '000120'
    )
  )
```

XMLELEMENT

The XMLELEMENT function returns an XML value that is an XQuery element node.

```
►►──XMLELEMENT──(──NAME──element-name────────────────────────────────────►
                                        └─,──xmlnamespaces-declaration─┘

►──────────────────────────────────────────────────────────────────────►
   └─,──xmlattributes-function─┘  ┌──────────────────────────────┐
                                  │        ┌◄───────────────────┐ │
                                  └────────,──element-content-expression─┘

►────────────────────────────────────────────────)──►◄
   │          (1)  ┌◄──────────────────────────┐               │
   │               │  ┌─EMPTY ON NULL─┐  (2)          (3)      │
   └─OPTION────────┼──┼─NULL ON NULL──┤───────────────────────┘
                   │  │                                        │
                   │  │            ┌─USING─┐                   │
                   │  └─XMLBINARY───┴───────┴──BASE64─┐        │
                   │                 ┌─USING─┐         │        │
                   └──────XMLBINARY──┴───────┴──HEX────┘
```

Notes:

1   The OPTION clause can only be specified if at least one *xmlattributes-function* or *element-content-expression* is specified.

2   NULL ON NULL or EMPTY ON NULL can only be specified if at least one *element-content-expression* is specified.

3   The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

NAME *element-name*
Specifies the name of an XML element. The name is an SQL identifier that must be in the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635).

xmlnamespaces-declaration
Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLELEMENT function. The namespaces apply to any nested XML functions within the XMLELEMENT function, regardless of whether or not they appear inside another subselect.

If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed element.

xmlattributes-function
Specifies the XML attributes for the element. The attributes are the result of the XMLATTRIBUTES function.

element-content-expression
The content of the generated XML element node is specified by an expression or a list of expressions. The data type of *element-content-expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression.

If *element-content-expression* is not specified, an empty string is used as the content for the element and OPTION NULL ON NULL or EMPTY ON NULL must not be specified.

OPTION

Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is EMPTY ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL

Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is EMPTY ON NULL.

EMPTY ON NULL

If the value of each *element-content-expression* is null, an empty element is returned.

NULL ON NULL

If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX

Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64

Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX

Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more arguments that make up the content of the XML element. The result is an XML sequence containing an XML element node or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

Notes

- When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.

- **Constructing an element node**: The resulting element node is constructed as follows:

  1. The *xmlnamespaces-declaration* adds a set of in-scope namespaces for the constructed element. Each in-scope namespace associates a namespace prefix (or the default namespace) with a namespace URI. The in-scope namespaces define the set of namespace prefixes that are available for interpreting QNames within the scope of the element.

  2. If the xmlattributes-function is specified, it is evaluated and the result is a sequence of attribute nodes.

  3. Each *element-content-expression* is evaluated and the result is converted into a sequence of nodes as follows:

     - If the result type is not XML, it is converted to an XML text node whose content is the result of *element-content-expression* mapped to XML according to the rules of mapping SQL data values to XML data values (see the table that describes supported casts from non-XML values to XML values in "Casting between data types").

     - If the result type is XML, then in general the result is a sequence of items. Some of the items in that sequence might be document nodes. Each document node in the sequence is replaced by the sequence of its top-level children. Then for each node in the resulting sequence, a new deep copy of the node is constructed, including its children and attributes. Each copied node has a new node identity. Copied element and attribute nodes preserve their type annotation. For each adjacent sequence of one or more atomic values returned in the sequence, a new text node is constructed, containing the result of casting each atomic value to a string, with a single blank character inserted between adjacent values. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.

  4. The result sequence of XML attributes and the resulting sequences of all *element-content-expression* specifications are concatenated into one sequence which is called the content sequence. Any sequence of adjacent text nodes in the content sequence is merged into a single text node. If all the *element-content-expression* arguments are empty strings, or an *element-content-expression* argument is not specified, an empty element is returned.

  5. The content sequence must not contain an attribute node following a node that is not an attribute node (SQLSTATE 10507). Attribute nodes occurring in the content sequence become attributes of the new element node. Two or more of these attribute nodes must not have the same name (SQLSTATE 10503). A namespace declaration is created corresponding to any namespace used in the names of the attribute nodes if the namespace URI is not in the in-scope namespaces of the constructed element.

  6. Element, text, comment, and processing instruction nodes in the content sequence become the children of the constructed element node.

7. The constructed element node is given a type annotation of `xs:anyType`, and each of its attributes is given a type annotation of `xdt:untypedAtomic`. The node name of the constructed element node is element-name specified after the NAME keyword.

- **Rules for using namespaces within XMLELEMENT**: Consider the following rules about scoping of namespaces:
  - The namespaces declared in the XMLNAMESPACES declaration are the in-scope namespaces of the element node constructed by the XMLELEMENT function. If the element node is serialized, then each of its in-scope namespaces will be serialized as a namespace attribute unless it is an in-scope namespace of the parent of the element node and the parent element is serialized too.
  - If an XMLQUERY or XMLEXISTS is in an *element-content-expression*, then the namespaces becomes the statically known namespaces of the XQuery expression of the XMLQUERY or XMLEXISTS. Statically known namespaces are used to resolve the QNames in the XQuery expression. If the XQuery prolog declares a namespace with the same prefix, within the scope of the XQuery expression, the namespace declared in the prolog will override the namespaces declared in the XMLNAMESPACES declaration.
  - If an attribute of the constructed element comes from an *element-content-expression*, its namespace might not already be declared as an in-scope namespace of the constructed element, in this case, a new namespace is created for it. If this would result in a conflict, which means that the prefix of the attribute name is already bound to a different URI by a in-scope namespace, a prefix is generated that does not cause such a conflict and the prefix used in the attribute name is changed to the new prefix, and a namespace is created for this new prefix. The generated new prefix follows the following pattern: "db2ns-xx", where "x" is a character chosen from the set [A-Z,a-z,0-9]. For example:

```
VALUES XMLELEMENT(
  NAME "c", XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
      PASSING XMLPARSE(
        DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
      ) AS "m"
  )
)
```

returns:

```
<c xmlns:tst="www.ipo.com" tst:b="2"/>
```

A second example:

```
VALUES XMLELEMENT(
  NAME "tst:c", XMLNAMESPACES(
    'www.tst.com' AS "tst"
  ),
  XMLQUERY(
    'declare namespace ipo="www.ipo.com"; $m/ipo:a/@ipo:b'
      PASSING XMLPARSE(
        DOCUMENT '<tst:a xmlns:tst="www.ipo.com" tst:b="2"/>'
      ) AS "m"
  )
)
```

returns:

```
<tst:c xmlns:tst="www.tst.com" xmlns:db2ns-a1="www.ipo.com"
  db2ns-a1:b="2"/>
```

Examples

Note: XMLELEMENT does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Construct an element with the NULL ON NULL option.

```
SELECT E.FIRSTNME, E.LASTNAME, XMLELEMENT(
  NAME "Emp", XMLELEMENT(
    NAME "firstname", E.FIRSTNME
  ),
  XMLELEMENT(
    NAME "lastname", E.LASTNAME
  )
  OPTION NULL ON NULL
)
AS "Result"
FROM EMPLOYEE E
WHERE E.EDLEVEL = 12
```

This query produces the following result:

```
FIRSTNME    LASTNAME      Emp
JOHN        PARKER        <Emp><firstname>JOHN</firstname>
                            <lastname>PARKER</lastname></Emp>
MAUDE       SETRIGHT      <Emp><firstname>MAUDE</firstname>
                            <lastname>SETRIGHT</lastname></Emp>
MICHELLE    SPRINGER      <Emp><firstname>MICHELLE</firstname>
                            <lastname>SPRINGER</lastname></Emp>
```

- *Example 2*: Produce an element with a list of elements nested as child elements.

```
SELECT XMLELEMENT(
  NAME "Department", XMLATTRIBUTES(
    E.WORKDEPT AS "name"
  ),
  XMLAGG(
    XMLELEMENT(
      NAME "emp", E.FIRSTNME
    )
    ORDER BY E.FIRSTNME
  )
)
AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('A00', 'B01')
GROUP BY WORKDEPT
```

This query produces the following result:

```
dept_list
<Department name="A00">
<emp>CHRISTINE</emp>
<emp>SEAN</emp>
<emp>VINCENZO</emp>
</Department>
<Department name="B01">
<emp>MICHAEL</emp>
</Department>
```

- *Example 3*: Creating nested XML elements specifying a default XML element namespace and using a subselect.

```
SELECT XMLELEMENT(
        NAME "root",
        XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
        XMLATTRIBUTES(cid),
            (SELECT
                XMLAGG(
                  XMLELEMENT(
```

```
                    NAME "poid", poid
                )
            )
         FROM purchaseorder
         WHERE purchaseorder.custid = customer.cid
        )
    )
FROM customer
WHERE cid = '1002'
```

The statement returns the following XML document with the default element namespace declared in the root element:

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

- *Example 4*: Using a common table expression with XML namespaces.

  When an XML element is constructed with a common table expression and the element is used in elsewhere in the same SQL statement, any namespace declarations should be specified as part of the element construction. The following statement specifies the default XML namespace in both the common table expression that uses the PURCHASEORDER table to create the poid elements and the SELECT statement that uses the CUSTOMER table to create the root element.

```
WITH tempid(id, elem) AS
  (SELECT custid, XMLELEMENT(NAME "poid",
     XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
       poid)
   FROM purchaseorder )
SELECT XMLELEMENT(NAME "root",
         XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
            XMLATTRIBUTES(cid),
            (SELECT XMLAGG(elem)
             FROM tempid
             WHERE tempid.id = customer.cid )
        )
FROM customer
WHERE cid = '1002'
```

The statement returns the following XML document with a default element namespace declared in the root element.

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid>5000</poid>
  <poid>5003</poid>
  <poid>5006</poid>
</root>
```

In the following statement, the default element namespace is declared only in the SELECT statement that uses the CUSTOMER table to create the root element:

```
WITH tempid(id, elem) AS
    (SELECT custid, XMLELEMENT(NAME "poid", poid)
     FROM purchaseorder )
SELECT XMLELEMENT(NAME "root",
         XMLNAMESPACES(DEFAULT 'http://mytest.uri'),
            XMLATTRIBUTES(cid),
            (SELECT XMLAGG(elem)
             FROM tempid
             WHERE tempid.id = customer.cid )
        )
FROM customer
WHERE cid = '1002'
```
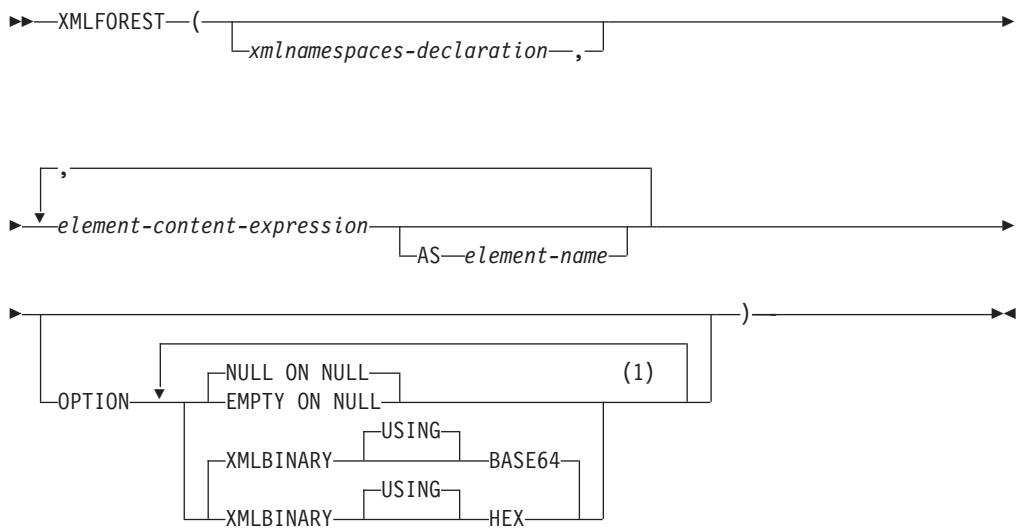
XMLELEMENT

The statement returns the following XML document with the default element namespace declared in the root element. Because the poid elements are created in the common table expression without a default element namespace declaration, the default element namespace for the poid elements is not defined. In the XML document, the default element namespace for the poid elements is set to an empty string "" because the default element namespace for the poid elements is not defined, and the poid elements do not belong to the default element namespace of the root element xmlns="http://mytest.uri".

```
<root xmlns="http://mytest.uri" CID="1002">
  <poid xmlns="">5000</poid>
  <poid xmlns="">5003</poid>
  <poid xmlns="">5006</poid>
</root>
```

XMLFOREST

The XMLFOREST function returns an XML value that is a sequence of XQuery element nodes.

```
►►─XMLFOREST─(─────────────────────────────────────────────────────────
                └─xmlnamespaces-declaration─,─┘


    ┌─,──────────────────────────────────────┐
    ▼                                         ┤
►────┴─element-content-expression───────────────────────────────────────
                              └─AS─element-name─┘


►──────────────────────────────────────────────────────)──────────────►◄
    │                           ┌─NULL ON NULL─┐           (1)        │
    └─OPTION──▼─┬──────────────┬─┴─EMPTY ON NULL─┘───────────────────┤
               │              ┌─USING─┐                              │
               ├─XMLBINARY────┴───────┴─BASE64─┤                     │
               │              ┌─USING─┐                              │
               └─XMLBINARY────┴───────┴─HEX────┘
```

Notes:

1   The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
:   Specifies the XML namespace declarations that are the result of the XMLNAMESPACES declaration. The namespaces that are declared are in the scope of the XMLFOREST function. The namespaces apply to any nested XML functions within the XMLFOREST function, regardless of whether or not they appear inside another subselect.

    If *xmlnamespaces-declaration* is not specified, namespace declarations are not associated with the constructed elements.

element-content-expression
:   The content of the generated XML element node is specified by an expression. The data type of *element-content-expression* cannot be a structured type (SQLSTATE 42884). The expression can be any SQL expression. If the expression is not a simple column reference, an element name must be specified.

AS *element-name*
:   Specifies the XML element name as an SQL identifier. The element name must be of the form of an XML qualified name, or QName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. If the name is qualified, the namespace prefix must be declared within the scope (SQLSTATE 42635). If *element-name* is not specified, *element-content-expression* must be a column name (SQLSTATE 42703). The element name is created from the column name using the fully escaped mapping from a column name to an QName.

XMLFOREST

OPTION
> Specifies additional options for constructing the XML element. If no OPTION clause is specified, the default is NULL ON NULL XMLBINARY USING BASE64. This clause has no impact on nested XMLELEMENT invocations specified in *element-content-expression*.

EMPTY ON NULL or NULL ON NULL
> Specifies whether a null value or an empty element is to be returned if the values of each *element-content-expression* is a null value. This option only affects null handling of element contents, not attribute values. The default is NULL ON NULL.

EMPTY ON NULL
> If the value of each *element-content-expression* is null, an empty element is returned.

NULL ON NULL
> If the value of each *element-content-expression* is null, a null value is returned.

XMLBINARY USING BASE64 or XMLBINARY USING HEX
> Specifies the assumed encoding of binary input data, character string data with the FOR BIT DATA attribute, or a distinct type that is based on one of these types. The encoding applies to element content or attribute values. The default is XMLBINARY USING BASE64.

XMLBINARY USING BASE64
> Specifies that the assumed encoding is base64 characters, as defined for XML schema type xs:base64Binary encoding. The base64 encoding uses a 65-character subset of US-ASCII (10 digits, 26 lowercase characters, 26 uppercase characters, '+', and '/') to represent every six bits of the binary or bit data with one printable character in the subset. These characters are selected so that they are universally representable. Using this method, the size of the encoded data is 33 percent larger than the original binary or bit data.

XMLBINARY USING HEX
> Specifies that the assumed encoding is hexadecimal characters, as defined for XML schema type xs:hexBinary encoding. The hexadecimal encoding represents each byte (8 bits) with two hexadecimal characters. Using this method, the encoded data is twice the size of the original binary or bit data.

This function takes an optional set of namespace declarations and one or more arguments that make up the name and element content for one or more element nodes. The result is an XML sequence containing a sequence of XQuery element nodes or the null value.

The data type of the result is XML. If any of the *element-content-expression* arguments can be null, the result can be null; if all the *element-content-expression* argument values are null and the NULL ON NULL option is in effect, the result is the null value.

The XMLFOREST function can be expressed by using XMLCONCAT and XMLELEMENT. For example, the following two expressions are semantically equivalent.

```
XMLFOREST(xmlnamespaces-declaration, arg1 AS name1, arg2 AS name2 ...)
```

```
XMLCONCAT(
  XMLELEMENT(
    NAME name1, xmlnamespaces-declaration, arg1
  ),
  XMLELEMENT(
    NAME name2, xmlnamespaces-declaration, arg2
  )
  ...
)
```

Notes

- When constructing elements that will be copied as content of another element that defines default namespaces, default namespaces should be explicitly undeclared in the copied element to avoid possible errors that could result from inheriting the default namespace from the new parent element. Predefined namespace prefixes ('xs', 'xsi', 'xml', and 'sqlxml') must also be declared explicitly when they are used.

Example

Note: XMLFOREST does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability. Construct a forest of elements with a default namespace.

```
SELECT EMPNO,
  XMLFOREST(
    XMLNAMESPACES(
      DEFAULT 'http://hr.org', 'http://fed.gov' AS "d"
    ),
    LASTNAME, JOB AS "d:job"
  )
  AS "Result"
  FROM EMPLOYEE
  WHERE EDLEVEL = 12
```

This query produces the following result:

```
EMPNO  Result
000290 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">PARKER
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

000310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SETRIGHT
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>

200310 <LASTNAME xmlns="http://hr.org" xmlns:d="http://fed.gov">SPRINGER
  </LASTNAME>
<d:job xmlns="http://hr.org" xmlns:d="http://fed.gov">OPERATOR</d:job>
```
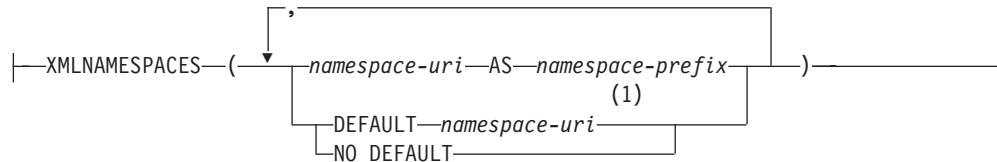
XMLNAMESPACES

The XMLNAMESPACES declaration constructs namespace declarations from the arguments.

xmlnamespaces-declaration:

Notes:

1   DEFAULT or NO DEFAULT can only be specified once in arguments of XMLNAMESPACES.

The schema is SYSIBM. The declaration name cannot be specified as a qualified name.

This declaration can only be used as an argument for specific functions such as XMLELEMENT, XMLFOREST and XMLTABLE. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

namespace-uri
     Specifies the namespace universal resource identifier (URI) as an SQL character string constant. This character string constant must not be empty if it is used with a *namespace-prefix* (SQLSTATE 42815).

namespace-prefix
     Specifies a namespace prefix. The prefix is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The prefix cannot be `xml` or `xmlns` and the prefix must be unique within the list of namespace declarations (SQLSTATE 42635).

DEFAULT *namespace-uri*
     Specifies the default namespace to use within the scope of this namespace declaration. The *namespace-uri* applies for unqualified names in the scope unless overridden in a nested scope by another DEFAULT declaration or a NO DEFAULT declaration.

NO DEFAULT
     Specifies that no default namespace is to be used within the scope of this namespace declaration. There is no default namespace in the scope unless overridden in a nested scope by a DEFAULT declaration.

The data type of the result is XML. The result is an XML namespace declaration for each specified namespace. The result cannot be null.

Examples

Note: XMLNAMESPACES does not insert blank spaces or new line characters in the output. All example output has been formatted to enhance readability.

- *Example 1*: Produce an XML element named adm:employee and an XML attribute adm:department, both associated with a namespace whose prefix is adm.

```
SELECT EMPNO, XMLELEMENT(
  NAME "adm:employee", XMLNAMESPACES(
    'http://www.adm.com' AS "adm"
  ),
  XMLATTRIBUTES(
    WORKDEPT AS "adm:department"
  ),
LASTNAME
)
FROM EMPLOYEE
WHERE JOB = 'ANALYST'
```

This query produces the following result:

```
000130 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  QUINTANA</adm:employee>
000140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  NICHOLLS</adm:employee>
200140 <adm:employee xmlns:adm="http://www.adm.com" adm:department="C01">
  NATZ</adm:employee>
```

- *Example 2*: Produce an XML element named 'employee', which is associated with a default namespace, and a sub-element named 'job', which does not use a default namespace, but whose sub-element named 'department' does use a default namespace.
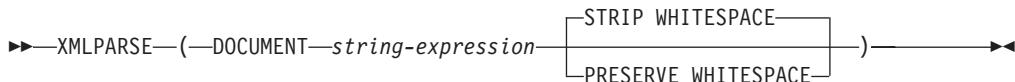
```
SELECT EMP.EMPNO, XMLELEMENT(
  NAME "employee", XMLNAMESPACES(
    DEFAULT 'http://hr.org'
  ),
  EMP.LASTNAME, XMLELEMENT(
    NAME "job", XMLNAMESPACES(
      NO DEFAULT
    ),
    EMP.JOB, XMLELEMENT(
      NAME "department", XMLNAMESPACES(
        DEFAULT 'http://adm.org'
      ),
      EMP.WORKDEPT
    )
  )
)
FROM EMPLOYEE EMP
WHERE EMP.EDLEVEL = 12
```

This query produces the following result:

```
000290 <employee xmlns="http://hr.org">PARKER<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
000310 <employee xmlns="http://hr.org">SETRIGHT<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
200310 <employee xmlns="http://hr.org">SPRINGER<job xmlns="">OPERATOR
  <department xmlns="http://adm.org">E11</department></job></employee>
```

XMLPARSE

The XMLPARSE function parses the argument as an XML document and returns an XML value.

```
►►──XMLPARSE──(──DOCUMENT──string-expression──┬─STRIP WHITESPACE─────┬──)────►◄
                                              └─PRESERVE WHITESPACE──┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

DOCUMENT
> Specifies that the character string expression to be parsed must evaluate to a well-formed XML document that conforms to XML 1.0, as modified by the XML Namespaces recommendation (SQLSTATE 2200M).

string-expression
> Specifies an expression that returns a character string or BLOB value. If a parameter marker is used, it must explicitly be cast to one of the supported data types.

STRIP WHITESPACE or PRESERVE WHITESPACE
> Specifies whether or not whitespace in the input argument is to be preserved. If neither is specified, STRIP WHITESPACE is the default.
>
> **STRIP WHITESPACE**
>> Specifies that text nodes containing only whitespace characters up to 1000 bytes in length will be stripped, unless the nearest containing element has the attribute xml:space='preserve'. If any text node begins with more that 1000 bytes of whitespace, an error is returned (SQLSTATE 54059).
>>
>> The whitespace characters in the CDATA section are also affected by this option. DTDs may have DOCTYPE declarations for elements, but the content models of elements are not used to determine if whitespace is stripped or not.
>
> **PRESERVE WHITESPACE**
>> Specifies that all whitespace is to be preserved, even when the nearest containing element has the attribute xml:space='default'.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value.

Notes

- **Encoding of the input string**: The input string may contain an XML declaration that identifies the encoding of the characters in the XML document. If the string is passed to the XMLPARSE function as a character string, it will be converted to the code page at the database server. This code page may be different from the originating code page and the encoding identified in the XML declaration.

  Therefore, applications should avoid direct use of XMLPARSE with character string input and should send strings containing XML documents directly using host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLPARSE must be used in this situation, a BLOB type should be specified as the argument to avoid code page conversion.

- **Handling of DTDs**: External document type definitions (DTDs) and entities must be registered in a database. Both internal and external DTDs are checked for valid syntax. During the parsing process, the following actions are also performed:
  - Default values that are defined by the internal and external DTDs are applied.
  - Entity references and parameter entities are replaced by their expanded forms.
  - If an internal DTD and an external DTD define the same element, an error is returned (SQLSTATE 2200M).
  - If an internal DTD and an external DTD define the same entity or attribute, the internal definition is chosen.

  After parsing, internal DTDs and entities, as well as references to external DTDs and entities, are not preserved in the stored representation of the value.

- **Character conversion in non-UTF-8 databases**: Code page conversion occurs when an XML document is parsed into a non-Unicode database server, if the document is passed in from a host variable or parameter marker of a character data type, or from a character string literal. Parsing an XML document using a host variable or parameter marker of type XML, BLOB or FOR BIT DATA (CHAR FOR BIT DATA or VARCHAR FOR BIT DATA) prevents code page conversion. When a character data type is used, care must be taken to ensure that all characters in the XML document have a matching code point in the target database code page, otherwise substitution characters may be introduced. The configuration parameter **enable_xmlchar** can be used to help ensure the integrity of XML data stored in a non-Unicode database. Setting this parameter to "NO" blocks the insertion of XML documents from character data types. The BLOB and FOR BIT DATA data types are still allowed, as documents passed into a database using these data types avoid code page conversion.

Example

Using the `PRESERVE WHITESPACE` option preserves the white space characters in the XML document inserted into the table, including the white space characters in the description element.

```
INSERT INTO PRODUCT VALUES ('100-103-99','Tool bag',14.95,NULL,NULL,NULL,
XMLPARSE( DOCUMENT
  '<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

The XMLPI function returns an XML value with a single XQuery processing instruction node.

►►─XMLPI─(─NAME─*pi-name*─────────────────────)─────────────►◄
                         └─,─*string-expression*─┘

The schema is SYSIBM. The function name cannot be specified as a qualified name.

NAME *pi-name*
> Specifies the name of a processing instruction. The name is an SQL identifier that must be in the form of an XML NCName (SQLSTATE 42634). See the W3C XML namespace specifications for more details on valid names. The name cannot be the word 'xml' in any case combination (SQLSTATE 42634).

string-expression
> An expression that returns a value that is a character string. The resulting string is converted to UTF-8 and must conform to the content of an XML processing instruction as specified in XML 1.0 rules (SQLSTATE 2200T):
>
> - The string must not contain the substring '?>' since this substring terminates a processing instruction
> - Each character of the string can be any Unicode character excluding the surrogate blocks, X'FFFE' and X'FFFF'.
>
> The resulting string becomes the content of the constructed processing instruction node.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the result of *string-expression* is null, the result is the null value. If *string-expression* is an empty string or is not specified, an empty processing instruction node is returned.

Examples

- *Example 1*: Generate an XML processing instruction node.

```
SELECT XMLPI(
  NAME "Instruction", 'Push the red button'
)
FROM SYSIBM.SYSDUMMY1
```

This query produces the following result:

```
<?Instruction Push the red button?>
```

- *Example 2*: Generate an empty XML processing instruction node.

```
SELECT XMLPI(
  NAME "Warning"
)
FROM SYSIBM.SYSDUMMY1
```
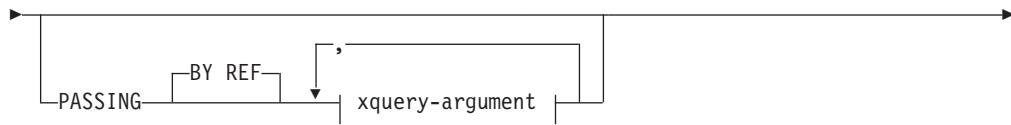
This query produces the following result:

```
<?Warning ?>
```

XMLQUERY

The XMLQUERY function returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.

```
►►──XMLQUERY──(──xquery-expression-constant─────────────────────────────────►

►──┬────────────────────────────────────────────────────┬──────────────────►
   │              ┌─BY REF─┐   ┌─,───────────────┐        │
   └─PASSING──────┴────────┴──▼──xquery-argument─┴─────────┘

   ┌────────────────────────────────────────────────────────────
   │                    ┌─BY REF─┐
   └─RETURNING──SEQUENCE─┴────────┴──┬─────────────────┬──)─────►◄
                                     └─EMPTY ON EMPTY──┘
```

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.

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 way of the default passing mechanism that is provided through the syntax that follows the PASSING keyword. This option cannot be specified for non-XML values. When a non-XML value is passed, the value is converted to XML; this process creates a copy.

RETURNING SEQUENCE
Indicates that the XMLQUERY expression returns a sequence.

BY REF
Indicates that the result of the XQuery expression is returned by reference. If

this value contains nodes, any expression using the return value of the XQuery expression will receive node references directly, preserving all node properties, including the original node identities and document order. Referenced nodes will remain connected within their node trees. If the BY REF clause is not specified and the PASSING is specified, the default passing mechanism is used. If BY REF is not specified and PASSING is not specified, the default returning mechanism is BY REF.

EMPTY ON EMPTY
Specifies that an empty sequence result from processing the XQuery expression is returned as an empty sequence.

The data type of the result is XML; it cannot be null.
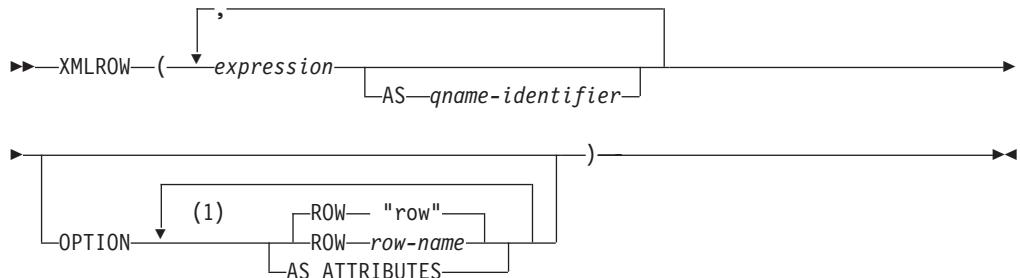
If the evaluation of the XQuery expression results in an error, then the XMLQUERY function returns the XQuery error (SQLSTATE class '10').

Notes

- **XMLQUERY usage restrictions**: The XMLQUERY function cannot be:
  - Part of the ON clause that is associated with a JOIN operator or a MERGE statement (SQLSTATE 42972)
  - Part of the GENERATE KEY USING or RANGE THROUGH clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E3)
  - Part of the FILTER USING clause in the CREATE FUNCTION (External Scalar) statement, or the FILTER USING clause in the CREATE INDEX EXTENSION statement (SQLSTATE 428E4)
  - Part of a check constraint or a column generation expression (SQLSTATE 42621)
  - Part of a group-by-clause (SQLSTATE 42822)
  - Part of an argument for a column-function (SQLSTATE 42607)
- **XMLQUERY as a subquery**: An XMLQUERY expression that acts as a subquery can be restricted by statements that restrict subqueries.

XMLROW

The XMLROW function returns an XML value with a single XQuery document node containing one top-level element node.

```
>>─XMLROW──(──┬──expression──┬────────────────────┬──┬───────────
              │              └─AS──qname-identifier─┘  │
              └─────────────────────,◄─────────────────┘

  ┌────────────────────────────────────)──────────────────────><
  │          ┌─(1)─┬─ROW── "row"──────┐
  └─OPTION──┬─▼────┼─ROW──row-name─────┼──┐
            │      └─AS ATTRIBUTES─────┘  │
            └────────────────────────────┘
```

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). The element name or attribute name is created from the column name using the fully escaped mapping from a column name to an QName.

OPTION
> Specifies additional options for constructing the XML value. If no OPTION clause is specified, the default behavior applies.

AS ATTRIBUTES
> Specifies that each expression is mapped to an attribute value with column name or *qname-identifier* serving as the attribute name.

ROW *row-name*
> Specifies the name of the element to which each row is mapped. If this option is not specified, the default element name is "row".

Notes

By default, each row in the result set is mapped to an XML value as follows:
- Each row is transformed into an XML element named "row" and each column is transformed into a nested element with the column name as the element name.

- The null handling behavior is NULL ON NULL. A null value in a column maps to the absence of the subelement. If all column values are null, a null value is returned by the function.
- The binary encoding scheme for BLOB and FOR BIT DATA data types is base64Binary encoding.
- A document node will be added implicitly to the row element to make the XML result a well-formed single-rooted XML document.

Examples

Assume the following table T1 with columns C1 and C2 that contain numeric data stored in a relational format:

```
C1          C2
----------- -----------
          1           2
          -           2
          1           -
          -           -
```

```
  4 record(s) selected.
```

- *Example 1*: The following example shows an XMLRow query and output fragment with default behavior, using a sequence of row elements to represent the table:

SELECT XMLROW(C1, C2) **FROM** T1

```
<row><C1>1</C1><C2>2</C2></row>
<row><C2>2</C2></row>
<row><C1>1</C1></row>
```

```
    4 record(s) selected.
```

- *Example 2*: The following example shows an XMLRow query and output fragment with attribute centric mapping. Instead of appearing as nested elements as in the previous example, relational data is mapped to element attributes:

SELECT XMLROW(C1, C2 **OPTION AS ATTRIBUTES**) **FROM** T1

```
<row C1="1" C2="2"/>
<row C2="2"/>
<row C1="1"/>
```

```
    4 record(s) selected.
```

- *Example 3*: The following example shows an XMLRow query and output fragment with the default <row> element replaced by <entry>. Columns C1 and C2 are returned as <column1> and <column2> elements, and the total of C1 and C2 is returned inside a <total> element:
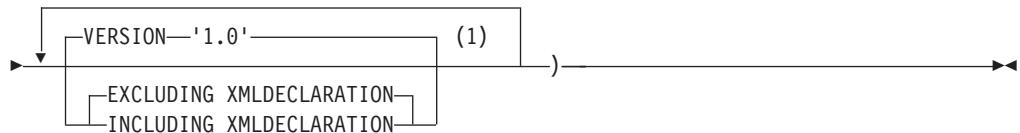
SELECT XMLROW(
    C1 **AS** "column1", C2 **AS** "column2",
    C1+C2 **AS** "total" **OPTION ROW** "entry")
FROM T1

```
<entry><column1>1</column1><column2>2</column2><total>3</total></entry>
<entry><column2>2</column2></entry>
<entry><column1>1</column1></entry>
```

```
    4 record(s) selected.
```
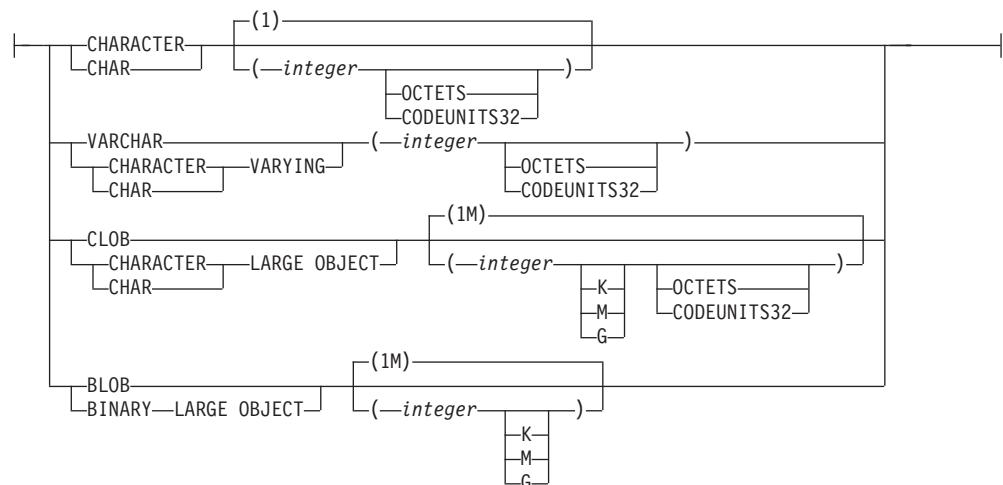
XMLSERIALIZE

The XMLSERIALIZE function returns a serialized XML value of the specified data type generated from the *XML-expression* argument.

```
►►─ XMLSERIALIZE ─(─┬─────────────┬─ XML-expression ─ AS ─┤ data-type ├ ──────────►
                    └─ CONTENT ───┘


   ┌─────────────────────────────────┐         (1)
   ▼ ┬─ VERSION ─ '1.0' ─────────────┬┴──────────── )───────────────────────────►◄
     ├─ EXCLUDING XMLDECLARATION ────┤
     └─ INCLUDING XMLDECLARATION ────┘
```

data-type:

```
                        (1)
├─┬─ CHARACTER ─┬─┬──────────────────────────────────────┬────────────────────────┤
  └─ CHAR ──────┘ └─ ( ─ integer ─┬──────────────┬─ ) ──┘
                                  ├─ OCTETS ──────┤
                                  └─ CODEUNITS32 ─┘
    ┌─ VARCHAR ─────────────────────┐
    ├─ CHARACTER ─┬─ VARYING ───────┤── ( ─ integer ─┬──────────────┬─ ) ─┤
    └─ CHAR ──────┘                                  ├─ OCTETS ──────┤
                                                     └─ CODEUNITS32 ─┘
                                   (1M)
    ┌─ CLOB ────────────────────────────────┐
    ├─ CHARACTER ─┬─ LARGE OBJECT ─┐─ ( ─ integer ─┬─────┬─┬──────────────┬─ ) ─┤
    └─ CHAR ──────┘                               ├─ K ─┤ ├─ OCTETS ──────┤
                                                  ├─ M ─┤ └─ CODEUNITS32 ─┘
                                                  └─ G ─┘
                                   (1M)
    ┌─ BLOB ────────────────────────┐
    └─ BINARY ─ LARGE OBJECT ───────┘─ ( ─ integer ─┬─────┬─ ) ─┤
                                                    ├─ K ─┤
                                                    ├─ M ─┤
                                                    └─ G ─┘
```

Notes:

1   The same clause must not be specified more than once.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

CONTENT
> Specifies that any XML value can be specified and the result of the serialization is based on this input value.

XML-expression
> Specifies an expression that returns a value of data type XML. The XML sequence value must not contain an item that is an attribute node (SQLSTATE 2200W). This is the input to the serialization process.

AS *data-type*
> Specifies the result type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the serialized output (SQLSTATE 22001).

VERSION '1.0'
> Specifies the XML version of the serialized value. The only version supported is '1.0' which must be specified as a string constant (SQLSTATE 42815).

EXCLUDING XMLDECLARATION or INCLUDING XMLDECLARATION
> Specifies whether an XML declaration is included in the result. The default is EXCLUDING XMLDECLARATION.

> **EXCLUDING XMLDECLARATION**
>> Specifies that an XML declaration is not included in the result.

> **INCLUDING XMLDECLARATION**
>> Specifies that an XML declaration is included in the result. The XML declaration is the string '<?xml version="1.0" encoding="UTF-8"?>'.

AS *data-type*
> Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001).

> If either the *xml-document* argument or the *xsl-stylesheet* argument is null, the result will be null.

> Code page conversion might occur when storing any of the previously mentioned documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

The result has the data type specified by the user. An XML sequence is effectively converted to have a single document node by applying XMLDOCUMENT to *XML-expression* before serializing the resulting XML nodes. If the result of *XML-expression* can be null, the result can be null; if the result of *XML-expression* is null, the result is the null value.

Notes

- **Encoding in the serialized result**: The serialized result is encoded with UTF-8. If XMLSERIALIZE is used with a character data type, and the INCLUDING XMLDECLARATION clause is specified, the resulting character string containing serialized XML might have an XML encoding declaration that does not match the code page of the character string. Following serialization, which uses UTF-8 encoding, the character string that is returned from the server to the client is converted to the code page of the client, and that code page might be different from UTF-8.

  Therefore, applications should avoid direct use of XMLSERIALIZE INCLUDING XMLDECLARATION that return character string types and should retrieve XML values directly into host variables to maintain the match between the external code page and the encoding in the XML declaration. If XMLSERIALIZE must be used in this situation, a BLOB type should be specified to avoid code page conversion.

- **Syntax alternative**: XML2CLOB(*XML-expression*) can be specified in place of XMLSERIALIZE(*XML-expression* AS CLOB(2G)). It is supported only for compatibility with previous DB2 releases.

XMLTEXT

The XMLTEXT function returns an XML value with a single XQuery text node having the input argument as the content.

►►—XMLTEXT—(—*string-expression*—)———————————————————————►◄

The schema is SYSIBM. The function name cannot be specified as a qualified name.

string-expression
An expression whose value has a character string type: CHAR, VARCHAR or CLOB.

The data type of the result is XML. If the result of *string-expression* can be null, the result can be null; if the input value is null, the result is the null value. If the result of *string-expression* is an empty string, the result value is an empty text node.

Examples

- *Example 1*: Create a simple XMLTEXT query.

```
VALUES(
  XMLTEXT(
    'The stock symbol for Johnson&Johnson is JNJ.'
  )
)
```

This query produces the following serialized result:

```
1
--------------------------------------------------
The stock symbol for Johnson&amp;Johnson is JNJ.
```

Note that the '&' sign is mapped to '&' when a text node is serialized.

- *Example 2*: Use XMLTEXT with XMLAGG to construct mixed content. Suppose that the content of table T is as follows:

```
seqno  plaintext
                                                         emphtext
------ ----------------------------------------------------------
-------------
1      This query shows how to construct
 mixed content
2        using XMLAGG and XMLTEXT. Without
 XMLTEXT
3      XMLAGG will not have text nodes to group with other nodes,
 mixed content
       therefore, cannot generate
  SELECT
XMLELEMENT(
    NAME "para", XMLAGG(
      XMLCONCAT(
        XMLTEXT(
          PLAINTEXT
        ),
        XMLELEMENT(
          NAME "emphasis", EMPHTEXT
        )
      )
      ORDER BY SEQNO
    ), '.'
  ) AS "result"
  FROM T
```

This query produces the following result:
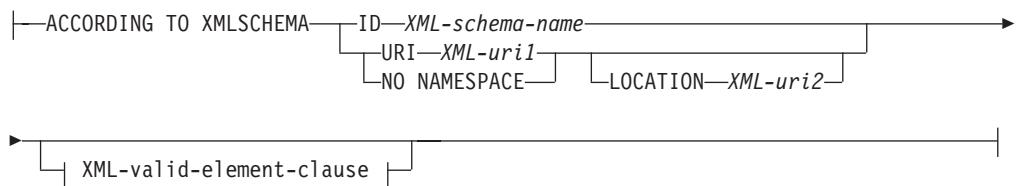
```
result
--------------------------------------------------------------------------------
<para>This query shows how to construct <emphasis>mixed content</emphasis>
using XMLAGG and XMLTEXT. Without <emphasis>XMLTEXT</emphasis>
, XMLAGG
will not have text nodes to group with other nodes, therefore, cannot
generate
<emphasis>mixed content</emphasis>.</para>
```

XMLVALIDATE
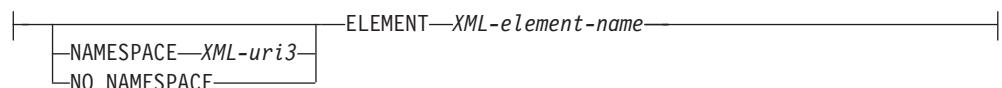
The XMLVALIDATE function returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.

```
►►─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:

```
├──┬──────────────────────┬──ELEMENT─XML-element-name──┤
   ├─NAMESPACE─XML-uri3──┤
   └─NO NAMESPACE────────┘
```

The schema is SYSIBM. The function name cannot be specified as a qualified name.

DOCUMENT
Specifies that the XML value resulting from *XML-expression* must be a well-formed XML document that conforms to XML Version 1.0 (SQLSTATE 2200M).

XML-expression
An expression that returns a value of data type XML. If *XML-expression* is an XML host variable or an implicitly or explicitly typed parameter marker, the function performs a validating parse that strips ignorable whitespace and the CURRENT IMPLICIT XMLPARSE OPTION setting is not considered.

XML-validate-according-to-clause
Specifies the information that is to be used when validating the input XML value.

ACCORDING TO XMLSCHEMA
Indicates that the XML schema information for validation is explicitly specified. If this clause is not included, the XML schema information must be provided in the content of the *XML-expression* value.

ID *XML-schema-name*
Specifies an SQL identifier for the XML schema that is to be used for validation. The name, including the implicit or explicit SQL schema qualifier, must uniquely identify an existing XML schema in the XML schema repository at the current server. If no XML schema by this name exists in the implicitly or explicitly specified SQL schema, an error is returned (SQLSTATE 42704).

URI *XML-uri1*
Specifies the target namespace URI of the XML schema that is to be used for validation. The value of *XML-uri1* specifies a URI as a character string constant that is not empty. The URI must be the target

namespace of a registered XML schema (SQLSTATE 4274A) and, if no LOCATION clause is specified, it must uniquely identify the registered XML schema (SQLSTATE 4274B).

NO NAMESPACE
Specifies that the XML schema for validation has no target namespace. The target namespace URI is equivalent to an empty character string that cannot be specified as an explicit target namespace URI.

LOCATION *XML-uri2*
Specifies the XML schema location URI of the XML schema that is to be used for validation. The value of *XML-uri2* specifies a URI as a character string constant that is not empty. The XML schema location URI, combined with the target namespace URI, must identify a registered XML schema (SQLSTATE 4274A), and there must be only one such XML schema registered (SQLSTATE 4274B).

XML-valid-element-clause
Specifies that the XML value in *XML-expression* must have the specified element name as the root element of the XML document.

NAMESPACE *XML-uri3* **or NO NAMESPACE**
Specifies the target namespace for the element that is to be validated. If neither clause is specified, the specified element is assumed to be in the same namespace as the target namespace of the registered XML schema that is to be used for validation.

NAMESPACE *XML-uri3*
Specifies the namespace URI for the element that is to be validated. The value of *XML-uri3* specifies a URI as a character string constant that is not empty. This can be used when the registered XML schema that is to be used for validation has more than one namespace.

NO NAMESPACE
Specifies that the element for validation has no target namespace. The target namespace URI is equivalent to an empty character string which cannot be specified as an explicit target namespace URI.

ELEMENT *xml-element-name*
Specifies the name of a global element in the XML schema that is to be used for validation. The specified element, with implicit or explicit namespace, must match the root element of the value of *XML-expression* (SQLSTATE 22535 or 22536).

The data type of the result is XML. If the value of *XML-expression* can be null, the result can be null; if the value of *XML-expression* is null, the result is the null value.

The XML validation process is performed on a serialized XML value. Because XMLVALIDATE is invoked with an argument of type XML, this value is automatically serialized before validation processing with the follow two exceptions.

- If the argument to XMLVALIDATE is an XML host variable or an implicitly or explicitly typed parameter marker, then a validating parse operation is performed on the input value (no implicit non-validating parse is performed and CURRENT IMPLICIT XMLPARSE OPTION setting is not considered).
- If the argument to XMLVALIDATE is an XMLPARSE invocation using the option PRESERVE WHITESPACE, then the XML parsing and XML validation of the document may be combined into a single validating parse operation.

If an XML value has previously been validated, the annotated type information from the previous validation is removed by the serialization process. However, any default values and entity expansions from the previous validation remain unchanged. If validation is successful, all ignorable whitespace characters are stripped from the result.

To validate a document whose root element does not have a namespace, an xsi:noNamespaceSchemaLocation attribute must be present on the root element.

Notes

- **Determining the XML schema**: The XML schema can be either specified explicitly with the ACCORDING TO XMLSCHEMA clause as part of the XMLVALIDATE invocation, or determined implicitly from the XML schema location information in the input XML value. The explicit or implicit XML schema information must identify an XML schema registered in the XML schema repository (SQLSTATE 42704, 4274A, or 22532), and there must be only one such registered XML schema (SQLSTATE 4274B or 22533).

  If an XML schema for validation is explicitly specified with the ACCORDING TO XMLSCHEMA clause, the schema location information specified in the input XML value is ignored.

  If the XML schema information is not specified with the ACCORDING TO XMLSCHEMA clause, the input XML value must contain XML schema location information (SQLSTATE 2200M). The schema location information in the input XML value, a namespace name, and a schema location specifies the XML schema document in the XML schema repository used for validation.

- **XML schema authorization**: The XML schema used for validation must be registered in the XML schema repository before use. The privileges held by the authorization ID of the statement must include at least one of the following authorities:
  - USAGE privilege on the XML schema that is to be used during validation
  - DBADM authority

- **Using a maxOccurs attribute value that is greater than 5000 in XML schemas:** In DB2 Version 9.7 Fix Pack 1 and later, if an XML schema that is registered in the XSR uses the maxOccurs attribute where the value is greater than 5000, the maxOccurs attribute value is treated as if you specified "unbounded". Because document elements that have a maxOccurs attribute value that is greater than 5000 are processed as if you specified "unbounded", an XML document might pass validation when you use the XMLVALIDATE function even if the number of occurrences of an element exceeds the maximum according to the XML schema that you used to validate the document.

  If you use an XML schema that defines an element that has a maxOccurs attribute value that is greater than 5000 and you want to reject XML documents that have a maxOccurs attribute value greater than 5000, you can define a trigger or procedure to check for that condition. In the trigger or procedure, use an XPath expression to count the number of occurrences of the element and return an error if the number of elements exceeds the maxOccurs attribute value.

  For example, the following trigger ensures that a document never has more than 6500 phone elements:

```
CREATE TRIGGER CUST_INSERT
  AFTER INSERT ON CUSTOMER
  REFERENCING NEW AS NEWROW
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  SELECT CASE WHEN X <= 6500 THEN 'OK - Do Nothing'
             ELSE RAISE_ERROR('75000', 'TooManyPhones') END
```

```
    FROM (
      SELECT XMLCAST(XMLQUERY('$INFO/customerinfo/count(phone)') AS INTEGER) AS X
      FROM CUSTOMER
      WHERE CUSTOMER.CID = NEWROW.CID );
  END
```

Examples

- *Example 1*: Validate using the XML schema identified by the XML schema hint in the XML instance document.

```
INSERT INTO T1(XMLCOL)
  VALUES (XMLVALIDATE(?))
```

  Assume that the input parameter marker is bound to an XML value that contains the XML schema information.

```
<po:order
   xmlns:po="http://my.world.com"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://my.world.com http://my.world.com/world.xsd" >
...
</po:order>
```

  Further, assume that the XML schema that is associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository.

  Based on these assumptions, the input XML value will be validated according to that XML schema.

- *Example 2*: Validate using the XML schema identified by the SQL name PODOCS.WORLDPO.

```
INSERT INTO T1(XMLCOL)
  VALUES (
    XMLVALIDATE(
      ? ACCORDING TO XMLSCHEMA ID PODOCS.WORLDPO
    )
  )
```

  Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML repository, the input XML value will be validated according to that XML schema.

- *Example 3*: Validate a specified element of the XML value.

```
INSERT INTO T1(XMLCOL)
  VALUES (
    XMLVALIDATE(
      ? ACCORDING TO XMLSCHEMA ID FOO.WORLDPO
      NAMESPACE 'http://my.world.com/Mary'
      ELEMENT "po"
    )
  )
```

  Assuming that the XML schema that is associated with SQL name FOO.WORLDPO is found in the XML repository, the XML schema will be validated against the element "po", whose namespace is 'http://my.world.com/Mary'.

- *Example 4*: XML schema is identified by target namespace and schema location.

```
INSERT INTO T1(XMLCOL)
  VALUES (
    XMLVALIDATE(
      ? ACCORDING TO XMLSCHEMA URI 'http://my.world.com'
      LOCATION 'http://my.world.com/world.xsd'
    )
  )
```

Assuming that an XML schema associated with the target namespace "http://my.world.com" and by schemaLocation hint "http://my.world.com/world.xsd" is found in the XML schema repository, the input XML value will be validated according to that schema.

XMLXSROBJECTID

The XMLXSROBJECTID function returns the XSR object identifier of the XML schema used to validate the XML document specified in the argument. The XSR object identifier is returned as a BIGINT value and provides the key to a single row in SYSCAT.XSROBJECTS.

►►──XMLXSROBJECTID──(──*xml-value-expression*──)──────────────────────►◄

The schema is SYSIBM.

xml-value-expression
> Specifies an expression that results in a value with a data type of XML. The resulting XML value must be an XML sequence with a single item that is an XML document or the null value (SQLSTATE 42815). If the argument is null, the function returns null. If *xml-value-expression* does not specify a validated XML document, the function returns 0.

Notes

- The XML schema corresponding to an XSR object ID returned by the function might no longer exist, because an XML schema can be dropped without affecting XML values that were validated using the XML schema. Therefore, queries that use the XSR object ID to fetch further XML schema information from the catalog views might return an empty result set.
- Applications can use the XSR object identifier to retrieve additional information about the XML schema. For example, the XSR object identifier can be used to return the individual XML schema documents that make up a registered XML schema from SYSCAT.SYSXSROBJECTCOMPONENTS, and the hierarchy of XML schema documents in the XML schema from SYSCAT.XSROBJECTHIERARCHIES.

Examples

- *Example 1*: Retrieve the XML schema identifier for the XML document XMLDOC stored in the table MYTABLE.
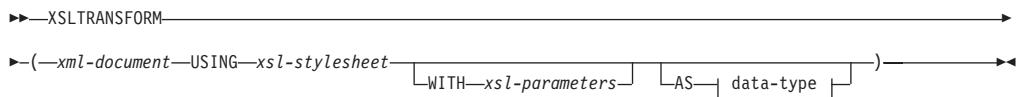
  ```
  SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
  ```
- *Example 2*: Retrieve the XML schema documents associated with the XML document that has a specific ID (in this case where DOCKEY = 1) in the table MYTABLE, including the hierarchy of the XML schema documents that make up the XML schema.
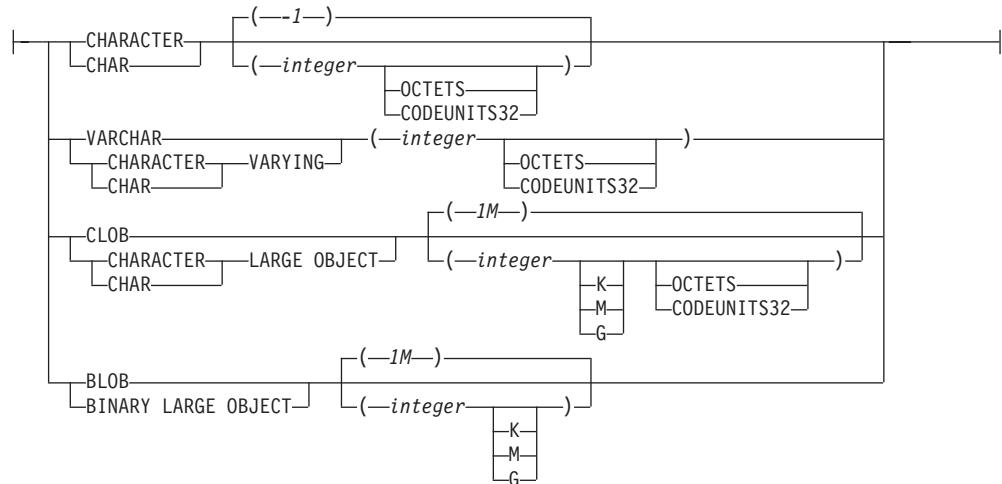
  ```
  SELECT H.HTYPE, C.TARGETNAMESPACE, C.COMPONENT
  FROM SYSCAT.XSROBJECTCOMPONENTS C, SYSCAT.XSROBJECTHIERARCHIES H
  WHERE C.OBJECTID =
    (SELECT XMLXSROBJECTID(XMLDOC) FROM MYTABLE
     WHERE DOCKEY = 1)
  AND C.OBJECTID = H.OBJECTID
  ```

XSLTRANSFORM

Use XSLTRANSFORM to convert XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

```
►►──XSLTRANSFORM────────────────────────────────────────────────────────►

►──(──xml-document──USING──xsl-stylesheet─────────────────────────)──────►◄
                                          └─WITH──xsl-parameters─┘ └─AS─┤ data-type ├─┘
```

data-type:

```
                              (──1──)
├─┬─┬─CHARACTER─┬──────────────────────────────────────┬──────────────────────────────┤
│ │ └─CHAR──────┘ └─(──integer─┬──────────────┬──)─┘                                    │
│ │                            ├─OCTETS───────┤                                         │
│ │                            └─CODEUNITS32──┘                                         │
│ ├─VARCHAR─────────────┬──(──integer─┬──────────────┬──)───────┤                      │
│ │ ┌─CHARACTER──┬─VARYING─┘           ├─OCTETS───────┤                                 │
│ │ └─CHAR───────┘                     └─CODEUNITS32──┘                                 │
│ │                              (──1M──)                                               │
│ ├─CLOB────────────────┬──────────────────────────────────────────┤                  │
│ │ ┌─CHARACTER──┬─LARGE OBJECT─┘ └─(──integer─┬────┬─┬──────────────┬──)─┘             │
│ │ └─CHAR───────┘                             ├─K──┤ ├─OCTETS───────┤                  │
│ │                                            ├─M──┤ └─CODEUNITS32──┘                  │
│ │                                            └─G──┘                                   │
│ │                              (──1M──)                                               │
│ └─BLOB──────────────────┬──────────────────────────────┤                            │
│   └─BINARY LARGE OBJECT─┘ └─(──integer─┬────┬──)─┘                                    │
│                                        ├─K──┤                                         │
│                                        ├─M──┤                                         │
│                                        └─G──┘                                         │
```

The schema is SYSIBM. This function cannot be specified as a qualified name.

The XSLTRANSFORM function transforms an XML document into a different data format. The data can be transformed into any form possible for the XSLT processor, including but not limited to XML, HTML, or plain text.

All paths used by XSLTRANSFORM are internal to the database system. This command cannot currently be used directly with files or stylesheets residing in an external file system.

xml-document
> An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is the document that is transformed using the XSL style sheet specified in *xsl-stylesheet*.

> The XML document must at minimum be single-rooted and well-formed.

xsl-stylesheet
> An expression that returns a well-formed XML document with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. The document is an XSL style sheet that conforms to the W3C XSLT Version 1.0 Recommendation. Style sheets incorporating XQUERY statements or the `xsl:include` declaration are not supported. This stylesheet is applied to transform the value specified in *xml-document*.

xsl-parameters
> An expression that returns a well-formed XML document or null with a data type of XML, CHAR, VARCHAR, CLOB, or BLOB. This is a document that

provides parameter values to the XSL stylesheet specified in *xsl-stylesheet*. The value of the parameter can be specified as an attribute, or as a text node.

The syntax of the parameter document is as follows:

```
<params xmlns="http://www.ibm.com/XSLTransformParameters">
<param name="..." value="..."/>
<param name="...">enter value here</param>
  ...
</params>
```

The stylesheet document must have xsl:param element(s) in it with name attribute values that match the ones specified in the parameter document.

AS *data-type*
Specifies the result data type. The implicit or explicit length attribute of the specified result data type must be sufficient to contain the transformed output (SQLSTATE 22001). The default result data type is CLOB(2G OCTETS) except in a Unicode database where the string units of the environment is set to CODEUNITS32 when the default is CLOB(536870911 CODEUNITS32).

If either the *xml-document* argument or the *xsl-stylesheet* argument is null, the result will be null.

Code page conversion might occur when storing any of the previously mentioned documents in a CHAR, VARCHAR, or CLOB column, which might result in a character loss.

Example

This example illustrates how to use XSLT as a formatting engine. To get set up, first insert the following two example documents 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"/>
<xsl:param name="showUniversity"/>
<xsl:template match="students">
                <html>
                    <head/>
                        <body>
                                <h1><xsl:value-of select="$headline"/></h1>
                                <table border="1">
                                        <th>
                                            <tr>
                                    <td width="80">StudentID</td>
                                    <td width="200">First Name</td>
                                    <td width="200">Last Name</td>
                                    <td width="50">Age</td>
                                    <xsl:choose>
        <xsl:when test="$showUniversity ='true'">
                                                    <td width="200">University</td>
                                        </xsl:when>
        </xsl:choose>
                                </tr>
                                    </th>
                                    <xsl:apply-templates/>
```

```
                                        </table>
                                    </body>
                                </html>
                            </xsl:template>
                                <xsl:template match="student">
                                <tr>
                        <td><xsl:value-of select="@studentID"/></td>
                        <td><xsl:value-of select="@firstName"/></td>
                        <td><xsl:value-of select="@lastName"/></td>
                        <td><xsl:value-of select="@age"/></td>
                        <xsl:choose>
                            <xsl:when test="$showUniversity = 'true' ">
                                <td><xsl:value-of select="@university"/></td>
                            </xsl:when>
                        </xsl:choose>
                                    </tr>
                                </xsl:template>
</xsl:stylesheet>'
);
```

Next, call the XSLTRANSFORM function to convert the XML data into HTML and
display it.

SELECT XSLTRANSFORM (XML_DOC **USING** XSL_DOC **AS CLOB**(1M)) **FROM** XML_TAB;

The result is this document:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1></h1>
<table border="1">
<th>
<tr>
<td width="80">StudentID</td>
<td width="200">First Name</td>
<td width="200">Last Name</td>
<td width="50">Age</td>
</tr>
</th>
 <tr>
<td>1</td>
<td>Steffen</td><td>Siegmund</td>
<td>23</td>
</tr>
            </table>
</body>
</html>
```

In this example, the output is HTML and the parameters influence only what
HTML is produced and what data is brought over to it. As such it illustrates the
use of XSLT as a formatting engine for end-user output.

Usage note

There are many methods you can use to transform XML documents including the
XSLTRANSFORM function, an XQuery update expression, and XSLT processing by
an external application server. For documents stored in an XML column of a
database table, many transformations can be performed more efficiently by using
an XQuery update expression rather than with XSLT because XSLT always requires
parsing of the XML documents that are being transformed. If you decide to
transform XML documents with XSLT, you should make careful decisions about

whether to transform the document in the database or in an application server.

YEAR

The YEAR function returns the year part of a value.

►►──YEAR──(──*expression*──)──►◄

The schema is SYSIBM.

expression
> An expression that returns a value of one of the following built-in data types: DATE, TIMESTAMP, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is not a CLOB. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:
- If the argument is a DATE, TIMESTAMP, or valid string representation of a date or timestamp:
  - The result is the year part of the value, which is an integer between 1 and 9999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between -9999 and 9999. A nonzero result has the same sign as the argument.

Examples

- *Example 1*: Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

  ```
  SELECT * FROM PROJECT
    WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
  ```

- *Example 2*: Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

  ```
  SELECT * FROM PROJECT
    WHERE YEAR(PRENDATE - PRSTDATE) < 1
  ```

Table functions

Table functions return columns of a table, resembling a table created through a simple CREATE TABLE statement.

A table function can be used only in the FROM clause of a statement.

Table functions can be qualified with a schema name.

BASE_TABLE

The BASE_TABLE function returns both the object name and schema name of the object found after any alias chains have been resolved.

```
►►──BASE_TABLE──(──objectschema──,──objectname──)──────────────────────────►◄
```

The schema is SYSPROC.

The specified objectname (and objectschema) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name and the unqualified name of the starting point are returned. The function returns a single row table consisting of the following columns:

Table 82. Information returned by the BASE_TABLE function

Column name	Data type	Description
BASESCHEMA	VARCHAR(128)	Schema name of the object found after any alias chains have been resolved. Matches objectschema if no matching alias was found.
BASENAME	VARCHAR(128)	Unqualified name of the object found after any alias chains have been resolved. Matches objectname if no matching alias was found. The name may identify a table, a view, or an undefined object.

objectschema
> A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

objectname
> A character expression representing the unqualified name to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 bytes.

Note: The BASE_TABLE table function improves performance in partitioned database configurations by avoiding the unnecessary communication that occurs between the coordinator partition and catalog partition when using the TABLE_SCHEMA and TABLE_NAME scalar functions.

Example

The following statement using the TABLE_SCHEMA and TABLE_NAME functions is written as:

```
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES
WHERE OWNER = TABLE_SCHEMA(:H00031 ,:H00032 )
AND TABNAME = TABLE_NAME(:H00031 ,:H00032 )
```
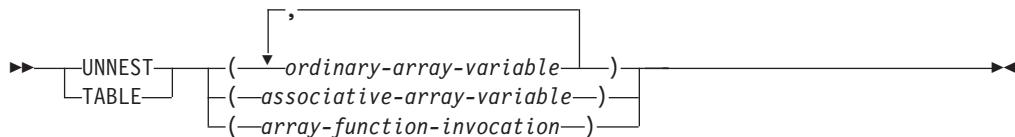
The equivalent statement using the BASE_TABLE function can be written as:

```
SELECT COLCOUNT INTO :H00030
FROM SYSCAT.TABLES A, TABLE(SYSPROC.BASE_TABLE(:H00032, :H00031)) AS B
WHERE A.OWNER = B.BASESCHEMA
AND A.TABNAME = B.BASENAME
```

UNNEST

The UNNEST function returns a result table that includes a row for each element of the specified array. If there are multiple ordinary array arguments specified, the number of rows will match the array with the largest cardinality.

```
>>─┬─UNNEST─┬─┬─(──┬─ordinary-array-variable──┬─)────┬──────────────────><
   └─TABLE──┘ │    └──────────,◄───────────────┘     │
             ├─(──associative-array-variable──)─┤
             └─(──array-function-invocation──)──┘
```

The schema is SYSIBM.

ordinary-array-variable
  An SQL variable, SQL parameter, or global variable of an ordinary array type, or a CAST specification of a parameter marker to an ordinary array type.

associative-array-variable
  An SQL variable, SQL parameter, or global variable of an associative array type, or a CAST specification of a parameter marker to an associative array type.

array-function-invocation
  A function invocation which resolves to a function that returns an ordinary or an associative array type.

Names for the result columns produced by the UNNEST function can be provided as part of the *correlation-clause* of the *collection-derived-table* clause.

The UNNEST function can only be used in a *collection-derived-table* clause in a context where arrays are supported (SQLSTATE 42887).

The result table depends on the input arguments.
* If a single ordinary array argument or an array-function-invocation returning an ordinary array is specified:
  – If the array element is not a row data type, the result is a single column table with a column data type that matches the array element data type.
  – If the array element is a row data type, the result is a table with one column for each row field in the element data type. The result table column data types match the corresponding array element row field data types.
* If more than one ordinary array argument is specified and none of the array elements have a row data type, the first array provides the first column in the result table, the second array provides the second column, and so on. The data type of each column matches the data type of the array elements of the corresponding array argument. If the cardinalities of the arrays are not identical, the cardinality of the resulting table is the same as the array with the largest cardinality. The column values in the table are set to the null value for all rows whose array index value is greater than the cardinality of the corresponding array. In other words, if each array is viewed as a table with two columns, one for the array indexes and one for the data, then UNNEST performs an OUTER JOIN among the arrays, using equality on the array indexes as a join predicate.
* If a single associative array argument or an array-function-invocation returning an associate array is specified:

– If the array element is not a row data type, the result is a table with 2 columns where the first column data type matches the array index data type and the second column data type matches the array element data type.
– If the array element is a row data type, the result is a table with one more column than the number of fields in the row data type, where the first column data type matches the array index data type and the remaining column data types match the array element row field data types.
- An error is returned (SQLSTATE 42884):
  – If more than one associative array argument is specified.
  – If more than one array argument is specified and at least one of the arrays has a element data type that is a row type.
  – If both ordinary array arguments and associative array arguments are specified.

This special table function is only used in *collection-derived-table* of *table-reference* in a FROM clause.

If more than one array is provided and at least one of the arguments is an associative array, an error is returned (SQLSTATE 42884).

If the WITH ORDINALITY clause is used when unnesting an associative array, an error is returned (SQLSTATE 428HT).

Examples

- *Example 1*: Assume the ordinary array variable RECENT_CALLS of array type PHONENUMBERS contains only the three element values 9055553907, 4165554213, and 4085553678.The following query:

```
SELECT T.ID, T.NUM
FROM UNNEST(RECENT_CALLS) WITH ORDINALITY AS T(NUM, ID)
```

returns a table formatted as follows:

```
ID    NUM
----------
1     9055553907
2     4165554213
3     4085553678
```

- *Example 2*: Return the list of personal phone numbers from the array variable PHONELIST of array type PERSONAL_PHONENUMBERS along with the index string values The following query:

```
SELECT T.ID, T.PHONE
FROM UNNEST(PHONELIST) AS T(ID, PHONE)
```
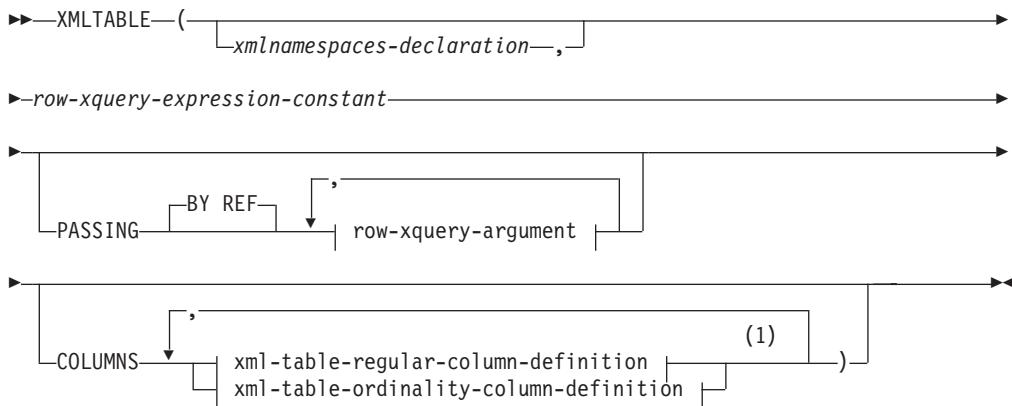
returns a table formatted as follows:

```
ID     PHONE
----------------
Home   4163053745
Work   4163053746
Mom    4164789683
```
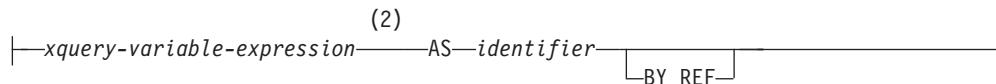
XMLTABLE
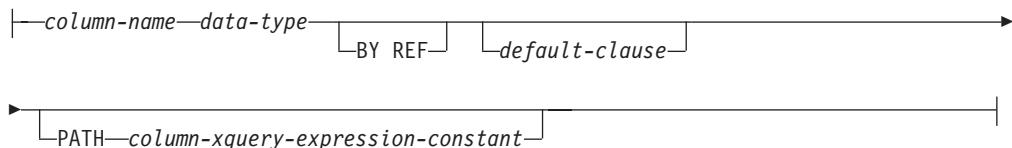
The XMLTABLE function returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.

▶▶──XMLTABLE──(─┬──────────────────────────────────┬──────────────────────▶
　　　　　　　　　 └─*xmlnamespaces-declaration*──,─┘

▶──*row-xquery-expression-constant*──────────────────────────────────────▶

▶──┬──┬──▶
　　│　　　　　　　┌─BY REF─┐　　┌────────,────────┐　　　　　　　　　│
　　└─PASSING──┴────────┴──▼──*row-xquery-argument*─┴─────────────┘

▶──┬──┬──▶◀
　　│　　　　　　　　┌─────────────,─────────────┐　　　(1)　　　│
　　└─COLUMNS──▼─┬─xml-table-regular-column-definition─┬────────┴──)─┘
　　　　　　　　　　 └─xml-table-ordinality-column-definition─┘

row-xquery-argument:

　　　　　　　　　　　　　　　　　　　　(2)
├──*xquery-variable-expression*────AS──*identifier*──┬────────┬──┤
　　　　　　　　　　　　　　　　　　　　　　　　　　　└─BY REF─┘

xml-table-regular-column-definition:

├──*column-name*──*data-type*──┬────────┬──┬─────────────────┬──▶
　　　　　　　　　　　　　　　　　└─BY REF─┘　└─*default-clause*─┘

▶──┬──┬──┤
　　└─PATH──*column-xquery-expression-constant*─┘

xml-table-ordinality-column-definition:

├──*column-name*──FOR ORDINALITY────────────────────────────────┤

Notes:

1   The xml-table-ordinality-column-definition clause must not be specified more than once (SQLSTATE 42614).

2   The data type of the expression cannot be DECFLOAT.

The schema is SYSIBM. The function name cannot be specified as a qualified name.

xmlnamespaces-declaration
　　Specifies one or more XML namespace declarations that become part of the static context of the *row-xquery-expression-constant* and the *column-xquery-expression-constant*. The set of statically known namespaces for XQuery expressions which are arguments of XMLTABLE is the combination of the pre-established set of statically known namespaces and the namespace declarations specified in this clause. The XQuery prolog within an XQuery expression may override these namespaces.

If *xmlnamespaces-declaration* is not specified, only the pre-established set of statically known namespaces apply to the 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').

Example

List as a table result the purchase order items for orders with a status of 'NEW'.

```
SELECT U."PO ID", U."Part #", U."Product Name",
       U."Quantity", U."Price", U."Order Date"
  FROM PURCHASEORDER P,
     XMLTABLE('$po/PurchaseOrder/item' PASSING P.PORDER AS "po"
              COLUMNS "PO ID"         INTEGER       PATH '../@PoNum',
                      "Part #"        CHAR(10)      PATH 'partid',
                      "Product Name"  VARCHAR(50)   PATH 'name',
```
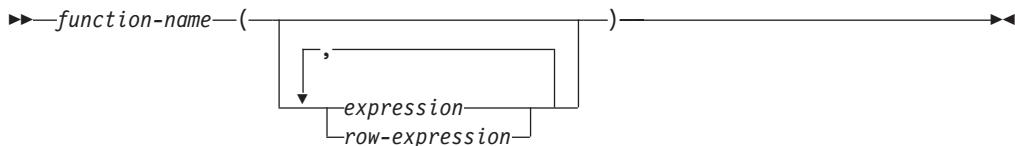
```
                                "Quantity"    INTEGER       PATH 'quantity',
                                "Price"       DECIMAL(9,2)  PATH 'price',
                                "Order Date"  DATE          PATH '../@OrderDate'
                ) AS U
       WHERE P.STATUS = 'Unshipped'
```

User-defined functions

User-defined functions (UDFs) are extensions or additions to the existing built-in functions of the SQL language.

A user-defined function can be a scalar function, which returns a single value each time it is called; an aggregate function, which is passed a set of like values and returns a single value for the set; a row function, which returns one row; or a table function, which returns a table.

A number of user-defined functions are provided in the SYSFUN and SYSPROC schemas.

A UDF can be an aggregate function only if it is sourced on an existing aggregate function. A UDF is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any). A user-defined column or scalar function registered with the database can be referenced in the same contexts in which any built-in function can appear. A user-defined row function can be referenced only implicitly when registered as a transform function for a user-defined type. A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT statement.

Function arguments must correspond in number and position to the parameters specified for the user-defined function when it was registered with the database. In addition, the arguments must be of data types that are promotable to the data types of the corresponding defined parameters.

The result of the function is specified in the RETURNS clause. The RETURNS clause, defined when the UDF was registered, determines whether or not a function is a table function. If the RETURNS NULL ON NULL INPUT clause is specified (or defaulted to) when the function is registered, the result is null if any argument is null. In the case of table functions, this is interpreted to mean a return table with no rows (that is, an empty table).

See "Row expressions" for more information about rules and row data types.

Following are some examples of user-defined functions:

- A scalar UDF called ADDRESS extracts the home address from resumes stored in script format. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000) value:

```
SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
  WHERE RESUME_FORMAT = 'SCRIPT'
```

- Table T2 has a numeric column A. Invoking the scalar UDF called ADDRESS from the previous example:

  ```
  SELECT ADDRESS(A) FROM T2
  ```

  raises an error (SQLSTATE 42884), because no function with a matching name and with a parameter that is promotable from the argument exists.
- A table UDF called WHO returns information about the sessions on the server machine that were active at the time that the statement is executed. The WHO function is invoked from within a FROM clause that includes the keyword TABLE and a mandatory correlation variable. The column names of the WHO() table were defined in the CREATE FUNCTION statement.

  ```
  SELECT ID, START_DATE, ORIG_MACHINE
    FROM TABLE( WHO() ) AS QQ
    WHERE START_DATE LIKE 'MAY%'
  ```

User-defined functions

Chapter 5. Built-in procedures

A procedure is an application program that can be started through the SQL CALL statement. The procedure is specified by a procedure name, which may be followed by arguments that are enclosed within parentheses. *Built-in procedures* are procedures provided with the database manager.

This topic lists the supported built-in procedures for managing XML schema repository (XSR) objects. See Table 83.

There are additional built-in procedures documented under the following headings:
- ADMIN_CMD procedure and associated SQL routines
- Administrative task scheduler routines and views
- Audit routines and procedures
- Automatic maintenance SQL routines and views
- Common SQL API stored procedures
- Explain routines
- Monitor routines
- Snapshot SQL routines and views
- SQL procedures SQL routines
- Stepwise redistribute SQL routines
- Storage management tool SQL routines
- Text search SQL routines
- Workload management routines
- Miscellaneous SQL routines and views

For details about these additional built-in procedures, see "Supported built-in SQL routines and views" in *Administrative Routines and Views* .

Table 83. XML schema repository procedures

Function	Description
"XSR_ADDSCHEMADOC"	Add an XML schema document to an XML schema.
"XSR_COMPLETE" on page 775	Complete the XML schema registration process for an XML schema.
"XSR_DTD" on page 776	Register a document type declaration.
"XSR_EXTENTITY" on page 777	Register an external entity.
"XSR_REGISTER" on page 778	Register an XML schema.
"XSR_UPDATE" on page 780	Update an existing XML schema.

XSR_ADDSCHEMADOC

Each XML schema in the XML schema repository (XSR) can consist of one or more XML schema documents. Where an XML schema consists of multiple documents, the XSR_ADDSCHEMADOC procedure is used to add every XML schema other than the primary XML schema document.

XSR_ADDSCHEMADOC

```
►►──XSR_ADDSCHEMADOC──(──rschema──,──name──,──schemalocation──,──content──,──────►

►──docproperty──)────────────────────────────────────────────────────────►◄
```

The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema
> An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
> An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER procedure, and XML schema registration cannot yet be completed. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation
> An input argument of type VARCHAR (1000), which can have a null value, that indicates the schema location of the primary XML schema document to which the XML schema document is being added. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content
> An input parameter of type BLOB (30M) that contains the content of the XML schema document being added. This argument cannot have a null value; an XML schema document must be supplied.

docproperty
> An input parameter of type BLOB (5M) that indicates the properties for the XML schema document being added. This parameter can have a null value; otherwise, the value is an XML document.

Example

```
CALL SYSPROC.XSR_ADDSCHEMADOC(
  'user1',
  'POschema',
  'http://myPOschema/address.xsd',
  :content_host_var,
  0)
```

XSR_COMPLETE

The XSR_COMPLETE procedure is the final procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR). An XML schema is not available for validation until the schema registration completes through a call to this procedure.

►►──XSR_COMPLETE──(──*rschema*──,──*name*──,──*schemaproperties*──,───────────────►

►──*isusedfordecomposition*──)───►◄

The schema is SYSPROC.

Authorization:

The authorization ID of the caller of the procedure must be the owner of the XSR object as recorded in the catalog view SYSCAT.XSROBJECTS.

rschema
> An input argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR, which is to be moved to the complete state. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
> An input argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema, for which a completion check is to be performed, is *rschema.name*. The XML schema name must already exist as a result of calling the XSR_REGISTER procedure, and XML schema registration cannot yet be completed. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemaproperties
> An input argument of type BLOB (5M) that specifies properties, if any, associated with the XML schema. The value for this argument is either the null value, if there are no associated properties, or an XML document representing the properties for the XML schema.

isusedfordecomposition
> An input parameter of type integer that indicates if an XML schema is to be used for decomposition. If an XML schema is to be used for decomposition, this value should be set to 1; otherwise, it should be set to zero.

Example

```
CALL SYSPROC.XSR_COMPLETE(
  'user1',
  'POschema',
  :schemaproperty_host_var,
  0)
```

XSR_DTD

The XSR_DTD procedure registers a document type declaration (DTD) with the XML schema repository (XSR).

►►──XSR_DTD──(─*rschema*─,─*name*─,─*systemid*─,─*publicid*─,─*content*─)──────────►◄

The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the following:
- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
> An input and output argument of type VARCHAR (128) that specifies the SQL schema for the DTD. The SQL schema is one part of the SQL identifier used to identify this DTD in the XSR. (The other part of the SQL identifier is supplied by the *name* argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name
> An input and output argument of type VARCHAR (128) that specifies the name of the DTD. The complete SQL identifier for the DTD is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a null value. When a null value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

systemid
> An input parameter of type VARCHAR (1000) that specifies the system identifier of the DTD. The system ID of the DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a null value. The system ID can be specified together with a public ID.

publicid
> An input parameter of type VARCHAR (1000) that specifies the public identifier of the DTD. The public ID of a DTD should match the uniform resource identifier of the DTD in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a null value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content
>   An input parameter of type BLOB (30M) that contains the content of the DTD
>   document. This argument cannot have a null value.

Example

Register the DTD identified by the system ID *http://www.test.com/person.dtd* and
public ID *http://www.test.com/person*:

```
CALL SYSPROC.XSR_DTD ( 'MYDEPT' ,
    'PERSONDTD' ,
    'http://www.test.com/person.dtd' ,
    'http://www.test.com/person',
    :content_host_variable
    )
```

XSR_EXTENTITY

The XSR_EXTENTITY procedure registers an external entity with the XML schema
repository (XSR).

►►──XSR_EXTENTITY──(──*rschema*──,──*name*──,──*systemid*──,──*publicid*──,────────────►

►──*content*──)──►◄

The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the
following:
- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.
- CREATEIN privilege if the SQL schema exists.

rschema
>   An input and output argument of type VARCHAR (128) that specifies the SQL
>   schema for the external entity. The SQL schema is one part of the SQL
>   identifier used to identify this external entity in the XSR. (The other part of the
>   SQL identifier is supplied by the *name* argument.) This argument can have a
>   null value, which indicates that the default SQL schema, as defined in the
>   CURRENT SCHEMA special register, is used. Rules for valid characters and
>   delimiters that apply to any SQL identifier also apply to this argument.
>   Relational schemas that begin with the string 'SYS' must not be used for this
>   value. XSR objects will not experience name collisions with database objects
>   that exist outside of the XSR, because XSR objects occur in a different
>   namespace than objects outside of the XML schema repository.

name
>   An input and output argument of type VARCHAR (128) that specifies the
>   name of the external entity. The complete SQL identifier for the external entity
>   is *rschema.name* and should be unique among all objects in the XSR. This
>   argument accepts a null value. When a null value is provided for this
>   argument, a unique value is generated and stored within the XSR. Rules for
>   valid characters and delimiters that apply to any SQL identifier also apply to
>   this argument.

XSR_EXTENTITY

systemid
> An input parameter of type VARCHAR (1000) that specifies the system identifier of the external entity. The system ID of the external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the SYSTEM keyword, if used). This argument cannot have a null value. The system ID can be specified together with a public ID.

publicid
> An input parameter of type VARCHAR (1000) that specifies the public identifier of the external entity. The public ID of a external entity should match the uniform resource identifier of the external entity in the ENTITY declaration (as prefixed by the PUBLIC keyword, if used). This argument accepts a null value and should be used only if also specified in the DOCTYPE declaration of the XML instance document or in an ENTITY declaration.

content
> An input parameter of type BLOB (30M) that contains the content of the external entity document. This argument cannot have a null value.

Example

Register the external entities identified by the system identifiers *http://www.test.com/food/chocolate.txt* and *http://www.test.com/food/cookie.txt*:

```
CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
    'CHOCLATE' ,
    'http://www.test.com/food/chocolate.txt' ,
    NULL ,
    :content_of_chocolate.txt_as_a_host_variable
    )

CALL SYSPROC.XSR_EXTENTITY ( 'FOOD' ,
    'COOKIE' ,
    'http://www.test.com/food/cookie.txt' ,
    NULL ,
    :content_of_cookie.txt_as_a_host_variable
    )
```

XSR_REGISTER

The XSR_REGISTER procedure is the first procedure to be called as part of the XML schema registration process, which registers XML schemas with the XML schema repository (XSR).

►►──XSR_REGISTER──(─*rschema*─,─*name*─,─*schemalocation*─,─*content*─,──────────►

►─*docproperty*─)───►◄

The schema is SYSPROC.

Authorization

The authorization ID of the caller of the procedure must have at least one of the following authorities:
- DBADM authority.
- IMPLICIT_SCHEMA database authority if the SQL schema does not exist.

- CREATEIN privilege if the SQL schema exists.

rschema

An input and output argument of type VARCHAR (128) that specifies the SQL schema for the XML schema. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the name argument.) This argument can have a null value, which indicates that the default SQL schema, as defined in the CURRENT SCHEMA special register, is used. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument. Relational schemas that begin with the string 'SYS' must not be used for this value. XSR objects will not experience name collisions with database objects that exist outside of the XSR, because XSR objects occur in a different namespace than objects outside of the XML schema repository.

name

An input and output argument of type VARCHAR (128) that specifies the name of the XML schema. The complete SQL identifier for the XML schema is *rschema.name* and should be unique among all objects in the XSR. This argument accepts a null value. When a null value is provided for this argument, a unique value is generated and stored within the XSR. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

schemalocation

An input argument of type VARCHAR (1000), which can have a null value, that indicates the schema location of the primary XML schema document. This argument is the external name of the XML schema, that is, the primary document can be identified in the XML instance documents with the xsi:schemaLocation attribute.

content

An input parameter of type BLOB (30M) that contains the content of the primary XML schema document. This argument cannot have a null value; an XML schema document must be supplied.

docproperty

An input parameter of type BLOB (5M) that indicates the properties for the primary XML schema document. This parameter can have a null value; otherwise, the value is an XML document.

Examples

- *Example 1*: The following example shows how to call the XSR_REGISTER procedure from the command line:

```
CALL SYSPROC.XSR_REGISTER(
    'user1',
    'POschema',
    'http://myPOschema/PO.xsd',
    :content_host_var,
    :docproperty_host_var)
```

- *Example 2*: The following example shows how to call the XSR_REGISTER procedure from a Java application program:

```
stmt = con.prepareCall("CALL SYSPROC.XSR_REGISTER (?, ?, ?, ?, ?)");
String xsrObjectName = "myschema1";
String xmlSchemaLocation = "po.xsd";
stmt.setNull(1, java.sql.Types.VARCHAR);
stmt.setString(2, xsrObjectName);
stmt.setString(3, xmlSchemaLocation);
stmt.setBinaryStream(4, buffer, (int)length);
```

```
                    stmt.setNull(5, java.sql.Types.BLOB);
                    stmt.registerOutParameter(1, java.sql.Types.VARCHAR);
                    stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
                    stmt.execute();
```

XSR_UPDATE

The XSR_UPDATE procedure is used to evolve an existing XML schema in the XML schema repository (XSR). This enables you to modify or extend an existing XML schema so that it can be used to validate both already existing and newly inserted XML documents.

```
►►──XSR_UPDATE──(──rschema1──,──name1──,──rschema2──,──name2──,────────────────►

►──dropnewschema──)──────────────────────────────────────────────────────────►◄
```

The schema is SYSPROC.

The original XML schema and the new XML schema specified as arguments to XSR_UPDATE must both be registered and completed in the XSR before the procedure is called. These XML schemas must also be compatible. For details about the compatibility requirements see *Compatibility requirements for evolving an XML schema*.

Authorization

The privileges held by the authorization ID of the caller of the procedure must include at least one of the following:

*   DBADM authority.
*   SELECT privilege on the catalog views SYSCAT.XSROBJECTS and SYSCAT.XSROBJECTCOMPONENTS and one of the following sets of privileges:
    *   OWNER of the XML schema specified by the SQL schema *rschema1* and the object name *name1*
    *   ALTERIN privilege on the SQL schema specified by the *rschema1* argument and, if the *dropnewschema* argument is not equal to zero, DROPIN privilege on the SQL schema specified by the *rschema2* argument.

rschema1
> An input argument of type VARCHAR (128) that specifies the SQL schema for the original XML schema to be updated. The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name1* argument.) This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name1
> An input argument of type VARCHAR (128) that specifies the name of the original XML schema to be updated. The complete SQL identifier for the XML schema is *rschema1.name1* . This XML schema must already be registered and completed in the XSR. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

rschema2
> An input argument of type VARCHAR (128) that specifies the SQL schema for the new XML schema that will be used to update the original XML schema.

The SQL schema is one part of the SQL identifier used to identify this XML schema in the XSR. (The other part of the SQL identifier is supplied by the *name2* argument.) This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

name2
An input argument of type VARCHAR (128) that specifies the name of the new XML schema that will be used to update the original XML schema. The complete SQL identifier for the XML schema is *rschema2.name2*. This XML schema must already be registered and completed in the XSR. This argument cannot have a null value. Rules for valid characters and delimiters that apply to any SQL identifier also apply to this argument.

dropnewschema
An input parameter of type integer that indicates whether the new XML schema should be dropped after it is used to update the original XML schema. Setting this parameter to any nonzero value will cause the new XML schema to be dropped. This argument cannot have a null value.

Example

```
CALL SYSPROC.XSR_UPDATE(
  'STORE',
  'PROD',
  'STORE',
  'NEWPROD',
  1)
```

The contents of the XML schema STORE.PROD is updated with the contents of STORE.NEWPROD, and the XML schema STORE.NEWPROD is dropped.

Chapter 6. SQL queries

A *query* specifies a result table. A query is a component of certain SQL statements.

The three forms of a query are:
- subselect
- fullselect
- select-statement.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following authorities:
- For each table or view identified in the query, one of the following authorities:
  - SELECT privilege on the table or view
  - CONTROL privilege on the table or view
- DATAACCESS authority

For each global variable used as an expression in the query, the privileges held by the authorization ID of the statement must include one of the following authorities:
- READ privilege on the global variable that is not defined in a module
- EXECUTE privilege on the module of the global variable that is defined in a module

If the query contains an SQL data change statement, the authorization requirements of that statement also apply to the query.

Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements or DDL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Queries and table expressions

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

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

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

Queries and table expressions

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

subselect

The *subselect* is a component of the fullselect.

```
►►──select-clause──from-clause─────────────────────────────────────────────────►
                              └─where-clause─┘  └─group-by-clause─┘

►──────────────────────────────────────────────────────────────────────────────►
   └─having-clause─┘  └─order-by-clause─┘  └─fetch-first-clause─┘

►──────────────────────────────────────────────────────────────────────────────►◄
   └─isolation-clause─┘
```

A subselect specifies a result table derived from the tables, views or nicknames
identified in the FROM clause. The derivation can be described as a sequence of
operations in which the result of each operation is input for the next. (This is only
a way of describing the subselect. The method used to perform the derivation can
be quite different from this description. If portions of the subselect do not actually
need to be executed for the correct result to be obtained, they might or might not
be executed.)

The authorization for a *subselect* is described in the Authorization section in "SQL
queries".

The clauses of the subselect are processed in the following sequence:
1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
6. ORDER BY clause
7. FETCH FIRST clause

A subselect that contains an ORDER BY or FETCH FIRST clause cannot be
specified:
- In the outermost fullselect of a view.
- In the outer fullselect of a materialized query table.
- Unless the subselect is enclosed in parenthesis.

For example, the following is not valid (SQLSTATE 428FJ):

```
SELECT * FROM T1
   ORDER BY C1
UNION
SELECT * FROM T2
   ORDER BY C1
```

The following example *is* valid:

```
(SELECT * FROM T1
   ORDER BY C1)
UNION
(SELECT * FROM T2
   ORDER BY C1)
```
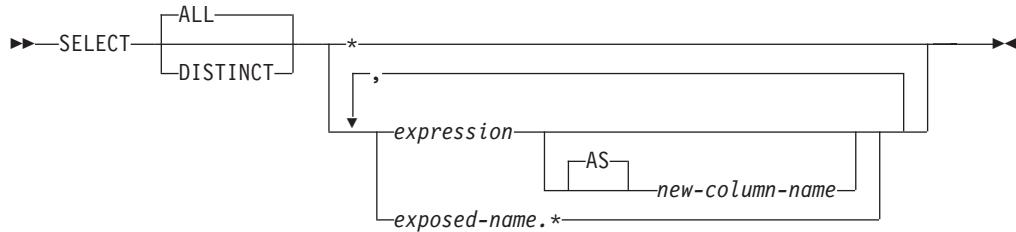
Note: An ORDER BY clause in a subselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

For details about the clauses in the subselect query, refer to the following topics:
- "select-clause" on page 787
- "from-clause" on page 791
- "where-clause" on page 816
- "group-by-clause" on page 817
- "having-clause" on page 831
- "order-by-clause" on page 832
- "fetch-first-clause" on page 835
- "isolation-clause (subselect query)" on page 836

select-clause

The SELECT clause specifies the columns of the final result table.

The column values are produced by the application of the *select list* to the final result table, R. The select list is the names or expressions specified in the SELECT clause, and R is the result of the previous operation of the subselect. For example, if the only clauses specified are SELECT, FROM, and WHERE, R is the result of that WHERE clause.

ALL

Retains all rows of the final result table, and does not eliminate redundant duplicates. This is the default.

DISTINCT

Eliminates all but one of each set of duplicate rows of the final result table. If DISTINCT is used, no string column of the result table can be a LOB type, distinct type based on LOB, or structured type. DISTINCT can be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in an aggregate function of the select list or HAVING clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value in the second. For determining duplicates, two null values are considered equal, and two different decimal floating-point representations of the same number are considered equal. For example, -0 is equal to +0 and 2.0 is equal to 2.00. Each of the decimal floating-point special values are also considered equal: -NAN equals -NAN, -SNAN equals -SNAN, -INFINITY equals -INFINITY, INFINITY equals INFINITY, SNAN equals SNAN, and NAN equals NAN.

When the data type of a column is decimal floating-point, and multiple representations of the same number exist in the column, the particular value that is returned for a SELECT DISTINCT can be any one of the representations in the column. For more information, see "Numeric comparisons" on page 136.

For compatibility with other SQL implementations, UNIQUE can be specified as a synonym for DISTINCT.

Select list notation

*   Represents a list of names that identify the columns of table R, excluding any columns defined as IMPLICITLY HIDDEN. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.

    The list of names is established when the program containing the SELECT clause is bound. Hence * (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

expression
> Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R. The result type of the expression cannot be a row type (SQLSTATE 428H2).

> *new-column-name* **or AS** *new-column-name*
>> Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:
>> - A *new-column-name* specified in the AS clause can be used in the order-by-clause, provided the name is unique.
>> - A *new-column-name* specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
>> - A *new-column-name* specified in the AS clause cannot be used in the update-clause.
>> - A *new-column-name* specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

exposed-name.*
> Represents the list of names that identify the columns of the result table identified by *exposed-name*, excluding any columns defined as IMPLICITLY HIDDEN. The *exposed-name* can be a table name, view name, nickname, or correlation name, and must designate a table, view or nickname named in the FROM clause. The first name in the list identifies the first column of the table, view or nickname, the second name in the list identifies the second column of the table, view or nickname, and so on.

> The list of names is established when the statement containing the SELECT clause is bound. Therefore, * does not identify any columns that have been added to a table after the statement has been bound.

> The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

Limitations on string columns

For restrictions using varying-length character strings on the select list, see "Character strings" on page 93.

Applying the select list

Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. The results are described in two separate lists.

If GROUP BY or HAVING is used
- An expression *X* (not an aggregate function) used in the select list must have a GROUP BY clause with:
  - a *grouping-expression* in which each expression or column-name unambiguously identifies a column of R (see "group-by-clause" on page 817) or
  - each column of R referenced in *X* as a separate *grouping-expression*.

- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the aggregate functions in the select list.

If neither GROUP BY nor HAVING is used

- Either the select list must not include any aggregate functions, or each *column-name* in the select list must be specified within an aggregate function or must be a correlated column reference.
- If the select does not include aggregate functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of aggregate functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns

Result columns do not allow null values if they are derived from:
- A column that does not allow null values
- A constant
- The COUNT or COUNT_BIG function
- A host variable that does not have an indicator variable
- A scalar function or expression that does not include an operand where nulls are allowed
- An expression with a result type that is a weakly typed distinct type defined with a NOT NULL data type constraint

Result columns allow null values if they are derived from:
- Any aggregate function except COUNT or COUNT_BIG
- A column where null values are allowed
- A scalar function or expression that includes an operand where nulls are allowed
- A NULLIF function with arguments containing equal values
- A host variable that has an indicator variable, an SQL parameter, an SQL variable, or a global variable
- A result of a set operation if at least one of the corresponding items in the select list is nullable
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with **dft_sqlmathwarn** set to Yes
- A scalar subselect
- A dereference operation
- A GROUPING SETS *grouping-expression*

Names of result columns

- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and 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.
a constant	the same as the data type of the constant.
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for DECIMAL variables, or the same precision for DECFLOAT variables.
the name of any string column	the same as the data type of the column, with the same length attribute.
the name of any string variable	the same as the data type of the variable, with the same length attribute; if the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
the name of a datetime column	the same as the data type of the column.
the name of a user-defined type column	the same as the data type of the column.
the name of a reference type column	the same as the data type of the column.

from-clause

The FROM clause specifies an intermediate result table.

```
              ┌─ , ──────────────┐
              │                  │
►►──FROM──────▼─table-reference──┴──────────────────────────────────►◄
```
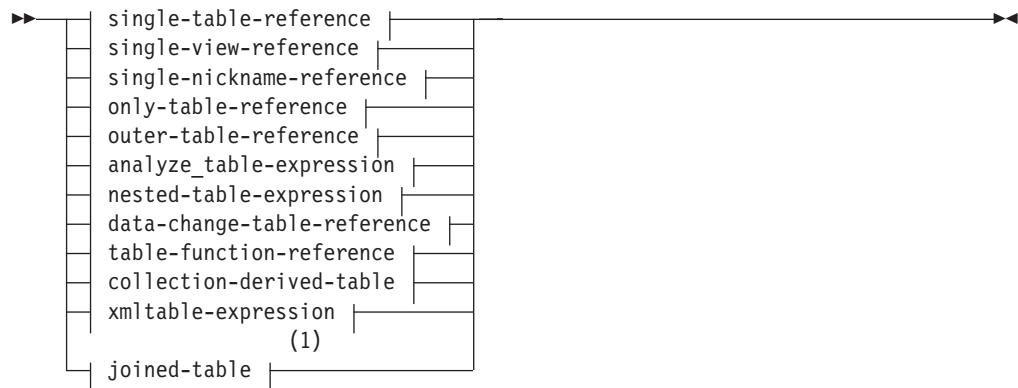
If only one *table-reference* is specified, the intermediate result table is the result of that *table-reference*. If more than one *table-reference* is specified, the intermediate result table consists of all possible combinations of the rows of the specified *table-reference* (the Cartesian product). Each row of the result is a row from the first *table-reference* concatenated with a row from the second *table-reference*, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table references. For a description of *table-reference*, see "table-reference" on page 792.
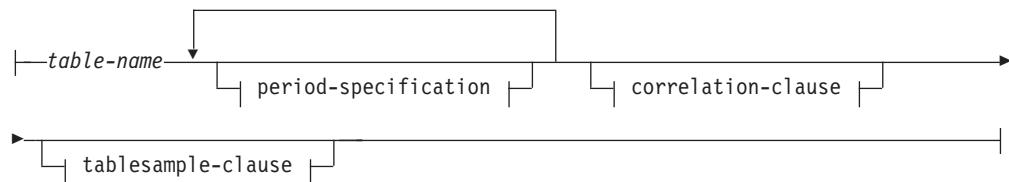
table-reference
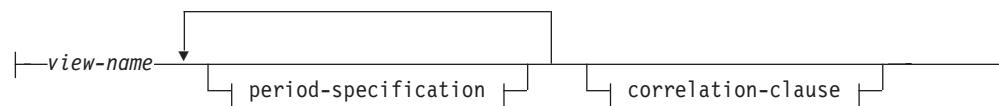
A *table-reference* specifies an intermediate result table.
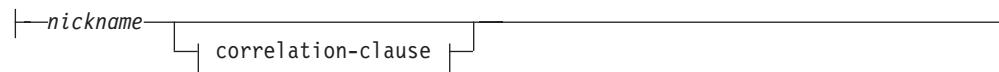
```
>>--+--| single-table-reference |--+------------------------------><
    +--| single-view-reference |---+
    +--| single-nickname-reference |-+
    +--| only-table-reference |---+
    +--| outer-table-reference |--+
    +--| analyze_table-expression |-+
    +--| nested-table-expression |--+
    +--| data-change-table-reference |-+
    +--| table-function-reference |-+
    +--| collection-derived-table |-+
    +--| xmltable-expression |-----+
    |              (1)             |
    +--| joined-table |------------+
```
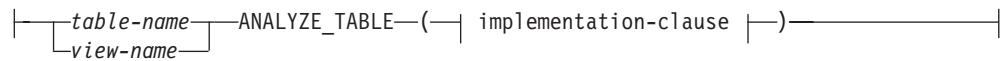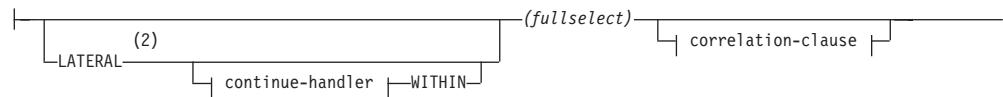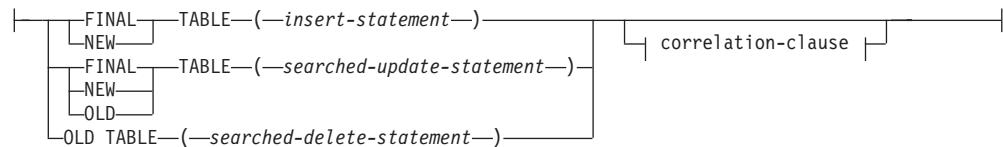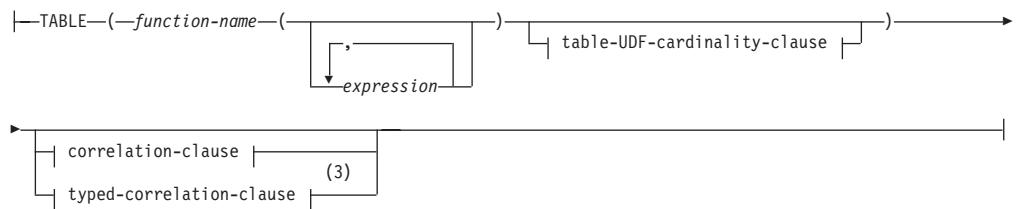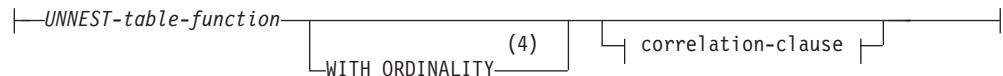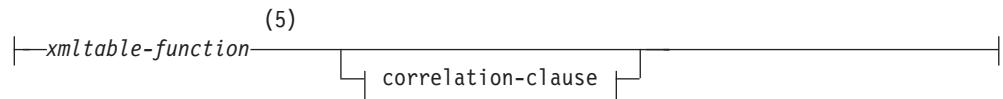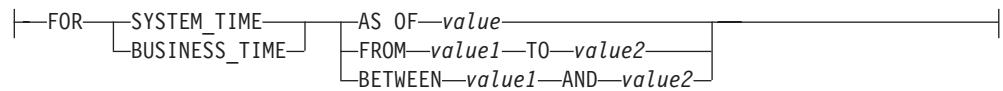
single-table-reference:

```
|--table-name--+------------------------------+--------------->
               +--| period-specification |--+  +--| correlation-clause |--+
               
>--+-----------------------------+--|
   +--| tablesample-clause |--+
```

single-view-reference:

```
|--view-name--+------------------------------+------------------|
              +--| period-specification |--+  +--| correlation-clause |--+
```

single-nickname-reference:

```
|--nickname--+-------------------------+--|
             +--| correlation-clause |-+
```

only-table-reference:

```
|--ONLY--(--+--table-name--+--)----------------------|
            +--view-name---+    +--| correlation-clause |--+
```

outer-table-reference:

```
|--OUTER--(--+--table-name--+--)--------------------|
             +--view-name---+    +--| correlation-clause |--+
```

analyze_table-expression:

```
├──┬─table-name─┬──ANALYZE_TABLE──(──┤ implementation-clause ├──)────────────┤
   └─view-name──┘
```

nested-table-expression:

```
├──┬──────────────────────────────────────┬──(fullselect)──┬────────────────────────┬──┤
   │                         (2)           │                └─┤ correlation-clause ├─┘
   └─LATERAL─┬──────────────────────────┬──┘
             └─┤ continue-handler ├──WITHIN─┘
```

data-change-table-reference:

```
├──┬──┬─FINAL─┬──TABLE──(──insert-statement──)────────────────┬──┬────────────────────────┬──┤
   │  └─NEW───┘                                                │  └─┤ correlation-clause ├─┘
   │  ┌─FINAL─┐                                                │
   ├──┼─NEW───┼──TABLE──(──searched-update-statement──)────────┤
   │  └─OLD───┘                                                │
   └─OLD TABLE──(──searched-delete-statement──)────────────────┘
```

table-function-reference:

```
├──TABLE──(──function-name──(──┬─────────────────┬──)──┬─────────────────────────────┬──)──►
                               │  ┌─,──────────┐  │     └─┤ table-UDF-cardinality-clause ├─┘
                               └──┴─expression─┴──┘

►──┬──────────────────────────────────┬────────────────────────────────────────────────────┤
   ├─┤ correlation-clause ├───────────┤
   │                           (3)    │
   └─┤ typed-correlation-clause ├─────┘
```

collection-derived-table:

```
├──UNNEST-table-function──┬────────────────────────┬──┬────────────────────────┬──┤
                          │                   (4)  │  └─┤ correlation-clause ├─┘
                          └─WITH ORDINALITY────────┘
```

xmltable-expression:

```
                      (5)
├──xmltable-function──────────────────┬────────────────────────┬──────────────┤
                                      └─┤ correlation-clause ├─┘
```

period-specification:

```
├──FOR──┬─SYSTEM_TIME───┬──┬─AS OF──value─────────────────────┬──────────────────┤
        └─BUSINESS_TIME─┘  ├─FROM──value1──TO──value2─────────┤
                           └─BETWEEN──value1──AND──value2──────┘
```

table-reference

correlation-clause:

```
         ┌─AS─┐
├────────┴────┴── correlation-name ──┬──────────────────────────────┬──┤
                                     │        ┌─,────────────┐       │
                                     └──( ──▼── column-name ──┴── ) ─┘
```

tablesample-clause:

```
├── TABLESAMPLE ──┬── BERNOULLI ──┬── ( ── numeric-expression1 ── ) ────────▶
                  └── SYSTEM ──────┘

▶────┬──────────────────────────────────────────────┬──┤
     └── REPEATABLE ── ( ── numeric-expression2 ── ) ─┘
```

implementation-clause:

```
├── IMPLEMENTATION ── ' ── string ── ' ──────────────────────────────┤
```

continue-handler:

```
                            ┌─,───────────────────────────┐
├── RETURN DATA UNTIL ──▼── specific-condition-value ──┴──────┤
```

table-UDF-cardinality-clause:

```
├──┬── CARDINALITY ── integer-constant ──────────────────┬─────────────┤
   └── CARDINALITY MULTIPLIER ── numeric-constant ────────┘
```

typed-correlation-clause:

```
         ┌─AS─┐
├────────┴────┴── correlation-name ──┬──────────────────────────────────────┬──┤
                                     │        ┌─,─────────────────────┐      │
                                     └──( ──▼── column-name ── data-type ──┴── ) ─┘
```

specific-condition-value:

```
                              ┌─VALUE─┐
├── FEDERATED ── SQLSTATE ──┴───────┴── string-constant ──┬──────────────────────────┬──┤
                                                          │            ┌─,──────────┐ │
                                                          └── SQLCODE ──▼── integer-constant ──┴─┘
```

Notes:

1   The syntax for joined-table is covered in a separate topic; refer to "joined-table" on page 811.

2   TABLE can be specified in place of LATERAL.

3　The typed-correlation-clause is required for generic table functions. This clause cannot be specified for any other table functions.

4　WITH ORDINALITY can be specified only if the argument to the UNNEST table function is one or more ordinary array variables or functions with ordinary array return types; an associative array variable or function with an associative array return type cannot be specified (SQLSTATE 428HT).

5　An XMLTABLE function can be part of a table-reference. In this case, subexpressions within the XMLTABLE expression are in-scope of prior range variables in the FROM clause. For more information, see the description of "XMLTABLE".

A *table-reference* specifies an intermediate result table.
- If a single-table-reference is specified without a period-specification or a tablesample-clause, the intermediate result table is the rows of the table. If a period-specification is specified, the intermediate result table consists of the rows of the temporal table where the period matches the specification. If a tablesample-clause is specified, the intermediate result table consists of a sampled subset of the rows of the table.
- If a single-view-reference is specified without a period-specification, the intermediate result table is that view. If a period-specification is specified, temporal table references in the view consider only the rows where the period matches the specification.
- If a single-nickname-reference is specified, the intermediate result table is the data from the data source for that nickname.
- If an only-table-reference is specified, the intermediate result table consists of only the rows of the specified table or view without considering the applicable subtables or subviews.
- If an outer-table-reference is specified, the intermediate result table represents a virtual table based on all the subtables of a typed table or the subviews of a typed view.
- If an analyze_table-expression is specified, the result table contains the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data.
- If a nested-table-expression is specified, the result table is the result of the specified fullselect.
- If a data-change-table-reference is specified, the intermediate result table is the set of rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause.
- If a table-function-reference is specified, the intermediate result table is the set of rows that are returned by the table function.
- If a collection-derived-table is specified, the intermediate result table is the set of rows that are returned by the UNNEST function.
- If an xmltable-expression is specified, the intermediate result table is the set of rows that are returned by the XMLTABLE function.
- If a joined-table is specified, the intermediate result table is the result of one or more join operations. For more information, see "joined-table" on page 811.

single-table-reference

Each *table-name* specified as a table-reference must identify an existing table at the application server or an existing table at a remote server specified using a remote-object-name. The intermediate result table is the result of the table. If the *table-name* references a typed table, the intermediate result table is the

UNION ALL of the table with all its subtables, with only the columns of the *table-name*. A period-specification can be used with a temporal table to specify the period from which the rows are returned as the intermediate result table. A tablesample-clause can be used to specify that a sample of the rows be returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST* and *table-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT* and *table-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
table-name FOR BUSINESS_TIME AS OF CTBT
```

single-view-reference

Each *view-name* specified as a table-reference must identify one of the following objects:

- An existing view at the application server
- A view at a remote server specified using a remote-object-name
- The *table-name* of a common table expression

The intermediate result table is the result of the view or common table expression. If the *view-name* references a typed view, the intermediate result table is the UNION ALL of the view with all its subviews, with only the columns of the *view-name*. A period-specification can be used with a view defined over a temporal table to specify the period from which the rows are returned as the intermediate result table.

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a non-null value *CTST*, and *view-name* identifies a system-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR SYSTEM_TIME AS OF CTST
```

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a non-null value *CTBT*, and *view-name* identifies an application-period temporal table, the table reference is executed as if it contained the following specification with the special register set to the null value:

```
view-name FOR BUSINESS_TIME AS OF CTBT
```

single-nickname-reference

Each *nickname* specified as a table-reference must identify an existing nickname at the application server. The intermediate result table is the result of the nickname.

only-table-reference

The use of ONLY(*table-name*) or ONLY(*view-name*) means that the rows of the applicable subtables or subviews are not included in the intermediate result table. If the *table-name* used with ONLY does not have subtables, then ONLY(*table-name*) is equivalent to specifying *table-name*. If the *view-name* used with ONLY does not have subviews, then ONLY(*view-name*) is equivalent to specifying *view-name*.

The use of ONLY requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

outer-table-reference

The use of OUTER(*table-name*) or OUTER(*view-name*) represents a virtual table. If the *table-name* or *view-name* used with OUTER does not have subtables or subviews, then specifying OUTER is equivalent to not specifying OUTER. If the *table-name* does have subtables, the intermediate result table from OUTER(*table-name*) is derived from *table-name* as follows:

- The columns include the columns of *table-name* followed by the additional columns introduced by each of its subtables, if any. The additional columns are added on the right, traversing the subtable hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *table-name* and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

If the *view-name* does have subviews, the intermediate result table from OUTER(*view-name*) is derived from *view-name* as follows:

- The columns include the columns of *view-name* followed by the additional columns introduced by each of its subviews, if any. The additional columns are added on the right, traversing the subview hierarchy in depth-first order. Subviews that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *view-name* and all the rows of its subviews. Null values are returned for columns that are not in the subview for the row.

The use of OUTER requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

analyze_table-expression

table-name | *view-name*
>   The *table-name* or *view-name* variable must identify an existing table or view or identify the *table-name* of a common table expression that you define preceding the fullselect containing the table-reference. You can specify a nickname. However, in-database analytics are intended for local data, and retrieving the data for a nickname from another data source does not take advantage of the intended performance benefits.

ANALYZE_TABLE
>   Returns the result of executing a specific data mining model by using an in-database analytics provider, a named model implementation, and input data. A query referencing the ANALYZE_TABLE parameter cannot be a static SQL statement or a data definition language (DDL) statement. Input or output values cannot be of type CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BLOB, CLOB, DBCLOB, NCLOB, XML, or DB2SECURITYLABEL.

IMPLEMENTATION '*string***'**
>   Specifies how the expression is to be evaluated. The *string* parameter is a string constant whose maximum length is 1024 bytes. The specified value is used to establish a session with an in-database analytic provider. When you specify SAS as the provider, you must specify values for the following case-insensitive parameters:

>   **PROVIDER**
>   >   Currently, the only supported provider value is SAS.

ROUTINE_SOURCE_TABLE
Specifies a user table containing the DS2 code (and, optionally, any required format or metadata) to implement the algorithm that is specified by the ROUTINE_SOURCE_NAME parameter. DS2 is a procedural language processor for SAS, designed for data modeling, stored procedures, and data extraction, transformation, and load (ETL) processing.

The routine source table has a defined structure (see the examples at the end of the "*analyze_table-expression*" section) and, in a partitioned database environment, must be on the catalog database partition. The table cannot be a global temporary table. The MODELDS2 column for a particular row must not be empty or contain the null value. If the value of the MODELFORMATS or MODELMETADATA column is not null, the value must have a length greater than 0. If you do not specify a table schema name, the value of the CURRENT SCHEMA special register is used.

ROUTINE_SOURCE_NAME
Specifies the name of the algorithm to use.

For example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
  ROUTINE_SOURCE_NAME=SCORING_FUN1;'
```

If the table name, schema name, or algorithm name contains lowercase or mixed-case letters, specify delimited identifiers, as shown in the following example:

```
IMPLEMENTATION
  'PROVIDER=SAS;
  ROUTINE_SOURCE_TABLE="ETLin"."Source_Table";
  ROUTINE_SOURCE_NAME="Scoring_Fun1";'
```

The following examples show you how to use the ANALYZE_TABLE expression.

SAS tooling helps you to define a table to store model implementations for scoring functions. A row in this table stores an algorithm that is written in DS2, with any required SAS format information and metadata. The MODELNAME column serves as the primary key. For a particular value of the ROUTINE_SOURCE_NAME parameter, at most one row is retrieved from the table that the ROUTINE_SOURCE_TABLE parameter specifies. For example:

```
CREATE TABLE ETLIN.SOURCE_TABLE (
  MODELNAME VARCHAR(128) NOT NULL PRIMARY KEY,
  MODELDS2 BLOB(4M) NOT NULL,
  MODELFORMATS BLOB(4M),
  MODELMETADATA BLOB(4M)
);
```

The MODELNAME column contains the name of the algorithm. The MODELDS2 column contains the DS2 source code that implements the algorithm. The MODELFORMATS column contains the aggregated SAS format definition that the algorithm requires. If the algorithm does not require a SAS format, this column contains the null value. The MODELMETADATA column contains any additional metadata that the algorithm requires. If the algorithm does not require any additional metadata, this column contains the null value. If the SAS EP installer creates the table, it might include additional columns.

- Use the data in columns C1 and C2 in table T1 as input data with the scoring model SCORING_FUN1, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
WITH sas_score_in (c1,c2) AS
  (SELECT c1,c2 FROM t1)
    SELECT *
      FROM sas_score_in ANALYZE_TABLE(
        IMPLEMENTATION
          'PROVIDER=SAS;
           ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
           ROUTINE_SOURCE_NAME=SCORING_FUN1;');
```

- Use all the data in the table T2 with the scoring model SCORING_FUN2, whose implementation is stored in ETLIN.SOURCE_TABLE:

```
SELECT *
  FROM t2 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN2;');
```

- Use all the data in view V1 with the scoring model SCORING_FUN3, whose implementation is stored in ETLIN.SOURCE_TABLE, and return the output in ascending order of the first output column:

```
SELECT *
  FROM v1 ANALYZE_TABLE(
    IMPLEMENTATION
      'PROVIDER=SAS;
       ROUTINE_SOURCE_TABLE=ETLIN.SOURCE_TABLE;
       ROUTINE_SOURCE_NAME=SCORING_FUN3;')
    ORDER BY 1;
```

nested-table-expression

A fullselect in parentheses is called a *nested table expression*. The intermediate result table is the result of that fullselect. The columns of the result do not need unique names, but a column with a non-unique name cannot be explicitly referenced. If LATERAL is specified, the fullselect can include correlated references to results columns of table references specified to the left of the nested table expression. If the nested table expression involves data from a federated data source, a continue-handler can be specified to tolerate certain error conditions from the data source.

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is valid only when it does not include:

- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a FROM clause, the view must meet either of the following conditions:

- Be symmetric (have WITH CHECK OPTION specified)
- Satisfy the restriction for a WITH CHECK OPTION view

If the target of a data change statement within a FROM clause is a nested table expression, the following restrictions apply:

- Modified rows are not requalified

- WHERE clause predicates are not reevaluated
- ORDER BY or FETCH FIRST operations are not redone

A nested table expression can be used in the following situations:

- In place of a view to avoid creating the view (when general use of the view is not required)
- When the required intermediate result table is based on host variables

data-change-table-reference

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A *data-change-table-reference* can be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a *select-statement*, a SELECT INTO statement, or a common table expression. A *data-change-table-reference* can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change statement is considered to be a table or view that is referenced in the query; therefore, the authorization ID of the query must have SELECT privilege on that target table or view. A *data-change-table-reference* clause cannot be specified in a view definition, materialized query table definition, or FOR statement (SQLSTATE 428FL).

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807) or a nickname (SQLSTATE 25000).

Expressions in the select list of a view or fullselect as target of a data change statement in a *table-reference* can be selected only if OLD TABLE is specified or the expression does not include the following elements (SQLSTATE 428G6):

- A subquery
- A function that reads or modifies SQL data
- A function is that is non-deterministic or has an external action
- An OLAP function
- A NEXT VALUE FOR *sequence* reference

FINAL TABLE
    Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they exist at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

NEW TABLE
    Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement before the application of referential constraints and AFTER triggers. Data in the target table at the completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

OLD TABLE
    Specifies that the rows of the intermediate result table represent the set of

rows that are changed by the SQL data change statement as they existed before the application of the data change statement.

(searched-update-statement)
Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

(searched-delete-statement)
Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

(insert-statement)
Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a *data-change-table-reference* is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

table-function-reference

In general, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. Each *function-name* together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server. There are, however, some special considerations which apply.

- **Table function column names**: Unless alternative column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS or RETURNS GENERIC TABLE clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are defined in the CREATE TABLE statement.

- **Table function resolution**: The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.

- **Table function arguments**: As with scalar function arguments, table function arguments can generally be any valid SQL expression. The following examples are valid syntax:

```
Example 1:  SELECT c1
              FROM TABLE( tf1('Zachary') ) AS z
              WHERE c2 = 'FLORIDA';

Example 2:  SELECT c1
              FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;

Example 3:  SELECT c1
              FROM t
              WHERE c2 IN
                    (SELECT c3 FROM
                     TABLE( tf5(t.c4) ) AS z  -- correlated reference
                    )                         -- to previous FROM clause
```

```
Example 4:  SELECT c1
            FROM TABLE( tf6('abcd') )              -- tf6 is a generic
              AS z (c1 int, c2 varchar(100)) -- java table function
```

- **Table functions that modify SQL data**: Table functions that are specified with the MODIFIES SQL DATA option can be used only as the last table reference in a *select-statement*, *common-table-expression*, or RETURN statement that is a subselect, a SELECT INTO, or a *row-fullselect* in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

  ```
  Example 1:  SELECT c1
              FROM TABLE( tfmod('Jones') ) AS z

  Example 2:  SELECT c1
              FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) ) AS z

  Example 3:  SET var =
              (SELECT c1
              FROM TABLE( tfmod('Jones') ) AS z

  Example 4:  RETURN SELECT c1
              FROM TABLE( tfmod('Jones') ) AS z

  Example 5:  WITH v1(c1) AS
              (SELECT c1
              FROM TABLE( tfmod(:hostvar1) ) AS z)
              SELECT c1
              FROM v1, t1 WHERE v1.c1 = t1.c1

  Example 6: SELECT z.*
              FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) )
              AS z (col1 int)
  ```

collection-derived-table

A *collection-derived-table* can be used to convert the elements of an array into values of a column in separate rows. If WITH ORDINALITY is specified, an extra column of data type INTEGER is appended. This column contains the position of the element in the array. The columns can be referenced in the select list and the in rest of the subselect by using the names specified for the columns in the correlation-clause. The *collection-derived-table* clause can be used only in a context where arrays are supported (SQLSTATE 42887). See the "UNNEST table function" for details.

xmltable-expression

An *xmltable-expression* specifies an invocation of the built-in XMLTABLE function which determines the intermediate result table. See XMLTABLE for more information.

joined-table

A *joined-table* specifies an intermediate result set that is the result of one or more join operations. For more information, see "joined-table" on page 811.

period-specification

A *period-specification* identifies an intermediate result table consisting of the rows of the referenced table where the period matches the specification. A *period-specification* can be specified following the name of a temporal table or the name of a view. The same period name must not be specified more than once for the same table reference (SQLSTATE 428HY). The rows of the table reference are derived by application of the period specifications.

If the table is a system-period temporal table and a *period-specification* for the SYSTEM_TIME period is not specified, the table reference includes all current rows and does not include any historical rows of the table. If the table is an application-period temporal table and a *period-specification* for the BUSINESS_TIME period is not specified, the table reference includes all rows of the table. If the table is a bitemporal table and a *period-specification* is not specified for both SYSTEM_TIME and BUSINESS_TIME, the table reference includes all current rows of the table and does not include any historical rows of the table.

If the table reference is a single-view-reference, the rows of the view reference are derived by application of the period specifications to all of the temporal tables accessed when computing the result table of the view. If the view does not access any temporal table, then the *period-specification* has no effect on the result table of the view. If *period-specification* is used, the view definition or any view definitions referenced when computing the result table of the view must not include any references to compiled SQL functions or external functions with a data access indication other than NO SQL (SQLSTATE 428HY).

If the CURRENT TEMPORAL SYSTEM_TIME special register is set to a value other than the null value, then a *period-specification* that references SYSTEM_TIME must not be specified for the table reference or view reference, unless the value in effect for the SYSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

If the CURRENT TEMPORAL BUSINESS_TIME special register is set to a value other than the null value, then a period specification that references BUSINESS_TIME must not be specified for the table reference or view reference, unless the value in effect for the BUSTIMESENSITIVE bind option is NO (SQLSTATE 428HY).

FOR SYSTEM_TIME

Specifies that the SYSTEM_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, the table must be a system-period temporal table (SQLSTATE 428HY). FOR SYSTEM_TIME must not be specified if the value of the CURRENT TEMPORAL SYSTEM_TIME special register is not the null value and the SYSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

FOR BUSINESS_TIME

Specifies that the BUSINESS_TIME period is used for the *period-specification*. If the clause is specified following a *table-name*, BUSINESS_TIME must be a period defined in the table (SQLSTATE 4274M). FOR BUSINESS_TIME must not be specified if the value of the CURRENT TEMPORAL BUSINESS_TIME special register is not the null value and the BUSTIMESENSITIVE bind option is set to YES (SQLSTATE 428HY).

value, *value1*, **and** *value2*

The *value*, *value1*, and *value2* expressions return the null value or a value of one of the following built-in data types (SQLSTATE 428HY): a DATE, a TIMESTAMP, or a character string that is not a CLOB or DBCLOB. If the argument is a character string, it must be a valid character string representation of a timestamp or a date (SQLSTATE 22007). For the valid formats of string representations of timestamp values, see the section "String representations of datetime values" in the topic "Datetime values".

Each expression can contain any of the following supported operands (SQLSTATE 428HY):

- Constant
- Special register
- Variable (host-variable, SQL parameter, SQL variable, transition variable)
- Parameter marker
- Scalar function whose arguments are supported operands (user-defined functions and non-deterministic functions cannot be used)
- CAST specification where the cast operand is a supported operand
- Expression using arithmetic operators and operands

AS OF *value*

Specifies that the table reference includes each row for which the value of the begin column for the specified period is less than or equal to *value*, and the value of the end column for the period is greater than *value*. If *value* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy number 100 on August 31, 2010.

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  AS OF '2010-08-31' WHERE policy_id = '100'
```

FROM *value1* **TO** *value2*

Specifies that the table reference includes rows that exist for the period specified from *value1* to *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than *value2*, and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than or equal to *value2*. If *value1* or *value2* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy 100, during the year 2009 (from January 1, 2009 at 12:00 AM until before January 1, 2010).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  FROM '2009-01-01' TO '2010-01-01' WHERE policy_id = '100'
```

BETWEEN *value1* **AND** *value2*

Specifies that the table reference includes rows in which the specified period overlaps at any point in time between *value1* and *value2*. A row is included in the table reference if the value of the begin column for the specified period in the row is less than or equal to *value2* and the value of the end column for the specified period in the row is greater than *value1*. The table reference contains zero rows if *value1* is greater than *value2*. If *value1* is equal to *value2*, the expression is equivalent to AS OF *value1*. If *value1* or *value2* is the null value, the table reference is an empty table.

Example: The following query returns the insurance coverage information for insurance policy number 100, during the year 2008 (between January 1, 2008 and December 31, 2008 inclusive).

```
SELECT coverage FROM policy_info FOR BUSINESS_TIME
  BETWEEN '2008-01-01' AND '2008-12-31' WHERE policy_id = '100'
```

Following are syntax alternatives for *period-specification* clauses:

- AS OF TIMESTAMP can be specified in place of FOR SYSTEM_TIME AS OF

- VERSIONS BETWEEN TIMESTAMP can be specified in place of FOR SYSTEM_TIME BETWEEN

correlation-clause

The exposed names of all table references must be unique. An exposed name is:

- A *correlation-name*
- A *table-name* that is not followed by a *correlation-name*
- A *view-name* that is not followed by a *correlation-name*
- A *nickname* that is not followed by a *correlation-name*
- An *alias-name* that is not followed by a *correlation-name*

If a *correlation-clause* clause does not follow a *function-name* reference, *xmltable-expression* expression, nested table expression, or *data-change-table-reference* reference, or if a *typed-correlation-clause* clause does not follow a *function-name* reference, then there is no exposed name for that table reference.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference, *xmltable-expression*, nested table expression, or *data-change-table-reference*. Any qualified reference to a column must use the exposed name. If the same table name, view, or nickname is specified twice, at least one specification must be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table, view or nickname. When a *correlation-name* is specified, *column-name*s can also be specified to give names to the columns of the table reference. If the *correlation-clause* does not include *column-name*s, the exposed column names are determined as follows:

- Column names of the referenced table, view, or nickname when the *table-reference* is a *table-name*, *view-name*, *nickname*, or *alias-name*
- Column names specified in the RETURNS clause of the CREATE FUNCTION statement when the *table-reference* is a *function-name* reference
- Column names specified in the COLUMNS clause of the *xmltable-expression* when the *table-reference* is an *xmltable-expression*
- Column names exposed by the fullselect when the *table-reference* is a *nested-table-expression*
- Column names from the target table of the data change statement, along with any defined INCLUDE columns when the *table-reference* is a *data-change-table-reference*

tablesample-clause

The optional *tablesample-clause* can be used to obtain a random subset (a sample) of the rows from the specified *table-name,* rather than the entire contents of that *table-name,* for this query. This sampling is in addition to any predicates that are specified in the *where-clause*. Unless the optional REPEATABLE clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any sample must return the same rows. The size of the sample is controlled by the *numeric-expression1* in parentheses, representing an approximate percentage (P) of the table to be returned.

TABLESAMPLE

The method by which the sample is obtained is specified after the TABLESAMPLE keyword, and can be either BERNOULLI or SYSTEM. For both methods, the exact number of rows in the sample might be different

for each execution of the query, but on average is approximately P percent of the table, before any predicates further reduce the number of rows.

The *table-name* must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table can be sampled in a query.

BERNOULLI

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability P/100 (where P is the value of *numeric-expression1*), and excludes each row with probability 1 - P/100, independently of the other rows. So if the *numeric-expression1* evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

SYSTEM

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a *table-name* means that each page of *table-name* is included in the sample with probability P/100, and excluded with probability 1 - P/100. All rows on each page that is included qualify for the sample. SYSTEM sampling of a *table-name* generally executes much faster than BERNOULLI sampling, because fewer data pages are retrieved. However, SYSTEM sampling can often yield less accurate estimates for aggregate functions, such as SUM(SALES), especially if the rows of *table-name* are clustered on any columns referenced in that query. The optimizer might in certain circumstances decide that it is more efficient to perform SYSTEM sampling as if it were BERNOULLI sampling. An example is when a predicate on *table-name* can be applied by an index and is much more selective than the sampling rate P.

numeric-expression1

The *numeric-expression1* specifies the size of the sample to be obtained from *table-name*, expressed as a percentage. It must be a constant numeric expression that cannot contain columns. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 is sampled, on average. A *numeric-expression1* that evaluates to 100 is handled as if the tablesample-clause were not specified. If *numeric-expression1* evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

REPEATABLE (*numeric-expression2***)**

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query debugging. This can be accomplished by specifying the REPEATABLE clause. The REPEATABLE clause requires the specification of a *numeric-expression2* in parentheses, which serves the same role as the seed in a random number generator. Adding the REPEATABLE clause to the

tablesample-clause of any *table-name* ensures that repeated executions of that query (using the same value for *numeric-expression2*) return the same sample, assuming that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of *table-name* is used across multiple queries, use of a global temporary table is recommended. Alternatively, the multiple queries can be combined into one query, with multiple references to a sample that is defined using the WITH clause.

- *Example 1:* Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
   TABLESAMPLE BERNOULLI(10)
```

- *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-UDF-cardinality-clause

The *table-UDF-cardinality* clause can be specified for each user-defined table function reference within the FROM clause. This option indicates the expected number of rows to be returned only for the SELECT statement that contains it. The CARDINALITY and CARDINALITY MULTIPLIER clauses are not allowed if the table function is an inlined SQL table function (SQLSTATE 42887).

CARDINALITY *integer-constant*
Specifies an estimate of the expected number of rows that are returned by the reference to the user-defined function. The value range of *integer-constant* is from 0 to 9 223 372 036 854 775 807 inclusive.

CARDINALITY MULTIPLIER *numeric-constant*

The product of the specified CARDINALITY MULTIPLIER *numeric-constant* and the reference cardinality value are used by the database server as the expected number of rows that are returned by the table function reference.

In this case, *numeric-constant* can be in the integer, decimal, or floating-point format. The value must be greater than or equal to zero. If the decimal number notation is used, the number of digits can be up to 31. An integer value is treated as a decimal number with no fraction. If zero is specified or the computed cardinality is less than 1, the cardinality of the reference to the user-defined table function is assumed to be 1.

The value in the CARDINALITY column of SYSSTAT.ROUTINES for the table function name is used as the reference cardinality value. If no value is set in the CARDINALITY column of SYSSTAT.ROUTINES, a finite value is assumed as its default value for the reference cardinality value. This finite value is the same value that is assumed for tables for which the RUNSTATS utility has not gathered statistics.

Only a numeric constant can follow the keyword CARDINALITY or
CARDINALITY MULTIPLIER. A host variable or parameter marker is not
supported. Specifying a cardinality value in a table function reference does not
change the CARDINALITY column value for the function in the
SYSSTAT.ROUTINES catalog view.

The CARDINALITY value for external and compiled SQL user-defined table
functions can be changed by updating the CARDINALITY column in the
SYSSTAT.ROUTINES catalog view. The CARDINALITY value for an external
table function can also be initialized by specifying the CARDINALITY option
in the CREATE FUNCTION (external table) statement when a user-defined
table function is created.

typed-correlation-clause

A *typed-correlation-clause* clause defines the appearance and contents of the table
generated by a generic table function. This clause must be specified when the
table-function-references is a generic table function and cannot be specified for
any other table reference. The following *data-type* values are supported in
generic table functions:

Table 84. Data types supported in generic table functions

SQL column data type	Equivalent Java data type
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
DOUBLE	double
DECIMAL(p,s)	java.math.BigDecimal
NUMERIC(p,s)	java.math.BigDecimal
CHAR(n)	java.lang.String
CHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
VARCHAR(n)	java.lang.String
VARCHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
GRAPHIC(n)	java.lang.String
VARGRAPHIC(n)	String
BLOB(n)	COM.ibm.db2.app.Blob
CLOB(n)	COM.ibm.db2.app.Clob
DBCLOB(n)	COM.ibm.db2.app.Clob
DATE	String
TIME	String
TIMESTAMP	String
XML AS CLOB(n)	COM.ibm.db2.jcc.DB2Xml

continue-handler

Certain errors that occur within a *nested-table-expression* can be tolerated, and
instead of returning an error, the query can continue and return a result. This
is referred to as an *error tolerant nested-table-expression*.

Specifying the RETURN DATA UNTIL clause will cause any rows that are returned from the fullselect before the indicated condition is encountered to make up the result set from the fullselect. This means that a partial result set (which can also be an empty result set) from the fullselect is acceptable as the result for the *nested-table-expression*.

The FEDERATED keyword restricts the condition to handle only errors that occur at a remote data source.

The condition can be specified as an SQLSTATE value, with a *string-constant* length of 5. You can optionally specify an SQLCODE value for each specified SQLSTATE value. For portable applications, specify SQLSTATE values as much as possible, because SQLCODE values are generally not portable across platforms and are not part of the SQL standard.

Only certain conditions can be tolerated. Errors that do not allow the rest of the query to be executed cannot be tolerated, and an error is returned for the whole query. The *specific-condition-value* might specify conditions that cannot actually be tolerated by the database manager, even if a specific SQLSTATE or SQLCODE value is specified, and for these cases, an error is returned.

A query or view containing an error tolerant *nested-table-expression* is read-only.

The fullselect of an error tolerant *nested-table-expression* is not optimized using materialized query tables.

specific-condition-value

> The following SQLSTATE values and SQLCODE values have the potential, when specified, to be tolerated by the database manager:
> * SQLSTATE 08001; SQLCODEs -1336, -30080, -30081, -30082
> * SQLSTATE 08004
> * SQLSTATE 42501
> * SQLSTATE 42704; SQLCODE -204
> * SQLSTATE 42720
> * SQLSTATE 28000

Correlated references in table-references

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both of these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the LATERAL keyword must exist before the fullselect. The following examples have valid syntax:

```
Example 1:  SELECT t.c1, z.c5
            FROM t, TABLE( tf3(t.c2) ) AS z      -- t precedes tf3
            WHERE t.c3 = z.c4;                    -- in FROM, so t.c2
                                                  -- is known

Example 2:  SELECT t.c1, z.c5
            FROM t, TABLE( tf4(2 * t.c2) ) AS z  -- t precedes tf4
            WHERE t.c3 = z.c4;                    -- in FROM, so t.c2
                                                  -- is known

Example 3:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                LATERAL (SELECT AVG(e.salary) AS avgsal,
```

```
                            COUNT(*) AS empcount
            FROM employee e               -- department precedes nested
            WHERE e.workdept=d.deptno -- table expression and
          ) AS empinfo;                   -- LATERAL is specified,
                                          -- so d.deptno is known
```

But the following examples are not valid:
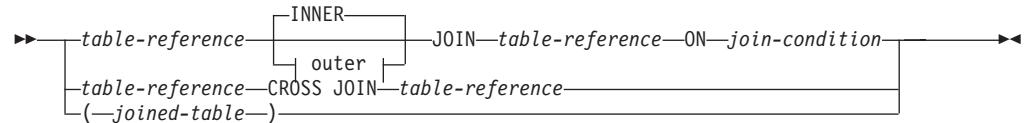
```
Example 4:  SELECT t.c1, z.c5
            FROM TABLE( tf6(t.c2) ) AS z, t  -- cannot resolve t in t.c2!
            WHERE t.c3 = z.c4;            -- compare to Example 1 above.

Example 5:  SELECT a.c1, b.c5
            FROM TABLE( tf7a(b.c2) ) AS a, TABLE( tf7b(a.c6) ) AS b
            WHERE a.c3 = b.c4;            -- cannot resolve b in b.c2!

Example 6:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
               (SELECT AVG(e.salary) AS avgsal,
                       COUNT(*) AS empcount
                FROM employee e               -- department precedes nested
                WHERE e.workdept=d.deptno -- table expression but
              ) AS empinfo;                   -- LATERAL is not specified,
                                              -- so d.deptno is unknown
```

joined-table

A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: CROSS, INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

```
>>─┬─table-reference──┬─INNER──────────┬─JOIN─table-reference─ON─join-condition─┬─><
   │                  └─outer──┘        │                                        │
   ├─table-reference──CROSS JOIN─table-reference────────────────────────────────┤
   └─(─joined-table─)───────────────────────────────────────────────────────────┘
```

outer:

```
├─┬─LEFT──┬─┬─OUTER─┬──────────────────────────────────┤
  ├─RIGHT─┤ └───────┘
  └─FULL──┘
```

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 include any dereference operations or the DEREF function, where the reference value is other than the object identifier column

- Any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- Any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action
- It cannot include an XMLQUERY or XMLEXISTS expression

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 null values are allowed in the columns.

The following list summarizes the result of the join operations:
- The result of T1 CROSS JOIN T2 consists of all possible pairings of their rows.
- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T2.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T2, the concatenation of that row with the null row of T1. Null values are allowed in all columns derived from T1.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. Null values are allowed in all columns derived from T1 and T2.

Examples of subselect queries with joins:

The following examples illustrate the use of joins in a subselect query.

- *Example 1:* This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

  ```
  SELECT * FROM J1
  ```

  ```
  W   X
  --- ------
  A      11
  B      12
  C      13
  ```

  ```
  SELECT * FROM J2
  ```

  ```
  Y   Z
  --- ------
  A      21
  C      22
  D      23
  ```

  The following query does an inner join of J1 and J2 matching the first column of both tables.

  ```
  SELECT * FROM J1 INNER JOIN J2 ON W=Y
  ```

  ```
  W   X      Y   Z
  --- ------ --- ------
  A      11 A      21
  C      13 C      22
  ```

  In this inner join example the row with column W='C' from J1 and the row with column Y='D' from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

  ```
  SELECT * FROM J1, J2 WHERE W=Y
  ```

  The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

  ```
  SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y
  ```

  ```
  W   X      Y   Z
  --- ------ --- ------
  A      11 A      21
  B      12 -      -
  C      13 C      22
  ```

  The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

  ```
  SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y
  ```

  ```
  W   X      Y   Z
  --- ------ --- ------
  A      11 A      21
  C      13 C      22
  -      -  D      23
  ```

  The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

  ```
  SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
  ```

  ```
  W   X      Y   Z
  --- ------ --- ------
  ```

Examples of subselect queries with joins

```
A       11 A       21
C       13 C       22
-        - D       23
B       12 -        -
```

- *Example 2:*   Using the tables J1 and J2 from the previous example, examine what happens when and additional predicate is added to the search condition.

   SELECT * FROM J1 **INNER JOIN** J2 **ON** W=Y **AND** X=13

```
W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

   The additional condition caused the inner join to select only 1 row compared to the inner join in Example 1.

   Notice what the affect of this is on the full outer join.

   SELECT * FROM J1 **FULL OUTER JOIN** J2 **ON** W=Y **AND** X=13

```
W   X      Y   Z
--- ------ --- ------
-        - A       21
C       13 C       22
-        - D       23
A       11 -        -
B       12 -        -
```

   The result now has 5 rows (compared to 4 without the additional predicate) because there was only 1 row in the inner join and all rows of both tables must be returned.

   The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

   SELECT * FROM J1 **FULL OUTER JOIN** J2 **ON** W=Y
      WHERE X=13

```
W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

   The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result is the same as the result of the full outer join query in Example 1. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate when performing outer joins can have a significant affect on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

   SELECT * FROM J1 **INNER JOIN** J2 **ON** W=Y **AND** X=12

   Hence, the full outer join returns 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.

   SELECT * FROM J1 **FULL OUTER JOIN** J2 **ON** W=Y **AND** X=12

```
W   X      Y   Z
--- ------ --- ------
-        - A       21
-        - C       22
-        - D       23
A       11 -        -
B       12 -        -
C       13 -        -
```

   If the additional predicate is in the WHERE clause instead, 1 row is returned.

```
      SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
         WHERE X=12

      W   X      Y   Z
      --- ------ --- ------
      B        12 -      -
```

- *Example 3:* List every department with the employee number and last name of the manager, including departments without a manager.

  ```
    SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
      FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
          ON MGRNO = EMPNO
  ```

- *Example 4:* List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

  ```
    SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
      FROM EMPLOYEE E LEFT OUTER JOIN
                                    DEPARTMENT INNER JOIN EMPLOYEE M
          ON MGRNO = M.EMPNO
          ON E.WORKDEPT = DEPTNO
  ```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

where-clause

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.

►►──WHERE──*search-condition*──────────────────────────────────────►◄
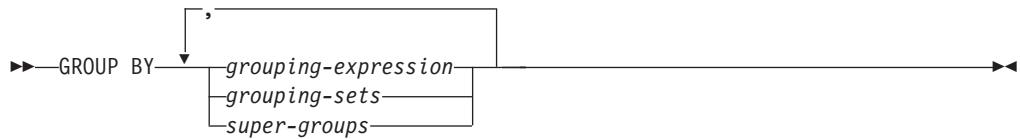
The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.
- An aggregate function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R, and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references might be executed just once, whereas a subquery with a correlated reference might be executed once for each row.

group-by-clause

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.

In its simplest form, a GROUP BY clause contains a *grouping expression*. A grouping expression is an *expression* used in defining the grouping of R. Each expression or *column name* included in grouping-expression must unambiguously identify a column of R (SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar fullselect or an XMLQUERY or XMLEXISTS expression (SQLSTATE 42822), or any expression or function that is not deterministic or has an external action (SQLSTATE 42845).

Note: The following expressions, which do not contain an explicit column reference, can be used in a *grouping-expression* to identify a column of R:
- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of *grouping-sets*, see "grouping-sets" on page 818. For a description of *super-groups*, see "super-groups" on page 819.

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.
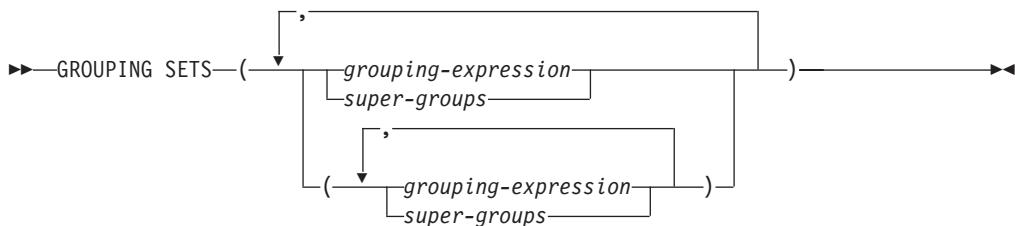
A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a *sort-key-expression* of an ORDER BY clause (see "order-by-clause" on page 832 for details). In each case, the reference specifies only one value for each group. For example, if the *grouping-expression* is *col1+col2*, then an allowed expression in the select list is *col1+col2+3*. Associativity rules for expressions disallow the similar expression, *3+col1+col2*, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, *3+(col1+col2)* is also allowed in the select list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the select list.

If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and might not all have the same length. In that case, a reference to the *grouping-expression* still specifies only one value for each group, 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, not deterministic or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the expression as a column of the result. For an example using nested table expressions, see Example 9 in "Examples of subselect queries" on page 838.

grouping-sets

```
►►──GROUPING SETS──(──┬──grouping-expression──┬──────────)──────────►◄
                      │  └─super-groups────────┘          │
                      │                    ,              │
                      └─(──┬──grouping-expression──┬──)────┘
                           └─super-groups──────────┘
```

A *grouping-sets* specification can be used to specify multiple grouping clauses in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. The groups can be computed with a single pass over the base table using *grouping-sets*.

A simple *grouping-expression* or the more complex forms of *super-groups* are supported by the *grouping-sets* specification. For a description of *super-groups*, see "super-groups" on page 819.

Note that grouping sets are the fundamental building blocks for GROUP BY operations. A simple GROUP BY with a single column can be considered a grouping set with one element. For example:

    GROUP BY a

is the same as

    GROUP BY GROUPING SETS((a)**)**

and
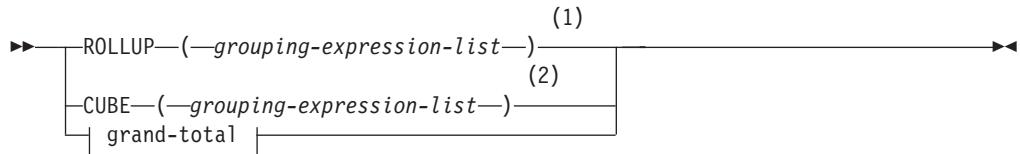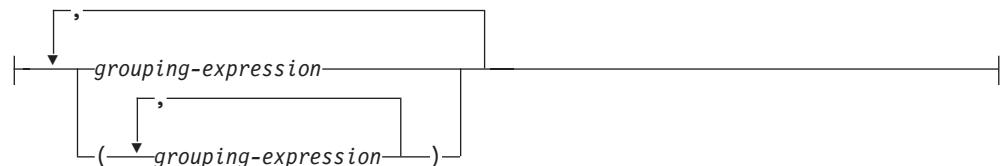
    GROUP BY a,b,c

is the same as

    GROUP BY GROUPING SETS((a,b,c)**)**

Non-aggregation columns from the select list of the subselect that are excluded from a grouping set will return a null for such columns for each row generated for that grouping set. This reflects the fact that aggregation was done without considering the values for those columns.

The use of grouping sets is illustrated in Example 2 through Example 7 in "Examples of grouping sets, cube, and rollup queries" on page 824.

super-groups

grouping-expression-list:

grand-total:

Notes:

1   Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.

2   Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

ROLLUP (*grouping-expression-list* **)**

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the "regular" grouped rows. *Sub-total* rows are "super-aggregate" rows that contain further aggregates whose values are derived by applying the same aggregate functions that were used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any aggregate function can be used for the aggregation. For instance, MAX and AVG are used in Example 8 in "Examples of grouping sets, cube, and rollup queries" on page 824. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

A ROLLUP grouping is a series of *grouping-sets*. The general specification of a ROLLUP with n elements

```
GROUP BY ROLLUP(C₁,C₂,...,Cₙ₋₁,Cₙ)
```

is equivalent to

```
GROUP BY GROUPING SETS((C₁,C₂,...,Cₙ₋₁,Cₙ)
                       (C₁,C₂,...,Cₙ₋₁)
                       ...
                       (C₁,C₂)
                       (C₁)
                       () )
```

Note that the n elements of the ROLLUP translate to $n+1$ grouping sets. Note also that the order in which the *grouping-expressions* is specified is significant for ROLLUP. For example, the following clause:

```
GROUP BY ROLLUP(a,b)
```

is equivalent to:
```
GROUP BY GROUPING SETS((a,b)
                       (a)
                       () )
```

Similarly, the following clause:
```
GROUP BY ROLLUP(b,a)
```

is equivalent to:
```
GROUP BY GROUPING SETS((b,a)
                       (b)
                       () )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. Example 3 in "Examples of grouping sets, cube, and rollup queries" on page 824 illustrates the use of ROLLUP.

CUBE (*grouping-expression-list*)

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains "cross-tabulation" rows. *Cross-tabulation* rows are additional "super-aggregate" rows that are not part of an aggregation with sub-totals. The GROUPING aggregate function can be used to indicate if a row was generated by the super-group.

Similar to a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2**n (2 to the power *n*) *grouping-sets*. For example, a specification of:
```
GROUP BY CUBE(a,b,c)
```

is equivalent to:
```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (b,c)
                       (a)
                       (b)
                       (c)
                       () )
```

Note that the three elements of the CUBE translate into eight grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales_Person)' and 'CUBE (Sales_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. The use of CUBE is illustrated in Example 4 in "Examples of grouping sets, cube, and rollup queries" on page 824.

grouping-expression-list

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expression*s.

For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However, the clause:
```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted subtotal rows for the County. In the clause:

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the required result. In other words, the two-element ROLLUP:

```
GROUP BY ROLLUP(Province, (County, City))
```

generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province)
                       () )
```

and the three-element ROLLUP generates:

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province, County)
                       (Province)
                       () )
```

Example 2 in "Examples of grouping sets, cube, and rollup queries" on page 824 also utilizes composite column values.

grand-total

Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This can be separately specified with empty parentheses within the GROUPING SET clause. It can also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. Example 4 in "Examples of grouping sets, cube, and rollup queries" on page 824 uses the grand-total syntax.

Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expression* fields are combined with other groups, they are "appended" to the beginning of the resulting *grouping sets*. When ROLLUP or CUBE expressions are combined, they operate similarly to "multipliers" on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows:

```
GROUP BY a, ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a) )
```

Or similarly,

```
GROUP BY a, b, ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b) )
```

Combining of *ROLLUP* elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                        (a,b)
                        (a)
                        (b,c)
                        (b)
                        () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                        (a,b)
                        (a,c)
                        (a)
                        (b,c)
                        (b)
                        (c)
                        () )
```

Combining of *CUBE* and *ROLLUP* elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                        (a,b,c)
                        (a,b)
                        (a,c,d)
                        (a,c)
                        (a)
                        (b,c,d)
                        (b,c)
                        (b)
                        (c,d)
                        (c)
                        () )
```

Similar to a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                        (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that might be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```
GROUP BY Region,
         ROLLUP(Sales_Person, WEEK(Sales_Date)),
         CUBE(YEAR(Sales_Date), MONTH (Sales_Date))
```

The column listed immediately to the right of GROUP BY is grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the GROUP BY clause results in a cube of

MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person or WEEK(Sales_Date) so produces fewer rows than the clause:

```
GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                 YEAR(Sales_Date), MONTH(Sales_Date) )
```

Examples of grouping sets, cube, and rollup queries

The following examples illustrate the grouping, cube, and rollup forms of subselect queries.

The queries in Example 1 through Example 4 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 LUCCHESSI                 3
         13           6 LUCCHESSI                 1
         13           6 LEE                       2
         13           6 LEE                       2
         13           6 LEE                       3
         13           6 LEE                       5
         13           6 GOUNOT                    3
         13           6 GOUNOT                    1
         13           6 GOUNOT                    7
         13           7 LUCCHESSI                 1
         13           7 LUCCHESSI                 2
         13           7 LUCCHESSI                 1
         13           7 LEE                       7
         13           7 LEE                       3
         13           7 LEE                       7
         13           7 LEE                       4
         13           7 GOUNOT                    2
         13           7 GOUNOT                   18
         13           7 GOUNOT                    1
```

- *Example 1:* Here is a query with a basic GROUP BY clause over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
```

- *Example 2:* Produce the result based on two different grouping sets of rows from the SALES table.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS ( (WEEK(SALES_DATE), SALES_PERSON),
                         (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
```

The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

- *Example 3:* If you use the 3 distinct columns involved in the grouping sets of Example 2 and perform a ROLLUP, you can see grouping sets for (WEEK,DAY_WEEK,SALES_PERSON), (WEEK, DAY_WEEK), (WEEK) and grand total.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
  FROM SALES
  WHERE WEEK(SALES_DATE) = 13
  GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
  ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - -                        73
          -           - -                        73
```

- *Example 4:* If you run the same query as Example 3 only replace ROLLUP with CUBE, you can see additional grouping sets for (WEEK,SALES_PERSON), (DAY_WEEK,SALES_PERSON), (DAY_WEEK), (SALES_PERSON) in the result.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
  FROM SALES
  WHERE WEEK(SALES_DATE) = 13
  GROUP BY CUBE ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
  ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - GOUNOT                   32
         13           - LEE                      33
```

```
            13           - LUCCHESSI          8
            13           - -                 73
             -         6 GOUNOT             11
             -         6 LEE                12
             -         6 LUCCHESSI           4
             -         6 -                  27
             -         7 GOUNOT             21
             -         7 LEE                21
             -         7 LUCCHESSI           4
             -         7 -                  46
             -           - GOUNOT           32
             -           - LEE              33
             -           - LUCCHESSI         8
             -           - -                73
```

- *Example 5:* Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

  ```
  SELECT SALES_PERSON,
         MONTH(SALES_DATE) AS MONTH,
         SUM(SALES) AS UNITS_SOLD
  FROM SALES
  GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)),
                           ()
                         )
  ORDER BY SALES_PERSON, MONTH
  ```

  This results in:

  ```
  SALES_PERSON    MONTH       UNITS_SOLD
  --------------- ----------- -----------
  GOUNOT                   3          35
  GOUNOT                   4          14
  GOUNOT                  12           1
  LEE                      3          60
  LEE                      4          25
  LEE                     12           6
  LUCCHESSI                3           9
  LUCCHESSI                4           4
  LUCCHESSI               12           1
  -                        -         155
  ```

- *Example 6:* This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.

  - *Example 6-1:*

    ```
    SELECT WEEK(SALES_DATE) AS WEEK,
           DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
           SUM(SALES) AS UNITS_SOLD
    FROM SALES
    GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
    ORDER BY WEEK, DAY_WEEK
    ```

    results in:

    ```
    WEEK        DAY_WEEK    UNITS_SOLD
    ----------- ----------- -----------
             13           6          27
             13           7          46
             13           -          73
             14           1          31
             14           2          43
             14           -          74
             53           1           8
             53           -           8
              -           -         155
    ```

  - *Example 6-2:*

Examples of grouping sets, cube, and rollup queries

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
  FROM SALES
  GROUP BY ROLLUP ( MONTH(SALES_DATE), REGION );
  ORDER BY MONTH, REGION
```

results in:

```
MONTH       REGION          UNITS_SOLD
----------- --------------- -----------
          3 Manitoba                 22
          3 Ontario-North             8
          3 Ontario-South            34
          3 Quebec                   40
          3 -                       104
          4 Manitoba                 17
          4 Ontario-North             1
          4 Ontario-South            14
          4 Quebec                   11
          4 -                        43
         12 Manitoba                  2
         12 Ontario-South             4
         12 Quebec                    2
         12 -                         8
          - -                       155
```

– *Example 6-3:*

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
  FROM SALES
  GROUP BY GROUPING SETS ( ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
                           ROLLUP( MONTH(SALES_DATE), REGION )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
13	6	-	-	27
13	7	-	-	46
13	-	-	-	73
14	1	-	-	31
14	2	-	-	43
14	-	-	-	74
53	1	-	-	8
53	-	-	-	8
-	-	3	Manitoba	22
-	-	3	Ontario-North	8
-	-	3	Ontario-South	34
-	-	3	Quebec	40
-	-	3	-	104
-	-	4	Manitoba	17
-	-	4	Ontario-North	1
-	-	4	Ontario-South	14
-	-	4	Quebec	11
-	-	4	-	43
-	-	12	Manitoba	2
-	-	12	Ontario-South	4
-	-	12	Quebec	2
-	-	12	-	8
-	-	-	-	155
-	-	-	-	155

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Examples of grouping sets, cube, and rollup queries

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end.
- In the second grouped set, month 12 has now been positioned to the end and the regions now display in alphabetic order.
- Null values are sorted high.

- *Example 7:* In queries that perform multiple ROLLUPs in a single pass (such as Example 6-3) you might want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the *origin* of each row in the result set. *Origin* means which one of the two grouping sets produced the row in the result set.

Step 1: Introduce a way of "generating" new data values, using a query which selects from a VALUES clause (which is an alternative form of a fullselect). This query shows how a table can be derived called "X" having 2 columns "R1" and "R2" and 1 row of data.

```
SELECT R1,R2
FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
```

results in:

```
R1      R2
------- -------
GROUP 1 GROUP 2
```

Step 2: Form the cross product of this table "X" with the SALES table. This add columns "R1" and "R2" to every row.

```
SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SALES AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
```

This add columns "R1" and "R2" to every row.

Step 3: Now these columns can be combined with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                             DAYOFWEEK(SALES_DATE))),
                        (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

R1	R2	WEEK	DAY_WEEK	MONTH	REGION	UNITS_SOLD
GROUP 1	-	13	6	-	-	27
GROUP 1	-	13	7	-	-	46
GROUP 1	-	13	-	-	-	73
GROUP 1	-	14	1	-	-	31
GROUP 1	-	14	2	-	-	43
GROUP 1	-	14	-	-	-	74
GROUP 1	-	53	1	-	-	8
GROUP 1	-	53	-	-	-	8
-	GROUP 2	-	-	3	Manitoba	22
-	GROUP 2	-	-	3	Ontario-North	8
-	GROUP 2	-	-	3	Ontario-South	34
-	GROUP 2	-	-	3	Quebec	40
-	GROUP 2	-	-	3	-	104
-	GROUP 2	-	-	4	Manitoba	17

```
-        GROUP 2     -       -        4 Ontario-North          1
-        GROUP 2     -       -        4 Ontario-South         14
-        GROUP 2     -       -        4 Quebec                11
-        GROUP 2     -       -        4 -                     43
-        GROUP 2     -       -       12 Manitoba               2
-        GROUP 2     -       -       12 Ontario-South          4
-        GROUP 2     -       -       12 Quebec                 2
-        GROUP 2     -       -       12 -                      8
-        GROUP 2     -       -        - -                    155
GROUP 1  -           -       -        - -                    155
```

Step 4: Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
  FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
  GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                              DAYOFWEEK(SALES_DATE))),
                          (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
  ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;
```

results in:

```
GROUP    WEEK      DAY_WEEK     MONTH       REGION           UNITS_SOLD
-------  --------- ----------   ----------- ---------------- -----------
GROUP 1      13        6         - -                            27
GROUP 1      13        7         - -                            46
GROUP 1      13        -         - -                            73
GROUP 1      14        1         - -                            31
GROUP 1      14        2         - -                            43
GROUP 1      14        -         - -                            74
GROUP 1      53        1         - -                             8
GROUP 1      53        -         - -                             8
GROUP 1      -         -         - -                           155
GROUP 2      -         -         3 Manitoba                     22
GROUP 2      -         -         3 Ontario-North                 8
GROUP 2      -         -         3 Ontario-South                34
GROUP 2      -         -         3 Quebec                       40
GROUP 2      -         -         3 -                           104
GROUP 2      -         -         4 Manitoba                     17
GROUP 2      -         -         4 Ontario-North                 1
GROUP 2      -         -         4 Ontario-South                14
GROUP 2      -         -         4 Quebec                       11
GROUP 2      -         -         4 -                            43
GROUP 2      -         -        12 Manitoba                      2
GROUP 2      -         -        12 Ontario-South                 4
GROUP 2      -         -        12 Quebec                        2
GROUP 2      -         -        12 -                             8
GROUP 2      -         -         - -                           155
```

- *Example 8:* The following example illustrates the use of various aggregate functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD,
       MAX(SALES) AS BEST_SALE,
```

Examples of grouping sets, cube, and rollup queries

```
              CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
    FROM SALES
    GROUP BY CUBE(MONTH(SALES_DATE),REGION)
    ORDER BY MONTH, REGION
```

This results in:

MONTH	REGION	UNITS_SOLD	BEST_SALE	AVG_UNITS_SOLD
3	Manitoba	22	7	3.14
3	Ontario-North	8	3	2.67
3	Ontario-South	34	14	4.25
3	Quebec	40	18	5.00
3	-	104	18	4.00
4	Manitoba	17	9	5.67
4	Ontario-North	1	1	1.00
4	Ontario-South	14	8	4.67
4	Quebec	11	8	5.50
4	-	43	9	4.78
12	Manitoba	2	2	2.00
12	Ontario-South	4	3	2.00
12	Quebec	2	1	1.00
12	-	8	3	1.60
-	Manitoba	41	9	3.73
-	Ontario-North	9	3	2.25
-	Ontario-South	52	14	4.00
-	Quebec	53	18	4.42
-	-	155	18	3.87

having-clause

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true.

R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered to be a single group with no grouping columns.

```
►►──HAVING──search-condition──────────────────────────────────────────────►◄
```

Each *column-name* in the search condition must satisfy one of the following conditions:
- Unambiguously identify a grouping column of R.
- Be specified within an aggregate function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each aggregate function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see Example 6 and Example 7 in "Examples of subselect queries" on page 838.

A correlated reference to a group of R must either identify a grouping column or be contained within an aggregate function.

When HAVING is used without GROUP BY, the select list can only include column names when they are arguments to an aggregate function, correlated column references, global variables, host variables, literals, special registers, SQL variables, or SQL parameters.
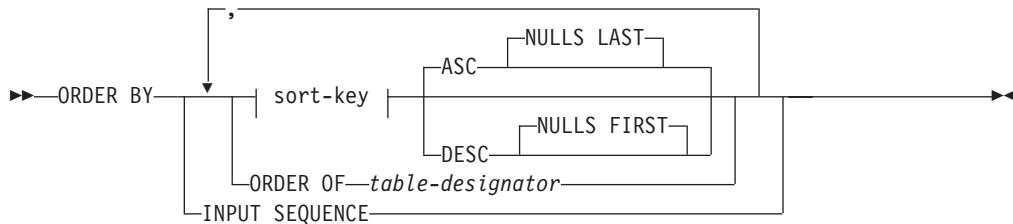
Note: The following expressions can only be specified in a HAVING clause if they are contained within an aggregate function (SQLSTATE 42803):
- ROW CHANGE TIMESTAMP FOR *table-designator*
- ROW CHANGE TOKEN FOR *table-designator*
- RID_BIT or RID scalar function

order-by-clause

The ORDER BY clause specifies an ordering of the rows of the result table.

sort-key:

If a single sort specification (one *sort-key* with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. Each *sort-key* cannot have a data type of CLOB, DBCLOB, BLOB, XML, distinct type on any of these types, or structured type (SQLSTATE 42907).

A named column in the select list can be identified by a *sort-key* that is a *simple-integer* or a *simple-column-name*. An unnamed column in the select list must be identified by an *simple-integer* or, in some cases, by a *sort-key-expression* that matches the expression in the select list (see details of *sort-key-expression*). A column is unnamed if the AS clause is not specified and it is derived from a constant, an expression with operators, or a function.

Ordering is performed in accordance with comparison rules. If an ORDER BY clause contains decimal floating-point columns, and multiple representations of the same number exist in these columns, the ordering of the multiple representations of the same number is unspecified. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

simple-column-name
> Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

> The *simple-column-name* can also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. This includes columns defined as implicitly hidden. An error occurs in the following situations:

> - If the subselect specifies DISTINCT in the select-clause (SQLSTATE 42822)
> - If the subselect produces a grouped result and the *simple-column-name* is not a *grouping-expression* (SQLSTATE 42803)

> Determining which column is used for ordering the result is described under "Column names in sort keys" in the "Notes" section.

simple-integer

Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer *n* identifies the *n*th column of the result table.

sort-key-expression

An expression that is not simply a column name or an unsigned integer constant. The query to which ordering is applied must be a *subselect* to use this form of sort-key. The *sort-key-expression* cannot include a correlated scalar fullselect (SQLSTATE 42703) or a function with an external action (SQLSTATE 42845).

Any *column-name* within a *sort-key-expression* must conform to the rules described under "Column names in sort keys" in the "Notes" section.

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).

  The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).

- The subselect is grouped (SQLSTATE 42803).

  The sort-key-expression can:

  – be an expression in the select list of the subselect,

  – include a *grouping-expression* from the GROUP BY clause of the subselect

  – include an aggregate function, constant or host variable.

ASC

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* applies to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:

```
(SELECT C1 FROM T1
   ORDER BY C1)
UNION
SELECT C1 FROM T2
   ORDER BY ORDER OF T1
```

The following example *is* valid:

```
SELECT C1 FROM
   (SELECT C1 FROM T1
      UNION
    SELECT C1 FROM T2
    ORDER BY C1 ) AS UTABLE
ORDER BY ORDER OF UTABLE
```

INPUT SEQUENCE
> Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See "table-reference" on page 792. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

Notes

- **Column names in sort keys**:
  - The column name is qualified.

    The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.
  - The column name is unqualified.
    - The query is a subselect.

      If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).
    - The query is not a subselect (it includes set operations such as union, except or intersect).
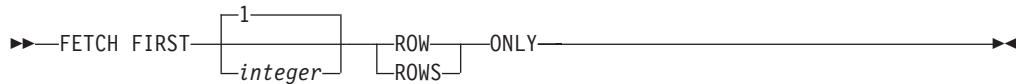
      The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.
- **Limits**: The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list might result in the addition of the column or expression to the temporary table used for sorting. This might result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

fetch-first-clause

The *fetch-first-clause* sets a maximum number of rows that can be retrieved.

```
                          ┌─1──────┐
►►──FETCH FIRST───┤        ├──┬─ROW──┬──ONLY──────────────────────────►◄
                          └─integer─┘  └─ROWS─┘
```

The *fetch-first-clause* 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.
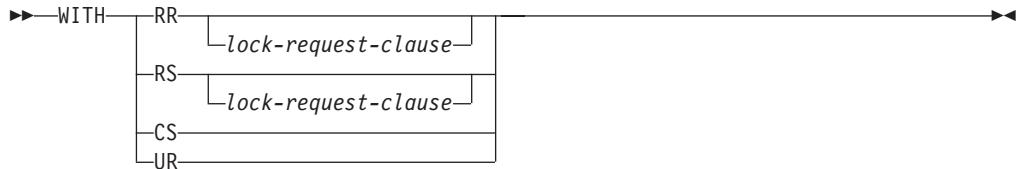
Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query when it has determined the first *integer* rows. If both the *fetch-first-clause* and the *optimize-for-clause* are specified, the lower of the *integer* values from these 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.

isolation-clause (subselect query)

The optional *isolation-clause* specifies the isolation level at which the subselect or fullselect is run, and whether a specific type of lock is to be acquired.

```
►►─WITH─────RR──────────────────────────────────────────────────────►◄
            └─lock-request-clause─┘
        ─RS─────────────────────
            └─lock-request-clause─┘
        ─CS─────────────────────
        ─UR─────────────────────
```

- RR - Repeatable Read (row-organized tables only)
- RS - Read Stability (row-organized tables only)
- CS - Cursor Stability
- UR - Uncommitted Read

lock-request-clause

```
►►─USE AND KEEP─────SHARE───────LOCKS──────────────────────────────►◄
                ─UPDATE───
                └EXCLUSIVE─┘
```

The *lock-request-clause* applies only to queries and to positioning read operations within an insert, update, or delete operation. The insert, update, and delete operations themselves will run using locking determined by the database manager.

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold:

SHARE
> Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE
> Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

EXCLUSIVE
> Concurrent processes cannot acquire a lock on the data.

isolation-clause **restrictions**:

- The *isolation-clause* is not supported for a CREATE TABLE, CREATE VIEW, or ALTER TABLE statement (SQLSTATE 42601).
- The *isolation-clause* cannot be specified for a subselect or fullselect that will cause trigger invocation, referential integrity scans, or MQT maintenance (SQLSTATE 42601).
- A subselect or fullselect cannot include a *lock-request-clause* if that subselect or fullselect references any SQL functions that are not declared with the option INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601).
- A subselect or fullselect that contains a *lock-request-clause* are not be eligible for MQT routing.
- If an *isolation-clause* is specified for a *subselect* or *fullselect* within the body of an SQL function, SQL method, or trigger, the clause is ignored and a warning is returned.

- If an *isolation-clause* is specified for a *subselect* or *fullselect* that is used by a scrollable cursor, the clause is ignored and a warning is returned.
- Neither *isolation-clause* nor *lock-request-clause* can be specified in the context where they will cause conflict isolation or lock intent on a common table expression (SQLSTATE 42601). This restriction does not apply to aliases or base tables. The following examples create an isolation conflict on *a* and returns an error:
  - View:
    ```
    create view a as (...);
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```
  - Common table expression:
    ```
    WITH a as (...)
    (select * from a with RR USE AND KEEP SHARE LOCKS)
    UNION ALL
    (select * from a with UR);
    ```
- An *isolation-clause* cannot be specified in an XML context (SQLSTATE 2200M).
- The WITH clause specifying isolation cannot be specified at a subselect level in any statement that accesses a column-organized table.

Examples of subselect queries

The following examples illustrate the susbelect query.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

  ```
  SELECT * FROM EMPLOYEE
  ```

- *Example 2:* Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

  ```
  SELECT EMP_ACT.*, LASTNAME
    FROM EMP_ACT, EMPLOYEE
    WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
  ```

- *Example 3:* Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1955.

  ```
  SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
    FROM EMPLOYEE, DEPARTMENT
    WHERE WORKDEPT = DEPTNO
    AND YEAR(BIRTHDATE) < 1955
  ```

- *Example 4:* Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

  ```
  SELECT JOB, MIN(SALARY), MAX(SALARY)
    FROM EMPLOYEE
    GROUP BY JOB
    HAVING COUNT(*) > 1
    AND MAX(SALARY) >= 27000
  ```

- *Example 5:* Select all the rows of EMP_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

  ```
  SELECT *
    FROM EMP_ACT
    WHERE EMPNO IN
            (SELECT EMPNO
               FROM EMPLOYEE
               WHERE WORKDEPT = 'E11')
  ```

- *Example 6:* From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

  ```
  SELECT WORKDEPT, MAX(SALARY)
    FROM EMPLOYEE
    GROUP BY WORKDEPT
    HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                            FROM EMPLOYEE)
  ```

  The subquery in the HAVING clause is executed once in this example.

- *Example 7:* Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

  ```
  SELECT WORKDEPT, MAX(SALARY)
    FROM EMPLOYEE EMP_COR
      GROUP BY WORKDEPT
      HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                              FROM EMPLOYEE
                              WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
  ```

In contrast to Example 6, the subquery in the HAVING clause is executed for each group.

- *Example 8:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

  This query must first create a nested table expression (DINFO) to get the AVGSALARY and EMPCOUNT columns, and the DEPTNO column that is used in the WHERE clause.

```
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
  FROM EMPLOYEE THIS_EMP,
       (SELECT OTHERS.WORKDEPT AS DEPTNO,
                          AVG(OTHERS.SALARY) AS AVGSALARY,
                          COUNT(*) AS EMPCOUNT
          FROM EMPLOYEE OTHERS
          GROUP BY OTHERS.WORKDEPT
        ) AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
   AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

  Using a nested table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the query, only the rows for the department of the sales representatives are considered by the view.

- *Example 9:* Display the average education level and salary for 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 10:* Query the EMP_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

```
SELECT EMP_ACT.EMPNO,PROJNO
  FROM EMP_ACT
  WHERE EMP_ACT.EMPNO IN
       (SELECT EMPLOYEE.EMPNO
          FROM EMPLOYEE
          ORDER BY SALARY DESC
          FETCH FIRST 10 ROWS ONLY)
```

- *Example 11:* Assuming that PHONES and IDS are two SQL variables with array values of the same cardinality, turn these arrays into a table with three columns (one for each array and one for the position), and one row per array element.
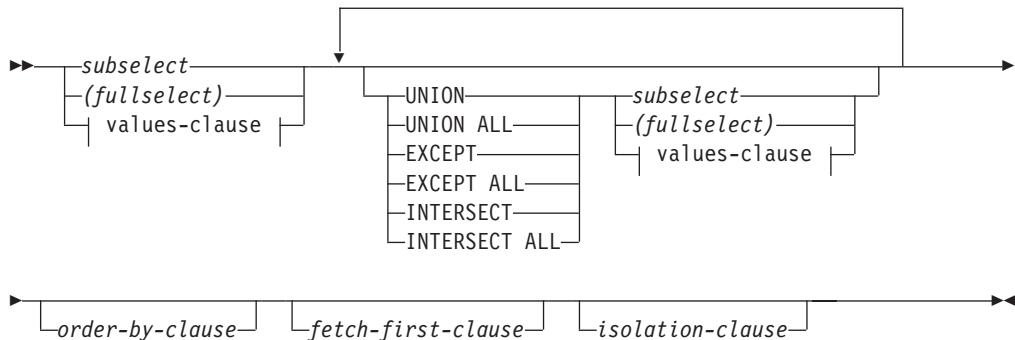
```
SELECT T.PHONE, T.ID, T.INDEX FROM UNNEST(PHONES, IDS)
  WITH ORDINALITY AS T(PHONE, ID, INDEX)
  ORDER BY T.INDEX
```

fullselect

The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement.
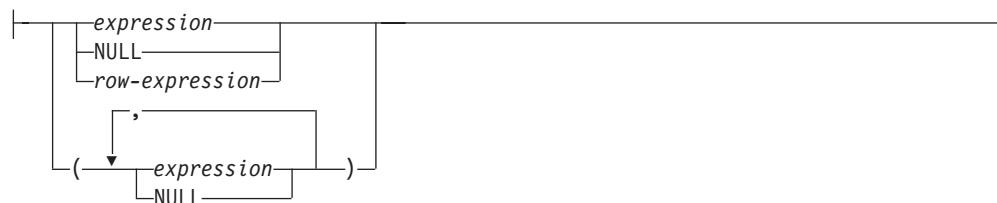
A fullselect that is a component of a predicate is called a *subquery*, and a fullselect that is enclosed in parentheses is sometimes called a subquery.

values-clause:

values-row:

The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or values-clause.

The authorization for a *fullselect* is described in the Authorization section in "SQL queries".

values-clause
> Derives a result table by specifying the actual values, using expressions or row expressions, for each column of a row in the result table. Multiple rows may be specified. If multiple rows are specified, the extended indicator variable values

of DEFAULT and UNASSIGNED must not be used (SQLSTATE 22539). The result type of any expression in the *values-clause* cannot be a row type (SQLSTATE 428H2).

NULL can only be used with multiple specifications of *values-row*, either as the column value of a single column result table or within a *row-expression*, and at least one row in the same column must not be NULL (SQLSTATE 42608).

A *values-row* is specified by:

- A single expression for a single column result table
- n expressions (or NULL) separated by commas and enclosed in parentheses, where n is the number of columns in the result table or, a row expression for a multiple column result table.

A multiple row VALUES clause must have the same number of columns in each *values-row* (SQLSTATE 42826).

The following examples show *values-clause* and their meaning.

```
VALUES (1),(2),(3)        - 3 rows of 1 column
VALUES 1, 2, 3            - 3 rows of 1 column
VALUES (1, 2, 3)          - 1 row of 3 columns
VALUES (1,21),(2,22),(3,23) - 3 rows of 2 columns
```

A *values-clause* that is composed of n specifications of *values-row*, RE_1 to RE_n, where n is greater than 1, is equivalent to:

```
RE₁ UNION ALL RE₂ ... UNION ALL REₙ
```

This means that the corresponding columns of each *values-row* must be comparable (SQLSTATE 42825).

UNION or UNION ALL

Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

EXCEPT or EXCEPT ALL

Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

For compatibility with other SQL implementations, MINUS can be specified as a synonym for EXCEPT.

INTERSECT or INTERSECT ALL

Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

order-by-clause

See "subselect" for details of the *order-by-clause*. A fullselect that contains an ORDER BY clause cannot be specified in (SQLSTATE 428FJ):

- A materialized query table
- The outermost fullselect of a view

> **Note:** An ORDER BY clause in a fullselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

fetch-first-clause
> See "subselect" for details of the *fetch-first-clause*. A fullselect that contains a FETCH FIRST clause cannot be specified in (SQLSTATE 428FJ):
> - A materialized query table
> - The outermost fullselect of a view
>
> **Note:** A FETCH FIRST clause in a fullselect does not affect the number of rows returned by a query. A FETCH FIRST clause only affects the number of rows returned if it is specified in the outermost fullselect.

isolation-clause
> See "subselect" for details of the *isolation-clause*. If *isolation-clause* is specified for a fullselect and it could apply equally to a subselect of the fullselect, *isolation-clause* is applied to the fullselect. For example, consider the following query.
>
> ```
> SELECT NAME FROM PRODUCT
> UNION
> SELECT NAME FROM CATALOG
> WITH UR
> ```
>
> Even though the isolation clause WITH UR could apply only to the subselect SELECT NAME FROM CATALOG, it is applied to the whole fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not include any columns having a data type of CLOB, DBCLOB, BLOB, distinct type on any of these types, or structured type (SQLSTATE 42907).

The column name of the *n*th column of the result table is the name of the *n*th column of R1 if it is named. Otherwise, the *n*th column of the result table is unnamed. If the fullselect is used as a select-statement, a generated name is provided when the statement is described. The generated name cannot be used in other parts of the SQL statement such as the ORDER BY clause or the UPDATE clause. The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

Duplicate rows: Two rows are duplicates if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal, and two decimal floating-point representations of the same number are considered equal. For example, 2.00 and 2.0 have the same value (2.00 and 2.0 compare as equal) but have different exponents, which allows you to represent both 2.00 and 2.0. So, for example, if the result table of a UNION operation contains a decimal floating-point column and multiple representations of the same number exist, the one that is returned (for example, 2.00 or 2.0) is unpredictable. For more information, see "Numeric comparisons" on page 136.

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

R1	R2	UNION ALL	UNION	EXCEPT ALL	EXCEPT	INTER-SECT ALL	INTER-SECT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					
		4					
		4					
		4					
		5					

Examples of fullselect queries

The following examples illustrate fullselect queries.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

- *Example 2:* List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E' **or** who are assigned to projects in the EMP_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
    FROM EMP_ACT
    WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 3:* Make the same query as in example 2, and, in addition, "tag" the rows from the EMPLOYEE table with 'emp' and the rows from the EMP_ACT table with 'emp_act'. Unlike the result from example 2, this query might return the same EMPNO more than once, identifying which table it came from by the associated "tag".

```
SELECT EMPNO, 'emp'
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
    WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 4:* Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```
SELECT EMPNO
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
    FROM EMP_ACT
    WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

- *Example 5:* Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as "new".

```
SELECT EMPNO, 'emp'
    FROM EMPLOYEE
    WHEREWORKDEPTLIKE 'E%'
UNION
SELECT EMPNO, 'emp_act'
    FROM EMP_ACT
    WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
UNION
    VALUES ('NEWAAA', 'new'), ('NEWBBB', 'new')
```

- *Example 6:* This example of EXCEPT produces all rows that are in T1 but not in T2.

```
(SELECT * FROM T1)
EXCEPT ALL
(SELECT * FROM T2)
```

If no null values are involved, this example returns the same results as

```
SELECT ALL *
    FROM T1
    WHERE NOT EXISTS (SELECT * FROM T2
                          WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
```

- *Example 7:* This example of INTERSECT produces all rows that are in both
  tables T1 and T2, removing duplicates.

  ```
  (SELECT * FROM T1)
  INTERSECT
  (SELECT * FROM T2)
  ```

  If no null values are involved, this example returns the same result as

  ```
  SELECT DISTINCT * FROM T1
     WHERE EXISTS (SELECT * FROM T2
                       WHERE T1.C1 = T2.C1 AND T1.C2 = T2.C2 AND...)
  ```
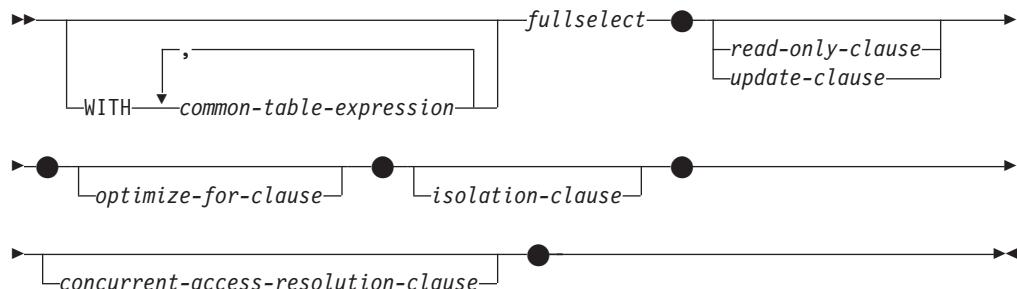
  where C1, C2, and so on represent the columns of T1 and T2.

select-statement

The *select-statement* is the form of a query that can be specified in a DECLARE CURSOR statement, either directly, or prepared and then referenced. It can also be issued through the use of dynamic SQL statements, causing a result table to be displayed on the user's screen.

The table specified by a *select-statement* is the result of the fullselect.
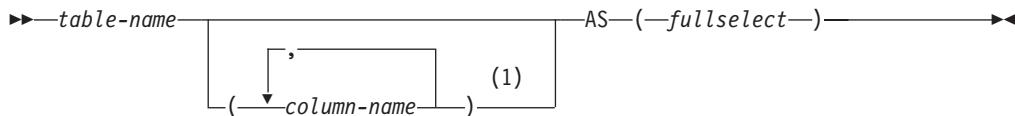
The authorization for a *select-statement* is described in the Authorization section in "SQL queries".

For details about the clauses in the select-statement query, refer to the following topics:
- "common-table-expression" on page 847
- "update-clause" on page 853
- "read-only-clause" on page 854
- "optimize-for-clause" on page 855
- "isolation-clause (select-statement query)" on page 856
- "lock-request-clause" on page 857
- "concurrent-access-resolution-clause" on page 858

common-table-expression

A *common table expression* permits defining a result table with a *table-name* that can be specified as a table name in any FROM clause of the fullselect that follows.

```
►►──table-name─────────────────────────────────AS──(──fullselect──)──────────────►◄
                    │                    (1)│
                    └─(──▼──column-name──┬──)──┘
                             └────,───────┘
```

Notes:

1   If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.

If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the fullselect used to define the common table expression.

The *table-name* of a common table expression must be different from any other common table expression *table-name* in the same statement (SQLSTATE 42726). If the common table expression is specified in an INSERT statement the *table-name* cannot be the same as the table or view name that is the object of the insert (SQLSTATE 42726). A common table expression *table-name* can be specified as a table name in any FROM clause throughout the fullselect. A *table-name* of a common table expression overrides any existing table, view or alias (in the catalog) with the same qualified name.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted (SQLSTATE 42835). A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*.

If the fullselect of a common table expression contains a *data-change-table-reference* in the FROM clause, the common table expression is said to modify data. A common table expression that modifies data is always evaluated when the statement is processed, regardless of whether the common table expression is used anywhere else in the statement. If there is at least one common table expression that reads or modifies data, all common table expressions are processed in the order in which they occur, and each common table expression that reads or modifies data is completely executed, including all constraints and triggers, before any subsequent common table expressions are executed.

The common table expression is also optional before to the fullselect in the CREATE VIEW and INSERT statements.

A common table expression can be used in the following situations:
* In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
* To enable grouping by a column that is derived from a scalar subselect or function that is not deterministic or has external action

- When the required result table is based on host variables
- When the same result table must be shared in a fullselect
- When the result must be derived using recursion
- When multiple SQL data change statements must be processed within the query

If the fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:
- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the *table-name* of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types can differ. In this case, the column in the iterative fullselect must have a length that will always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any aggregate functions, group-by-clauses, or having-clauses (SQLSTATE 42836).

  The FROM clauses of these fullselects can include at most one reference to a common table expression that is part of a recursion cycle (SQLSTATE 42836).
- The iterative fullselect and the overall recursive fullselect must not include an order-by-clause (SQLSTATE 42836).
- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will stop. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:
- In the iterative fullselect, an integer column incremented by a constant.
- A predicate in the where clause of the iterative fullselect in the form "counter_col < constant" or "counter _col < :hostvar".

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

Recursion example: bill of materials

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part.

For this example, create the table as follows:

```
CREATE TABLE PARTLIST
            (PART VARCHAR(8),
             SUBPART VARCHAR(8),
             QUANTITY INTEGER);
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
00       01                5
00       05                3
01       02                2
01       03                3
01       04                4
01       06                3
02       05                7
02       06                6
03       07                6
04       08               10
04       09               11
05       10               10
05       11               10
06       12               10
06       13               10
07       14                8
07       12                8
```

Example 1: Single level explosion

The first example is called single level explosion. It answers the question, "What parts are needed to build the part identified by '01'?". The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
    (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
       FROM PARTLIST ROOT
       WHERE ROOT.PART = '01'
     UNION ALL
       SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
       FROM RPL PARENT, PARTLIST CHILD
       WHERE  PARENT.SUBPART = CHILD.PART
    )
SELECT DISTINCT PART, SUBPART, QUANTITY
 FROM RPL
  ORDER BY PART, SUBPART, QUANTITY;
```

The preceding query includes a common table expression, identified by the name *RPL*, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the *initialization fullselect*, gets the direct children of part '01'. The FROM clause of this fullselect refers to the source table and will never reference itself (*RPL* in this case). The result of this first

fullselect goes into the common table expression *RPL* (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses *RPL* to compute subparts of subparts by having the FROM clause refer to the common table expression *RPL* and the source table with a join of a part from the source table (child) to a subpart of the current result contained in *RPL* (parent). The result goes back to *RPL* again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
01       02                2
01       03                3
01       04                4
01       06                3
02       05                7
02       06                6
03       07                6
04       08               10
04       09               11
05       10               10
05       11               10
06       12               10
06       13               10
07       12                8
07       14                8
```

Observe in the result that part '01' goes to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.

It is important to remember that with recursive common table expressions it is possible to introduce an *infinite loop*. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as:

```
    PARENT.SUBPART = CHILD.SUBPART
```

This example of causing an infinite loop is obviously a case of not coding what is intended. Exercise care when determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query can be produced in an application program without using a recursive common table expression. However, this approach would require starting of a new query for every level of recursion. Furthermore, the application needs to put all the results back in the database to order the result. This approach complicates the application logic and does not perform well. The application logic becomes even harder and more inefficient for other bill of material queries, such as summarized and indented explosion queries.

Example 2: Summarized explosion

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part '01'. The main difference from the single level explosion is the requirement to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part '01'.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
   (
      SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
        FROM PARTLIST ROOT
        WHERE ROOT.PART = '01'
    UNION ALL
      SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
        FROM RPL PARENT, PARTLIST CHILD
        WHERE PARENT.SUBPART = CHILD.PART
   )
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
 FROM RPL
  GROUP BY PART, SUBPART
  ORDER BY PART, SUBPART;
```

In the preceding query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name *RPL*, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping over the common table expression *RPL* and using the SUM aggregate function in the select list of the main fullselect.

The result of the query is as follows:

```
PART     SUBPART  Total Qty Used
-------- -------- --------------
01       02                    2
01       03                    3
01       04                    4
01       05                   14
01       06                   15
01       07                   18
01       08                   40
01       09                   44
01       10                  140
01       11                  140
01       12                  294
01       13                  150
01       14                  144
```

Looking at the output, consider the line for subpart '06'. The total quantity used value of 15 is derived from a quantity of 3 directly for part '01' and a quantity of 6 for part '02' which is needed 2 times by part '01'.

Example 3: Controlling depth

The question might come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, "What are the first two levels of parts needed to build the part identified by '01'?" For the sake of clarity in the example, the level is included in the result.

Recursion example: bill of materials

```
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
    (
        SELECT 1,                 ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
          FROM PARTLIST ROOT
          WHERE ROOT.PART = '01'
        UNION ALL
          SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
            FROM RPL PARENT, PARTLIST CHILD
            WHERE PARENT.SUBPART = CHILD.PART
              AND PARENT.LEVEL < 2
    )
 SELECT PART, LEVEL, SUBPART, QUANTITY
   FROM RPL;
```
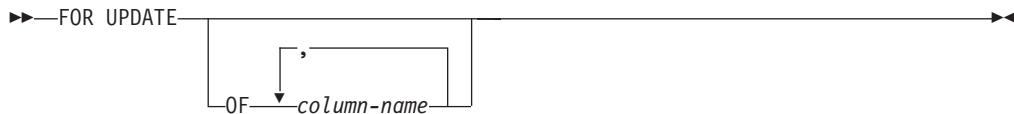
This query is similar to example 1. The column *LEVEL* was introduced to count the levels from the original part. In the initialization fullselect, the value for the *LEVEL* column is initialized to 1. In the subsequent fullselect, the level from the parent is incremented by 1. Then to control the number of levels in the result, the second fullselect includes the condition that the parent level must be less than 2. This ensures that the second fullselect only processes children to the second level.

The result of the query is:

```
PART     LEVEL       SUBPART  QUANTITY
-------- ----------- -------- -----------
01                 1 02                 2
01                 1 03                 3
01                 1 04                 4
01                 1 06                 3
02                 2 05                 7
02                 2 06                 6
03                 2 07                 6
04                 2 08                10
04                 2 09                11
06                 2 12                10
06                 2 13                10
```

update-clause

The FOR UPDATE clause identifies the columns that can be targets in an assignment clause in a subsequent positioned UPDATE statement. Each *column-name* must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect.

```
►►──FOR UPDATE─────────────────────────────────────────────────────►◄
                 │         ┌─,──────────┐         │
                 └─OF──────▼─column-name─┴─────────┘
```

If a FOR UPDATE clause is specified with a *column-name* list, and extended indicator variables are not enabled, then *column-name* must be an updatable column (SQLSTATE 42808).

If a FOR UPDATE clause is specified without a *column-name* list, then the implicit *column-name* list is determined as follows:
- If extended indicator variables are enabled, all of the columns of the table or view identified in the first FROM clause of the fullselect are included.
- If extended indicator variables are not enabled, all of the updatable columns of the table or view identified in the first FROM clause of the fullselect are included.
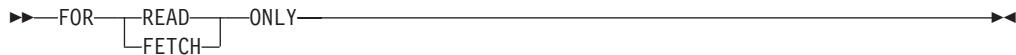
The FOR UPDATE clause cannot be used if one of the following conditions is true:
- The cursor associated with the select-statement is not deletable .
- One of the selected columns is a non-updatable column of a catalog table and the FOR UPDATE clause has not been used to exclude that column.

read-only-clause

The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

```
►►─FOR──┬─READ──┬─ONLY─────────────────────────────────────────────►◄
        └─FETCH─┘
```

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause were specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance, except in cases where queries will be used in positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY).

optimize-for-clause

The OPTIMIZE FOR clause requests special processing of the *select statement*.

```
►►──OPTIMIZE FOR──integer──┬─ROWS─┬──────────────────────────────────────────►◄
                           └─ROW──┘
```

If this clause is omitted, it is assumed that all rows of the result table will be retrieved; if it is specified, it is assumed that the number of rows retrieved will probably not exceed *n*, where *n* is the value of *integer*. The value of *n* must be a positive integer (not zero). Use of the OPTIMIZE FOR clause influences query optimization, based on the assumption that *n* rows will be retrieved. In addition, for cursors that are blocked, this clause will influence the number of rows that will be returned in each block (that is, no more than *n* rows will be returned in each block). If both the *fetch-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.
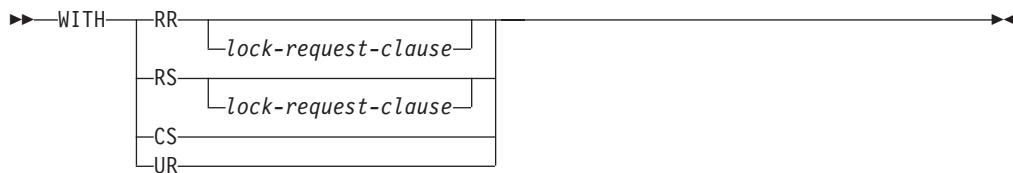
This clause does not limit the number of rows that can be fetched, or affect the result in any other way than performance. Using OPTIMIZE FOR *n* ROWS can improve performance if no more than *n* rows are retrieved, but might degrade performance if more than *n* rows are retrieved.

If the value of *n* multiplied by the size of the row exceeds the size of the communication buffer, the OPTIMIZE FOR clause will have no affect on the data buffers. The size of the communication buffer is defined by the **rqrioblk** or the **aslheapsz** configuration parameter.

isolation-clause (select-statement query)

The optional *isolation-clause* specifies the isolation level at which the statement is executed, and whether a specific type of lock is to be acquired.

```
>>─WITH─┬─RR──────────────────────┬──────────────────────────────────><
        │    └─lock-request-clause─┘ │
        ├─RS──────────────────────┤
        │    └─lock-request-clause─┘ │
        ├─CS──────────────────────┤
        └─UR──────────────────────┘
```

- 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 IBM family and Microsoft SQL Server data sources, the *isolation-clause* can be included in the statement to specify the statement isolation level. If the *isolation-clause* is included in statements that access other data sources, the specified isolation level is ignored. The current isolation level on the federated server is mapped to a corresponding isolation level at the data source on each connection to the data source. After a connection is made to a data source, the isolation level cannot be changed for the duration of the connection.

lock-request-clause

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold.

```
►►─USE AND KEEP──┬─SHARE─────┬──LOCKS──────────────────────────────────────►◄
                 ├─UPDATE────┤
                 └─EXCLUSIVE─┘
```

SHARE

Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE

Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

EXCLUSIVE

Concurrent processes cannot acquire a lock on the data.

The *lock-request-clause* applies to all base table and index scans required by the query, including those within subqueries, SQL functions and SQL methods. It has no affect on locks placed by procedures, external functions, or external methods. Any SQL function or SQL method called (directly or indirectly) by the statement must be created with INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601). The *lock-request-clause* cannot be used with a modifying query that might activate triggers or that requires referential integrity checks (SQLSTATE 42601).

concurrent-access-resolution-clause

The optional *concurrent-access-resolution-clause* specifies the concurrent access resolution to use for *select-statement*.

```
►►──WAIT FOR OUTCOME──────────────────────────────────────────────►◄
```

WAIT FOR OUTCOME specifies to wait for the commit or rollback when encountering data in the process of being updated or deleted. Rows encountered that are in the process of being inserted are not skipped. The settings for the registry variables DB2_EVALUNCOMMITTED, DB2_SKIPDELETED, and DB2_SKIPINSERTED are ignored. This clause applies when the isolation level is CS or RS and is ignored when an isolation level of UR or RR is in effect.

This clause causes the following behavior and settings to be overridden:

- The default behavior for currently committed that is defined by the `cur_commit` configuration parameter.
- Any higher level setting such as bind options, CLI settings, JDBC settings, or lock modifications.

Examples of select-statement queries

The following examples illustrate the select-statement query.

- *Example 1:* Select all columns and rows from the EMPLOYEE table.

      SELECT * FROM EMPLOYEE

- *Example 2:* Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

      SELECT PROJNAME, PRSTDATE, PRENDATE
        FROM PROJECT
        ORDER BY PRENDATE DESC

- *Example 3:* Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

      SELECT WORKDEPT, AVG(SALARY)
          FROM EMPLOYEE
          GROUP BY WORKDEPT
          ORDER BY 2

- *Example 4:* Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

      EXEC SQL DECLARE UP_CUR CURSOR FOR
                  SELECT PROJNO, PRSTDATE, PRENDATE
                    FROM PROJECT
                    FOR UPDATE OF PRSTDATE, PRENDATE;

- *Example 5:* This example names the expression SAL+BONUS+COMM as TOTAL_PAY

      SELECT SALARY+BONUS+COMM AS TOTAL_PAY
        FROM EMPLOYEE
        ORDER BY TOTAL_PAY

- *Example 6:* Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

  Using a common table expression for this case saves the processing resources of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives are considered by the view.

      WITH
          DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
              (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
                FROM EMPLOYEE OTHERS
                GROUP BY OTHERS.WORKDEPT
              ),
          DINFOMAX AS
              (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
       SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
              DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
        FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
        WHERE THIS_EMP.JOB = 'SALESREP'
        AND THIS_EMP.WORKDEPT = DINFO.DEPTNO

- *Example 7:* Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

Examples of select-statement queries

```
WITH
  NEWEMP AS (SELECT EMPNO FROM NEW TABLE
                 (INSERT INTO EMPLOYEE(EMPNO, FIRSTNME)
                     VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
  OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
  UPPROJ AS (SELECT PROJNAME FROM NEW TABLE
                 (UPDATE PROJECT
                     SET RESPEMP = (SELECT EMPNO FROM NEWEMP)
                     WHERE RESPEMP = (SELECT EMPNO FROM OLDEMP))),
  DELEMP AS (SELECT EMPNO FROM OLD TABLE
                 (DELETE FROM EMPLOYEE
                     WHERE EMPNO = (SELECT EMPNO FROM OLDEMP)))
  SELECT PROJNAME FROM UPPROJ;
```

- *Example 8:* Retrieve data from the DEPT table. That data will later be updated with a searched update, and will be locked when the query executes.

```
SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPT
  WHERE ADMRDEPT ='A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS
```

- *Example 9:* Select all columns and rows from the EMPLOYEE table. If another transaction is concurrently updating, deleting, or inserting data in the EMPLOYEE table, the select operation will wait to get the data until after the other transaction is completed.

```
SELECT * FROM EMPLOYEE WAIT FOR OUTCOME
```

Appendix A. SQL and XML limits

The tables in this topic describe SQL and XML limits. Adhering to the most restrictive case can help you to design application programs that are easily portable.

Table 85 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 85. 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[2]	128
Constraint name	128
Correlation name	128
Cursor name	128
Data partition name	128

SQL and XML limits

Table 85. 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 (*remote-table-name*)	128
Database partition group name	128
Database partition name	128
Event monitor name	128
External program name	128
Function mapping name	128
Group name	128
Host identifier[1]	255
Identifier for a data source user (*remote-authorization-name*)	128
Identifier in an SQL procedure (condition name, for loop identifier, label, result set locator, statement name, variable name)	128
Index name	128
Index extension name	18
Index specification name	128
Label name	128
Namespace uniform resource identifier (URI)	1000
Nickname	128
Package name	128
Package version ID	64
Parameter name	128
Password to access a data source	32
Procedure name	128
Role name	128
Savepoint name	128
Schema name[2]	128
Security label component name	128
Security label name	128
Security policy name	128
Sequence name	128
Server (database alias) name	8
Specific name	128
SQL condition name	128
SQL variable name	128
Statement name	128
Storage Group	128
Table name	128

Table 85. Identifier Length Limits (continued)

Description	Maximum in Bytes
Table space name	18
Transform group name	18
Trigger name	128
Trusted context name	128
Type mapping name	18
User-defined function name	128
User-defined method name	128
User-defined type name[2]	128
View name	128
Wrapper name	128
XML element name, attribute name, or prefix name	1000
XML schema location uniform resource identifier (URI)	1000
Note:	
1. Individual host language compilers might have a more restrictive limit on variable names.	
2. The SQLDA structure is limited to storing 30-byte column names, 18-byte user-defined type names, and 8-byte schema names for user-defined types. Because the SQLDA is used in the DESCRIBE statement, embedded SQL applications that use the DESCRIBE statement to retrieve column or user-defined type name information must conform to these limits.	

Table 86. 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

SQL and XML limits

Table 86. Numeric Limits (continued)

Description	Limit
Minimum exponent (E_{min}) for DOUBLE values	-307
Smallest positive DOUBLE value	+2.225E-307
Largest negative DOUBLE value	-2.225E-307
Maximum exponent (E_{max}) for DECFLOAT(16) values	384
Smallest DECFLOAT(16) value[1]	-9.999999999999999E+384
Largest DECFLOAT(16) value	9.999999999999999E+384
Minimum exponent (E_{min}) for DECFLOAT(16) values	-383
Smallest positive DECFLOAT(16) value	1.000000000000000E-383
Largest negative DECFLOAT(16) value	-1.000000000000000E-383
Maximum exponent (E_{max}) for DECFLOAT(34) values	6144
Smallest DECFLOAT(34) value[1]	-9.999999999999999999999999999999999E+6144
Largest DECFLOAT(34) value	9.999999999999999999999999999999999E+6144
Minimum exponent (E_{min}) for DECFLOAT(34) values	-6143
Smallest positive DECFLOAT(34) value	1.000000000000000000000000000000000E-6143
Largest negative DECFLOAT(34) value	-1.000000000000000000000000000000000E-6143

Note:

1. These are the limits of normal decimal floating-point numbers. Valid decimal floating-point values include the special values NAN, -NAN, SNAN, -SNAN, INFINITY and -INFINITY. In addition, valid values include subnormal numbers.

    Subnormal numbers are nonzero numbers whose adjusted exponents are less than E_{min}. For a subnormal number, the minimum value of the exponent is E_{min} - (*precision*-1), called E_{tiny}, where *precision* is the working precision (16 or 34). That is, subnormal numbers extend the range of numbers close to zero by 15 or 33 orders of magnitude for DECFLOAT(16) or DECFLOAT(34), respectively. Subnormal numbers are different from normal numbers because the maximum number of digits for a subnormal number is less than the working precision (16 or 34). Decimal floating-point cannot represent the subnormal numbers with the same accuracy as it can represent normal numbers. The smallest positive subnormal number for DECFLOAT(34) is 1×10^{-6176}, which contains only one digit, whereas the smallest positive normal number for DECFLOAT(34) is $1.000000000000000000000000000000000 \times 10^{-6143}$, which contains 34 digits. The smallest positive subnormal number for DECFLOAT(16) is 1×10^{-398}.

Table 87. String Limits

Description	Limit
Maximum length of CHAR (in bytes or OCTETS)	254

Table 87. String Limits (continued)

Description	Limit
Maximum length of CHAR (in CODEUNITS32)	63
Maximum length of VARCHAR (in bytes or OCTETS)[2]	32 672
Maximum length of VARCHAR (in CODEUNITS32)	8 168
Maximum length of LONG VARCHAR (in bytes)[1]	32 700
Maximum length of CLOB (in bytes or OCTETS))	2 147 483 647
Maximum length of CLOB (in CODEUNITS32)	536 870 911
Maximum length of serialized XML (in bytes)	2 147 483 647
Maximum length of GRAPHIC (in double-byte characters or CODEUNITS16)	127
Maximum length of GRAPHIC (in CODEUNITS32)	63
Maximum length of VARGRAPHIC (in double-byte characters or CODEUNITS16)[2]	16 336
Maximum length of VARGRAPHIC (in CODEUNITS32)	8 168
Maximum length of LONG VARGRAPHIC (in double-byte characters)[1]	16 350
Maximum length of DBCLOB (in double-byte characters or CODEUNITS16)	1 073 741 823
Maximum length of DBCLOB (in CODEUNITS32)	536 870 911
Maximum length of BLOB (in bytes)	2 147 483 647
Maximum length of character constant	32 672
Maximum length of graphic constant	16 336
Maximum length of concatenated character string	2 147 483 647
Maximum length of concatenated graphic string	1 073 741 823
Maximum length of concatenated binary string	2 147 483 647
Maximum number of hexadecimal constant digits	32 672
Largest instance of a structured type column object at run time (in gigabytes)	1
Maximum size of a catalog comment (in bytes)	254
Note:	
1. The LONG VARCHAR and LONG VARGRAPHIC data types are deprecated and might be removed in a future release.	
2. For page size-specific string limits for column-organized tables, see Table 88.	

Table 88. Page Size-specific String Limits for Column-organized Tables

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum length of VARCHAR (in bytes)	3920	8016	16 208	32 592
Maximum length of VARGRAPHIC (in bytes)	1960	4008	8104	16 296
Note: A column-organized table has an overhead of 176 bytes per page.				

SQL and XML limits

Table 89. 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 90. Datetime Limits

Description	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000000000
Smallest timestamp precision	0
Largest timestamp precision	12

Table 91. Database Manager Limits

Category	Description	Limit
Applications	Maximum number of host variable declarations in a precompiled program[3]	storage
	Maximum length of a host variable value (in bytes)	2 147 483 647
	Maximum number of declared cursors in a program	storage
	Maximum number of rows changed in a unit of work	storage
	Maximum number of cursors opened at one time	storage
	Maximum number of connections per process within a database client	512
	Maximum number of simultaneously opened LOB locators in a transaction	4 194 304
	Maximum size of an SQLDA (in bytes)	storage
	Maximum number of prepared statements	storage
Buffer Pools	Maximum NPAGES in a buffer pool for 32-bit releases	1 048 576
	Maximum NPAGES in a buffer pool for 64-bit releases	2 147 483 647
	Maximum total size of all buffer pool slots (4K)	2 147 483 646

Table 91. Database Manager Limits (continued)

Category	Description	Limit
Concurrency	Maximum number of concurrent users of a server[4]	64 000
	Maximum number of concurrent users per instance	64 000
	Maximum number of concurrent applications per database	60 000
	Maximum number of databases per instance concurrently in use	256
Constraints	Maximum number of constraints on a table	storage
	Maximum number of columns in a UNIQUE constraint (supported through a UNIQUE index)	64
	Maximum combined length of columns in a UNIQUE constraint (supported through a UNIQUE index, in bytes)[8]	8192
	Maximum number of referencing columns in a foreign key	64
	Maximum combined length of referencing columns in a foreign key (in bytes)[8]	8192
	Maximum length of a check constraint specification (in bytes)	65 535
Databases	Maximum database partition number	999
	Maximum members in a DB2 pureScale environment	128
Indexes	Maximum number of indexes on a table	32 767 or storage
	Maximum number of columns in an index key	64
	Maximum length of an index key including all overhead[6] [8]	*indexpagesize*/4
	Maximum length of a variable index key part (in bytes)[7]	1022 or storage
	Maximum size of an index per database partition in an SMS table space (in terabytes)[6]	64
	Maximum size of an index per database partition in a regular DMS table space (in gigabytes)[6]	512
	Maximum size of an index per database partition in a large DMS table space (in terabytes)[6]	64
	Maximum length of a variable index key part for an index over XML data (in bytes)[7]	*pagesize*/4 - 207
Log records	Maximum Log Sequence Number	0xFFFF FFFF FFFF FFFF

SQL and XML limits

Table 91. Database Manager Limits (continued)

Category	Description	Limit
Monitoring	Maximum number of simultaneously active event monitors	128
	In a partitioned database environment, maximum number of simultaneously active GLOBAL event monitors	32
Routines	Maximum number of parameters in a procedure with LANGUAGE SQL	32 767
	Maximum number of parameters in an external procedure with PROGRAM TYPE MAIN	32 767
	Maximum number of parameters in an external procedure with PROGRAM TYPE SUB	90
	Maximum number of parameters in a cursor value constructor	32 767
	Maximum number of parameters in a user-defined function	90
	Maximum number of nested levels for routines	64
	Maximum number of schemas in the SQL path	64
	Maximum length of the SQL path (in bytes)	2048
Security	Maximum number of elements in a security label component of type set or tree	64
	Maximum number of elements in a security label component of type array	65 535
	Maximum number of security label components in a security policy	16

Table 91. Database Manager Limits (continued)

Category	Description	Limit
SQL	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[6]	1012
	Maximum number of predicates in a WHERE or HAVING clause	storage
	Maximum number of columns in a GROUP BY clause[6]	1012
	Maximum total length of columns in a GROUP BY clause (in bytes)[6]	32 677
	Maximum number of columns in an ORDER BY clause[6]	1012
	Maximum total length of columns in an ORDER BY clause (in bytes)[6]	32 677
	Maximum level of subquery nesting	storage
	Maximum number of subqueries in a single statement	storage
	Maximum number of values in an insert operation[6]	1012
Storage Groups	Maximum number of storage groups in a database	256
	Maximum number of storage paths in a storage group	128
	Maximum length of a storage path (in bytes)	175

Table 91. Database Manager Limits (continued)

Category	Description	Limit
Tables and Views	Maximum number of columns in a table [6]	1012
	Maximum number of columns in a view[1]	5000
	Maximum number of columns in a data source table or view that is referenced by a nickname	5000
	Maximum number of columns in a distribution key[5]	500
	Maximum length of a row including all overhead[2][6][9]	32 677
	Maximum number of rows in a non-partitioned table, per database partition	128×10^{10}
	Maximum number of rows in a data partition, per database partition	128×10^{10}
	Maximum size of a table per database partition in a regular table space (in gigabytes)[3][6]	512
	Maximum size of a table per database partition in a large DMS table space (in terabytes)[6]	64
	Maximum number of data partitions for a single table	32 767
	Maximum number of table partitioning columns	16
	Maximum number of fields in a user-defined row data type	1012
Table Spaces	Maximum size of a LOB object per table or per table partition (in terabytes)	4
	Maximum size of an LF object per table or per table partition (in terabytes)	2
	Maximum number of table spaces in a database	32 768
	Maximum number of tables in an SMS table space	65 532
	Maximum size of a regular DMS table space (in gigabytes) [3][6]	512
	Maximum size of a large DMS table space (in terabytes) [3][6]	64
	Maximum size of a temporary DMS table space (in terabytes) [3][6]	64
	Maximum number of table objects in a DMS table space	See Table 92 on page 871
Triggers	Maximum runtime depth of cascading triggers	16
User-defined Types	Maximum number of attributes in a structured type	4082

Table 91. Database Manager Limits (continued)

Category	Description	Limit
Workload Manager	Maximum number of user-defined service superclasses per database	64
	Maximum number of user-defined service subclasses per service superclass	61
Note:		
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 `max_connections` and `max_coordagents` database manager configuration parameters.		
5. This is an architectural limit. The limit on the most columns in an index key should be used as the practical limit.		
6. For page size-specific values, see Table 92.		
7. This is limited only by the longest index key, including all overhead (in bytes). As the number of index key parts increases, the maximum length of each key part decreases.		
8. The maximum can be less, depending on index options.		
9. If the `extended_row_sz` database configuration parameter is set to ENABLE and there are VARCHAR or VARGRAPHIC columns in the table, the maximum row size is 1 048 319 bytes.		

Table 92. Database Manager Page Size-specific Limits

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum number of table objects in a DMS table space[1]	51 971[2] 53 212[3]	53 299	53 747	54 264
Maximum number of columns in a row-organized table	500	1012	1012	1012
Maximum number of columns in a column-organized table	1012	1012	1012	1012
Maximum length of a row in a row-organized table, including all overhead	4005	8101	16 293	32 677
Maximum length of a row in a column-organized table, including all overhead	32 677	32 677	32 677	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 terabytes)	8	16	32	64

SQL and XML limits

Table 92. Database Manager Page Size-specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Maximum length of an index key including all overhead (in bytes)	1024	2048	4096	8192
Maximum size of an index per database partition in an SMS table space (in terabytes)	8	16	32	64
Maximum size of an index per database partition in a regular DMS table space (in gigabytes)	64	128	256	512
Maximum size of an index per database partition in a large DMS table space (in terabytes)	8	16	32	64
Maximum size of a regular DMS table space (in gigabytes)	64	128	256	512
Maximum size of a large DMS table space (in terabytes)	8	16	32	64
Maximum size of a temporary DMS table space (in terabytes)	8	16	32	64
Maximum number of elements in a select list	500[4]	1012	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
Maximum records per page for a regular table space	251	253	254	253
Maximum records per page for a large table space	287	580	1165	2335

Table 92. Database Manager Page Size-specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Note:				
1. Table objects include table data, indexes, LONG VARCHAR columns, LONG VARGRAPHIC columns, and LOB columns. Table objects that are in the same table space as the table data do not count extra toward the limit. However, each table object that is in a different table space than the table data does contribute one toward the limit for each table object type per table in the table space in which the table object resides.				
2. When extent size is 2 pages.				
3. When extent size is any size other than 2 pages.				
4. In cases where the only system temporary table space is 4KB and the data overflows to the sort buffer, an error is generated. If the result set can fit into memory, there is no error.				

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 93. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlcaid	CHAR(8)	An "eye catcher" for storage dumps containing 'SQLCA'. The sixth byte is 'L' if line number information is returned from parsing an SQL routine, SQL trigger, or dynamic compound SQL statement. The sixth byte is 'M' if the line number and object ID information is returned from executing a compiled SQL routine, compiled SQL trigger, or dynamic compound SQL (compiled) statement.
sqlcabc	INTEGER	Contains the length of the SQLCA, 136.
sqlcode	INTEGER	Contains the SQL return code.
		Code **Means**
		0 Successful execution (although one or more SQLWARN indicators may be set).
		positive Successful execution, but with a warning condition.
		negative Error condition.
sqlerrml	SMALLINT	Length indicator for *sqlerrmc*, in the range 0 through 70. 0 means that the value of *sqlerrmc* is not relevant.

SQLCA (SQL communications area)

Table 93. 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)	Contains one or more tokens, separated by X'FF', which are substituted for variables in the descriptions of error conditions. This field is also used when a successful connection is completed. When a NOT ATOMIC compound SQL statement is issued, it can contain information about seven or fewer errors. The last token might be followed by X'FF'. The *sqlerrml* value will include any trailing X'FF'.
sqlerrp	CHAR(8)	Begins with a three-letter identifier indicating the product, followed by five alphanumeric characters indicating the version, release, and modification level of the product. The characters A-Z indicate a modification level higher than 9. "A" indicates modification level 10, "B" indicates modification level 11, and so on. For example, SQL0907C means DB2 Version 9, release 7, modification level 12). If SQLCODE indicates an error condition, this field identifies the module that returned the error. This field is also used when a successful connection is completed.
sqlerrd	ARRAY	Six INTEGER variables that provide diagnostic information. These values are generally empty if there are no errors, except for sqlerrd(6) from a partitioned database.
sqlerrd(1)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the database code page from the application code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. On successful return from an SQL procedure, contains the return status value from the SQL procedure.
sqlerrd(2)	INTEGER	If connection is invoked and successful, contains the maximum expected difference in length of mixed character data (CHAR data types) when converted to the application code page from the database code page. A value of 0 or 1 indicates no expansion; a value greater than 1 indicates a possible expansion in length; a negative value indicates a possible contraction. If the SQLCA results from a NOT ATOMIC compound SQL statement that encountered one or more errors, the value is set to the number of statements that failed.

SQLCA (SQL communications area)

Table 93. 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	If PREPARE is invoked and successful, contains an estimate of the number of rows that will be returned. After INSERT, UPDATE, DELETE, or MERGE, contains the actual number of rows that qualified for the operation. For a TRUNCATE statement, the value will be -1. If compound SQL is invoked, contains an accumulation of all sub-statement rows. If CONNECT is invoked, contains 1 if the database can be updated, or 2 if the database is read only. If the OPEN statement is invoked, and the cursor contains SQL data change statements, this field contains the sum of the number of rows that qualified for the embedded insert, update, delete, or merge operations. If an error is encountered during the compilation of an SQL routine, trigger, or dynamic compound SQL (inlined or compiled) statement, sqlerrd(3) contains the line number where the error was encountered. The sixth byte of sqlcaid must be 'L' for this entry to be a valid line number. If an error is encountered during the execution of a compiled SQL routine, trigger, or dynamic SQL (compiled) statement, sqlerrd(3) contains the line number where the error was raised. The sixth byte of sqlcaid must be 'M' for this to be a valid line number.
sqlerrd(4)	INTEGER	If PREPARE is invoked and successful, contains a relative cost estimate of the resources required to process the statement. If compound SQL is invoked, contains a count of the number of successful sub-statements. If CONNECT is invoked, contains 0 for a one-phase commit from a client which is not at the latest level; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit. If an error is encountered during the execution of a compiled SQL routine or trigger, sqlerrd(4) contains an integer number that uniquely identifies the routine or trigger within which the error was raised. The sixth byte of sqlcaid must be 'M' for this entry to be a valid line number

Appendix B. SQLCA (SQL communications area)   **877**

SQLCA (SQL communications area)

Table 93. 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(5)	INTEGER	Contains the total number of rows deleted, inserted, or updated as a result of both: • The enforcement of constraints after a successful delete operation • The processing of triggered SQL statements from activated inlined triggers If compound SQL is invoked, contains an accumulation of the number of such rows for all sub-statements. In some cases, when an error is encountered, this field contains a negative value that is an internal error pointer. If CONNECT is invoked, contains an authentication type value: • 0 for server authentication • 1 for client authentication • 2 for authentication using DB2 Connect • 4 for SERVER_ENCRYPT authentication • 5 for authentication using DB2 Connect with encryption; • 7 for KERBEROS authentication • 9 for GSSPLUGIN authentication • 11 for DATA_ENCRYPT authentication • 255 for unspecified authentication.
sqlerrd(6)	INTEGER	For a partitioned database, contains the partition number of the database partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator partition. The number in this field is the same as that specified for the database partition in the db2nodes.cfg file.
sqlwarn	Array	A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all sub-statements.
sqlwarn0	CHAR(1)	Blank if all other indicators are blank; contains 'W' if at least one other indicator is not blank.
sqlwarn1	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'N' if the null terminator was truncated. Contains 'A' if the CONNECT or ATTACH is successful, and the authorization name for the connection is longer than 8 bytes. Contains 'P' if the PREPARE statement relative cost estimate stored in sqlerrd(4) exceeded the value that could be stored in an INTEGER or was less than 1, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to a value other than NO.
sqlwarn2	CHAR(1)	Contains 'W' if null values were eliminated from the argument of an aggregate function. [a] If CONNECT is invoked and successful, contains 'D' if the database is in quiesce state, or 'I' if the instance is in quiesce state.

Table 93. 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
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.
sqlwarn8	CHAR(1)	Contains 'W' if a character that could not be converted was replaced with a substitution character. Contains 'Y' if there was an unsuccessful attempt to establish a trusted connection.
sqlwarn9	CHAR(1)	Contains 'W' if arithmetic expressions with errors were ignored during aggregate function processing.
sqlwarn10	CHAR(1)	Contains 'W' if there was a conversion error when converting a character data value in one of the fields in the SQLCA.
sqlstate	CHAR(5)	A return code that indicates the outcome of the most recently executed SQL statement.

[a] Some functions may not set SQLWARN2 to W, even though null values were eliminated, because the result was not dependent on the elimination of null values.

Error reporting

The order of error reporting is 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.

SQLCA (SQL communications area)

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. Array, cursor and row types are not supported as SQLDA variables in EXECUTE, FETCH and OPEN statements.)

SQLDA field descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table or a parameter marker. There are two types of SQLVAR entries:
- **Base SQLVARs:** These entries are always present. They contain the base information about the column, parameter marker, or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
- **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined previously. For user-defined types (excluding reference types), they contain the user-defined type name. For reference types, they contain the target type of the reference. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. (The distinct type and LOB information does not

881

overlap, so distinct types can be based on LOBs without forcing the number of SQLVAR entries on a DESCRIBE to be tripled.) If locators or file reference variables are used to represent LOBs, these entries are not necessary.

In SQLDAs that contain both types of entries, the base SQLVARs are in a block before the block of secondary SQLVARs. In each, the number of entries is equal to the value in SQLD (even though many of the secondary SQLVAR entries may be unused).

The circumstances under which the SQLVAR entries are set by DESCRIBE is detailed in "Effect of DESCRIBE on the SQLDA" on page 886.

Fields in the SQLDA header

Table 94. Fields in the SQLDA Header

C Name	SQL Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (set by the application before executing the statement)
sqldaid	CHAR(8)	The seventh byte of this field is a flag byte named SQLDOUBLED. The database manager sets SQLDOUBLED to the character '2' if two SQLVAR entries have been created for each column; otherwise it is set to a blank (X'20' in ASCII, X'40' in EBCDIC). See "Effect of DESCRIBE on the SQLDA" on page 886 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 95. Fields in a Base SQLVAR

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqltype	SMALLINT	Indicates the data type of the column or parameter marker, and whether it can contain nulls. (Parameter markers are always considered nullable.) Table 97 on page 887 lists the allowable values and their meanings. Note that for a distinct, array, cursor, row, or reference type, the data type of the base type is placed into this field. For a structured type, the data type of the result of the FROM SQL transform function of the transform group (based on the CURRENT DEFAULT TRANSFORM GROUP special register) for the type is placed into this field. There is no indication in the base SQLVAR that it is part of the description of a user-defined type or reference type.	Same for host variable. Host variables for datetime values must be character string variables. For FETCH, a datetime type code means a fixed-length character string. If sqltype is an even number value, the sqlind field is ignored.
sqllen	SMALLINT	The length attribute of the column or parameter marker. For datetime columns and parameter markers, the length of the string representation of the values. See Table 97 on page 887. Note that the value is set to 0 for large object strings (even for those whose length attribute is small enough to fit into a two byte integer).	The length attribute of the host variable. See Table 97 on page 887. Note that the value is ignored by the database manager for CLOB, DBCLOB, and BLOB columns. The len.sqllonglen field in the Secondary SQLVAR is used instead.
sqldata	pointer	For string SQLVARS, sqldata contains the code page. For character-string SQLVARs where the column is defined with the FOR BIT DATA attribute, sqldata contains 0. For other character-string SQLVARS, sqldata contains either the SBCS code page for SBCS data, or the SBCS code page associated with the composite MBCS code page for MBCS data. For Japanese EUC, Traditional Chinese EUC, and Unicode UTF-8 character-string SQLVARS, sqldata contains 954, 964, and 1208 respectively. For all other column types, sqldata is undefined.	Contains the address of the host variable (where the fetched data will be stored).
sqlind	pointer	For character-string SQLVARS, sqlind contains 0, except for MBCS data, when sqlind contains the DBCS code page associated with the composite MBCS code page. For all other types, sqlind is undefined.	Contains the address of an associated indicator variable, if there is one; otherwise, not used. If sqltype is an even number value, the sqlind field is ignored.

SQLDA (SQL descriptor area)

Table 95. Fields in a Base SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqlname	VARCHAR (30)	Contains the unqualified name of the column or parameter marker. For columns and parameter markers that have a system-generated name, the thirtieth byte is set to X'FF'. For column names specified by the AS clause, this byte is X'00'.	When connecting to a host database, sqlname can be set to indicate a FOR BIT DATA string as follows: • The sixth byte of the SQLDAID in the SQLDA header is set to '+' • The length of sqlname is 8 • The first two bytes of sqlname are X'0000' • The third and fourth bytes of sqlname are X'0000' • The remaining four bytes of sqlname are reserved and should be set to X'00000000' When working with XML data, sqlname can be set to indicate an XML subtype as follows: • The length of sqlname is 8 • The first two bytes of sqlname are X'0000' • The third and fourth bytes of sqlname are X'0000' • The fifth byte of sqlname is X'01' • The remaining three bytes of sqlname are reserved and should be set to X'000000'

Fields in an occurrence of a secondary SQLVAR

Table 96. 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 96. 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 the null value, then the actual length (in double-byte characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length. If this field is not the null value, it contains a pointer to a 4 byte long buffer that contains the actual length *in bytes* (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Note that, whether or not this field is used, the len.sqllonglen field must be set.
sqldatatype_name	VARCHAR(27)	For a user-defined type, the database manager sets this to the fully qualified user-defined type name.[1] For a reference type, the database manager sets this to the fully qualified type name of the target type of the reference.	For structured types, set to the fully qualified user-defined type name in the format indicated in the table note.[1]
reserved	CHAR(3)	Not used.	Not used.

[1] The first 8 bytes contain the schema name of the type (extended to the right with spaces, if necessary). Byte 9 contains a dot (.). Bytes 10 to 27 contain the low order portion of the type name, which is *not* extended to the right with spaces.

Note that, although the prime purpose of this field is for the name of user-defined types, the field is also set for IBM predefined data types. In this case, the schema name is SYSIBM, and the low order portion of the name is the name stored in the TYPENAME column of the DATATYPES catalog view. For example:

```
type name       length   sqldatatype_name
---------       ------   ----------------
A.B             10       A       .B
INTEGER         16       SYSIBM  .INTEGER
"Frank's".SMINT 13       Frank's .SMINT
MY."type  "     15       MY      .type
```

Effect of DESCRIBE on the SQLDA

For a DESCRIBE OUTPUT or PREPARE OUTPUT INTO statement, the database manager always sets SQLD to the number of columns in the result set, or the number of output parameter markers. For a DESCRIBE INPUT or PREPARE INPUT INTO statement, the database manager always sets SQLD to the number of input parameter markers in the statement. Note that a parameter marker that corresponds to an INOUT parameter in a CALL statement is described in both the input and output descriptors.

The SQLVARs in the SQLDA are set in the following cases:
- SQLN >= SQLD and no entry is either a LOB, user-defined type or reference type

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank.
- SQLN >= 2*SQLD and at least one entry is a LOB, user-defined type or reference type

  Two times SQLD SQLVAR entries are set, and SQLDOUBLED is set to '2'.
- SQLD <= SQLN < 2*SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:
- SQLN < SQLD and no entry is either a LOB, user-defined type or reference type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVARs for a successful DESCRIBE.
- SQLN < SQLD and at least one entry is a distinct, array, cursor, row, or reference type, but there are no LOB entries or structured type entries

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct, array, cursor, and row types and target types of reference types.
- SQLN < 2*SQLD and at least one entry is a LOB or a structured type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

References in the previous lists to LOB entries include distinct type entries whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB or structured type entries in the select list and there

are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB or structured type entry in the result set.

If a structured type entry is being described, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 42741) or because the name group does not have a FROM SQL transform function defined (SQLSTATE 42744)), the DESCRIBE will return an error. This error is the same error returned for a DESCRIBE of a table with a structured type entry.

If the database manager returns identifiers that are longer than those that can be stored in the SQLDA, the identifier is truncated and a warning is returned (SQLSTATE 01665); however, when the name of a structured type is truncated, an error is returned (SQLSTATE 42622). For details on identifier length limitations, see "SQL and XQuery limits" .

SQLTYPE and SQLLEN

Table 97 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 97. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE

SQLTYPE	Column data type for DESCRIBE and PREPARE INTO	SQLLEN for DESCRIBE and PREPARE INTO	Host variable data type for FETCH, OPEN, and EXECUTE	SQLLEN for FETCH, OPEN, and EXECUTE
384/385	date	10	fixed-length character string representation of a date	length attribute of the host variable
388/389	time	8	fixed-length character string representation of a time	length attribute of the host variable
392/393	timestamp	19 for TIMESTAMP(0) otherwise 20+p for TIMESTAMP(p)	fixed-length character string representation of a timestamp	length attribute of the host variable
400/401	N/A	N/A	NULL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 *	BLOB	Not used. *
408/409	CLOB	0 *	CLOB	Not used. *
412/413	DBCLOB	0 *	DBCLOB	Not used. *
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable
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

SQLDA (SQL descriptor area)

Table 97. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE (continued)

SQLTYPE	Column data type for DESCRIBE and PREPARE INTO	SQLLEN for DESCRIBE and PREPARE INTO	Host variable data type for FETCH, OPEN, and EXECUTE	SQLLEN for FETCH, OPEN, and EXECUTE
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable
472/473	long varying-length graphic string	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating-point	8 for double precision, 4 for single precision	floating-point	8 for double precision, 4 for single precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
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)
2440/2441	row	not applicable	row	not used
2440/2441	cursor	not applicable	row	not used

Note:

- The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.

- The SQLTYPE has changed from previous versions for portability reasons. The values from the previous version (see previous version SQL Reference) continue to be supported.

Unrecognized and unsupported SQLTYPEs

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data.

Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.

When the sender and receiver agree to use a compatible data type, the following table indicates the mapping that will take place. This mapping will take place when at least one of the sender or the receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Data Type	Compatible Data Type
BIGINT	DECIMAL(19, 0)
ROWID	VARCHAR(40) FOR BIT DATA

Note that no indication is given in the SQLDA that the data type is substituted.

Packed decimal numbers

Packed decimal numbers are stored in a variation of Binary Coded Decimal (BCD) notation. In BCD, each nybble (four bits) represents one decimal digit. For example, 0001 0111 1001 represents 179. Therefore, read a packed decimal value nybble by nybble. Store the value in bytes and then read those bytes in hexadecimal representation to return to decimal. For example, 0001 0111 1001 becomes 00000001 01111001 in binary representation. By reading this number as hexadecimal, it becomes 0179.

The decimal point is determined by the scale. In the case of a DEC(12,5) column, for example, the rightmost 5 digits are to the right of the decimal point.

Sign is indicated by a nybble to the right of the nybbles representing the digits. A positive or negative sign is indicated as follows:

Table 98. Values for Sign Indicator of a Packed Decimal Number

Sign	Binary representation	Decimal representation	Hexadecimal representation
Positive (+)	1100	12	C
Negative (-)	1101	13	D

In summary:
- To store any value, allocate $p/2+1$ bytes, where p is precision.
- Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
- The sign nybble will be the second nybble of the last byte.

For example:

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(8,3)	6574.23	00 65 74 23 0C
DEC(6,2)	-334.02	00 33 40 2D
DEC(7,5)	5.2323	05 23 23 0C

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(5,2)	-23.5	02 35 0D

SQLLEN field for decimal

The SQLLEN field contains the precision (first byte) and scale (second byte) of the decimal column. If writing a portable application, the precision and scale bytes should be set individually, versus setting them together as a short integer. This will avoid integer byte reversal problems.

For example, in C:

```
((char *)&(sqlda->sqlvar[i].sqllen))[0] = precision;
((char *)&(sqlda->sqlvar[i].sqllen))[1] = scale;
```

Appendix D. Catalog views

The database manager creates and maintains two sets of catalog views that are defined on top of the base system catalog tables.

- SYSCAT views are read-only catalog views that are found in the SYSCAT schema. The RESTRICT option on CREATE DATABASE statement determines how SELECT privilege is granted. When the RESTRICT option is not specified, SELECT privilege is granted to PUBLIC.

- SYSSTAT views are updatable catalog views that are found in the SYSSTAT schema. The updatable views contain statistical information that is used by the optimizer. The values in some columns in these views can be changed to test performance. (Before changing any statistics, it is recommended that the RUNSTATS command be invoked so that all the statistics reflect the current state.)

Applications should be written to the SYSCAT and SYSSTAT views rather than the base catalog tables.

All the catalog views are created at database creation time. The catalog views cannot be explicitly created or dropped. In a Unicode database, the catalog views are created with IDENTITY collation. In non-Unicode databases, the catalog views are created with the database collation. The views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the catalog views is available through normal SQL query facilities. The catalog views (with the exception of some updatable catalog views) cannot be modified using normal SQL data manipulation statements.

An object table, statistical view, column, or index object appears in a user's updatable SYSSTAT catalog view only if that user holds explicit CONTROL privilege on the object, or holds explicit DATAACCESS authority. An object table or statistical view also appears in a user's updatable SYSSTAT.TABLES catalog view if the user is a direct or indirect member of a role that has CONTROL privilege on the object, or a role that has DATAACCESS authority. Role privileges and authorities are not considered when determining the objects that appear in the other SYSSTAT catalog views. A routine object appears in a user's updatable SYSSTAT.ROUTINES catalog view if that user owns the routine or holds explicit SQLADM authority. Group privileges and authorities are not considered when determining the objects that appear in a user's updatable SYSSTAT catalog views.

The order of columns in the views may change from release to release. To prevent this from affecting programming logic, specify the columns in a select list explicitly, and avoid using SELECT *. Columns have consistent names based on the types of objects that they describe.

Table 99. Samples of consistent column names for objects they describe

Described Object	Column Names
Table	TABSCHEMA, TABNAME
Index	INDSCHEMA, INDNAME
Index extension	IESCHEMA, IENAME
View	VIEWSCHEMA, VIEWNAME

Catalog views

Table 99. Samples of consistent column names for objects they describe (continued)

Described Object	Column Names
Constraint	CONSTSCHEMA, CONSTNAME
Control	CONTROLSCHEMA, CONTROLNAME, CONTROLID
Trigger	TRIGSCHEMA, TRIGNAME
Package	PKGSCHEMA, PKGNAME
Type	TYPESCHEMA, TYPENAME, TYPEID
Function	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Method	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Procedure	ROUTINESCHEMA, ROUTINEMODULENAME, ROUTINENAME, ROUTINEID
Column	COLNAME
Schema	SCHEMANAME
Table Space	TBSPACE
Database partition group	DBPGNAME
Audit policy	AUDITPOLICYNAME, AUDITPOLICYID
Buffer pool	BPNAME
Event Monitor	EVMONNAME
Condition	CONDSCHEMA, CONDMODULENAME, CONDNAME, CONDMODULEID
Data source	SERVERNAME, SERVERTYPE, SERVERVERSION
Global variable	VARSCHEMA, VARMODULENAME, VARNAME, VARMODULEID
Histogram template	TEMPLATENAME, TEMPLATEID
Module	MODULESCHEMA, MODULENAME, MODULEID
Period	PERIODNAME
Role	ROLENAME, ROLEID
Security label	SECLABELNAME, SECLABELID
Security policy	SECPOLICYNAME, SECPOLICYID
Sequence	SEQSCHEMA, SEQNAME
Threshold	THRESHOLDNAME, THRESHOLDID
Trusted context	CONTEXTNAME, CONTEXTID
Usage list	USAGELISTSCHEMA, USAGELISTNAME, USAGELISTID
Work action	ACTIONNAME, ACTIONID
Work action set	ACTIONSETNAME, ACTIONSETID
Work class	WORKCLASSNAME, WORKCLASSID
Work class set	WORKCLASSSETNAME, WORKCLASSSETID
Workload	WORKLOADID, WORKLOADNAME
Wrapper	WRAPNAME

Table 99. Samples of consistent column names for objects they describe (continued)

Described Object	Column Names
Alteration Timestamp	ALTER_TIME
Creation Timestamp	CREATE_TIME

Road map to the catalog views

This topic lists the catalog views, grouped by object or functionality.

Table 100. Road map to the read-only catalog views

Description	Catalog View
attributes of structured data types	"SYSCAT.ATTRIBUTES" on page 899
audit policies	"SYSCAT.AUDITPOLICIES" on page 901 "SYSCAT.AUDITUSE" on page 903
authorities on database	"SYSCAT.DBAUTH" on page 942
buffer pool configuration on database partition group	"SYSCAT.BUFFERPOOLS" on page 906
buffer pool size exceptions for database partitions	"SYSCAT.BUFFERPOOLDBPARTITIONS" on page 904
buffer pool size exceptions for members	"SYSCAT.BUFFERPOOLEXCEPTIONS" on page 905
cast functions	"SYSCAT.CASTFUNCTIONS" on page 907
check constraints	"SYSCAT.CHECKS" on page 908
column masks	"SYSCAT.CONTROLS" on page 931
column mask dependences	"SYSCAT.CONTROLDEP" on page 930
column privileges	"SYSCAT.COLAUTH" on page 910
columns	"SYSCAT.COLUMNS" on page 919
columns referenced by check constraints	"SYSCAT.COLCHECKS" on page 911
columns used in dimensions	"SYSCAT.COLUSE" on page 925
columns used in keys	"SYSCAT.KEYCOLUSE" on page 982
conditions	"SYSCAT.CONDITIONS" on page 926
constraint dependencies	"SYSCAT.CONSTDEP" on page 927
controls	"SYSCAT.CONTROLS" on page 931
database partition group database partitions	"SYSCAT.DBPARTITIONGROUPDEF" on page 944
database partition group definitions	"SYSCAT.DBPARTITIONGROUPS" on page 945
data partitions	"SYSCAT.DATAPARTITIONEXPRESSION" on page 933 "SYSCAT.DATAPARTITIONS" on page 934
data type dependencies	"SYSCAT.DATATYPEDEP" on page 937
data types	"SYSCAT.DATATYPES" on page 938
detailed column group statistics	"SYSCAT.COLGROUPCOLS" on page 913 "SYSCAT.COLGROUPDIST" on page 914 "SYSCAT.COLGROUPDISTCOUNTS" on page 915 "SYSCAT.COLGROUPS" on page 916
detailed column options	"SYSCAT.COLOPTIONS" on page 918
detailed column statistics	"SYSCAT.COLDIST" on page 912
distribution maps	"SYSCAT.PARTITIONMAPS" on page 1004
event monitor definitions	"SYSCAT.EVENTMONITORS" on page 946
events currently monitored	"SYSCAT.EVENTS" on page 948 "SYSCAT.EVENTTABLES" on page 949
fields of row data types	"SYSCAT.ROWFIELDS" on page 1031
function dependencies[1]	"SYSCAT.ROUTINEDEP" on page 1012

Table 100. Road map to the read-only catalog views (continued)

Description	Catalog View
function mapping	"SYSCAT.FUNCMAPPINGS" on page 954
function mapping options	"SYSCAT.FUNCMAPOPTIONS" on page 952
function parameter mapping options	"SYSCAT.FUNCMAPPARMOPTIONS" on page 953
function parameters[1]	"SYSCAT.ROUTINEPARMS" on page 1016
functions[1]	"SYSCAT.ROUTINES" on page 1019
global variables	"SYSCAT.VARIABLEAUTH" on page 1085 "SYSCAT.VARIABLEDEP" on page 1086 "SYSCAT.VARIABLES" on page 1088
hierarchies (types, tables, views)	"SYSCAT.HIERARCHIES" on page 955 "SYSCAT.FULLHIERARCHIES" on page 951
identity columns	"SYSCAT.COLIDENTATTRIBUTES" on page 917
index columns	"SYSCAT.INDEXCOLUSE" on page 960
index data partitions	"SYSCAT.INDEXPARTITIONS" on page 977
index dependencies	"SYSCAT.INDEXDEP" on page 961
index exploitation	"SYSCAT.INDEXEXPLOITRULES" on page 970
index extension dependencies	"SYSCAT.INDEXEXTENSIONDEP" on page 971
index extension parameters	"SYSCAT.INDEXEXTENSIONPARMS" on page 974
index extension search methods	"SYSCAT.INDEXEXTENSIONMETHODS" on page 973
index extensions	"SYSCAT.INDEXEXTENSIONS" on page 975
index options	"SYSCAT.INDEXOPTIONS" on page 976
index privileges	"SYSCAT.INDEXAUTH" on page 959
indexes	"SYSCAT.INDEXES" on page 963
invalid objects	"SYSCAT.INVALIDOBJECTS" on page 981
member susbsets	"SYSCAT.MEMBERSUBSETATTRS" on page 983 "SYSCAT.MEMBERSUBSETMEMBERS" on page 984 "SYSCAT.MEMBERSUBSETS" on page 985
method dependencies[1]	"SYSCAT.ROUTINEDEP" on page 1012
method parameters[1]	"SYSCAT.ROUTINES" on page 1019
methods[1]	"SYSCAT.ROUTINES" on page 1019
module objects	"SYSCAT.MODULEOBJECTS" on page 987
module privileges	"SYSCAT.MODULEAUTH" on page 986
modules	"SYSCAT.MODULES" on page 988
nicknames	"SYSCAT.NICKNAMES" on page 990
object mapping	"SYSCAT.NAMEMAPPINGS" on page 989
package dependencies	"SYSCAT.PACKAGEDEP" on page 994
package privileges	"SYSCAT.PACKAGEAUTH" on page 993
packages	"SYSCAT.PACKAGES" on page 996
partitioned tables	"SYSCAT.TABDETACHEDDEP" on page 1058
pass-through privileges	"SYSCAT.PASSTHRUAUTH" on page 1005
periods	"SYSCAT.PERIODS" on page 1006
predicate specifications	"SYSCAT.PREDICATESPECS" on page 1007

Road map to the catalog views

Table 100. Road map to the read-only catalog views (continued)

Description	Catalog View
procedure options	"SYSCAT.ROUTINEOPTIONS" on page 1014
procedure parameter options	"SYSCAT.ROUTINEPARMOPTIONS" on page 1015
procedure parameters[1]	"SYSCAT.ROUTINEPARMS" on page 1016
procedures[1]	"SYSCAT.ROUTINES" on page 1019
protected tables	"SYSCAT.SECURITYLABELACCESS" on page 1035 "SYSCAT.SECURITYLABELCOMPONENTELEMENTS" on page 1036 "SYSCAT.SECURITYLABELCOMPONENTS" on page 1037 "SYSCAT.SECURITYLABELS" on page 1038 "SYSCAT.SECURITYPOLICIES" on page 1039 "SYSCAT.SECURITYPOLICYCOMPONENTRULES" on page 1040 "SYSCAT.SECURITYPOLICYEXEMPTIONS" on page 1041 "SYSCAT.SURROGATEAUTHIDS" on page 1052
provides DB2 for z/OS compatibility	"SYSIBM.SYSDUMMY1" on page 1115
referential constraints	"SYSCAT.REFERENCES" on page 1008
remote table options	"SYSCAT.TABOPTIONS" on page 1069
roles	"SYSCAT.ROLEAUTH" on page 1009 "SYSCAT.ROLES" on page 1010
routine dependencies	"SYSCAT.ROUTINEDEP" on page 1012
routine parameters[1]	"SYSCAT.ROUTINEPARMS" on page 1016
routine privileges	"SYSCAT.ROUTINEAUTH" on page 1011
routines[1]	"SYSCAT.ROUTINES" on page 1019 "SYSCAT.ROUTINESFEDERATED" on page 1029
row permisssions	"SYSCAT.CONTROLS" on page 931
row permission dependeencies	"SYSCAT.CONTROLDEP" on page 930
schema privileges	"SYSCAT.SCHEMAAUTH" on page 1032
schemas	"SYSCAT.SCHEMATA" on page 1033
sequence privileges	"SYSCAT.SEQUENCEAUTH" on page 1042
sequences	"SYSCAT.SEQUENCES" on page 1043
server options	"SYSCAT.SERVEROPTIONS" on page 1045
server-specific user options	"SYSCAT.USEROPTIONS" on page 1084
statements	"SYSCAT.STATEMENTS" on page 1049 "SYSCAT.STATEMENTTEXTS" on page 1051
storage groups	"SYSCAT.STOGROUPS" on page 1050
procedures	"SYSCAT.ROUTINES" on page 1019
system servers	"SYSCAT.SERVERS" on page 1046
table constraints	"SYSCAT.TABCONST" on page 1055
table dependencies	"SYSCAT.TABDEP" on page 1056
table privileges	"SYSCAT.TABAUTH" on page 1053
table space use privileges	"SYSCAT.TBSPACEAUTH" on page 1070
table spaces	"SYSCAT.TABLESPACES" on page 1067
tables	"SYSCAT.TABLES" on page 1059
transforms	"SYSCAT.TRANSFORMS" on page 1074
trigger dependencies	"SYSCAT.TRIGDEP" on page 1075

en

Table 100. Road map to the read-only catalog views (continued)

Description	Catalog View
triggers	"SYSCAT.TRIGGERS" on page 1077
trusted contexts	"SYSCAT.CONTEXTATTRIBUTES" on page 928 "SYSCAT.CONTEXTS" on page 929
type mapping	"SYSCAT.TYPEMAPPINGS" on page 1080
usage lists	"SYSCAT.USAGELISTS" on page 1083
user-defined functions	"SYSCAT.ROUTINES" on page 1019
view dependencies	"SYSCAT.TABDEP" on page 1056
views	"SYSCAT.TABLES" on page 1059 "SYSCAT.VIEWS" on page 1090
workload management	"SYSCAT.HISTOGRAMTEMPLATEBINS" on page 956 "SYSCAT.HISTOGRAMTEMPLATES" on page 957 "SYSCAT.HISTOGRAMTEMPLATEUSE" on page 958 "SYSCAT.SCPREFTBSPACES" on page 1034 "SYSCAT.SERVICECLASSES" on page 1047 "SYSCAT.THRESHOLDS" on page 1071 "SYSCAT.WORKACTIONS" on page 1091 "SYSCAT.WORKACTIONSETS" on page 1094 "SYSCAT.WORKCLASSATTRIBUTES" on page 1095 "SYSCAT.WORKCLASSES" on page 1096 "SYSCAT.WORKCLASSSETS" on page 1097 "SYSCAT.WORKLOADAUTH" on page 1098 "SYSCAT.WORKLOADCONNATTR" on page 1099 "SYSCAT.WORKLOADS" on page 1100
wrapper options	"SYSCAT.WRAPOPTIONS" on page 1103
wrappers	"SYSCAT.WRAPPERS" on page 1104
XML strings	"SYSCAT.XMLSTRINGS" on page 1107
Index on XML column	"SYSCAT.INDEXXMLPATTERNS" on page 980
XSR objects	"SYSCAT.XDBMAPGRAPHS" on page 1105 "SYSCAT.XDBMAPSHREDTREES" on page 1106 "SYSCAT.XSROBJECTAUTH" on page 1108 "SYSCAT.XSROBJECTCOMPONENTS" on page 1109 "SYSCAT.XSROBJECTDEP" on page 1110 "SYSCAT.XSROBJECTDETAILS" on page 1112 "SYSCAT.XSROBJECTHIERARCHIES" on page 1113 "SYSCAT.XSROBJECTS" on page 1114

[1] The following catalog views for functions, methods, and procedures defined in DB2 Version 7.1 and earlier are still available:

```
Functions:  SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Methods:    SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Procedures: SYSCAT.PROCEDURES, SYSCAT.PROCPARMS
```

However, these views have not been updated since DB2 Version 7.1. Use the SYSCAT.ROUTINES, SYSCAT.ROUTINEDEP, or SYSCAT.ROUTINEPARMS catalog view instead.

Table 101. Road map to the updatable catalog views

Description	Catalog View
columns	"SYSSTAT.COLUMNS" on page 1120

Road map to the catalog views

Table 101. Road map to the updatable catalog views (continued)

Description	Catalog View
detailed column group statistics	"SYSSTAT.COLGROUPDIST" on page 1117 "SYSSTAT.COLGROUPDISTCOUNTS" on page 1118 "SYSSTAT.COLGROUPS" on page 1119
detailed column statistics	"SYSSTAT.COLDIST" on page 1116
indexes	"SYSSTAT.INDEXES" on page 1122
routines[1]	"SYSSTAT.ROUTINES" on page 1126
tables	"SYSSTAT.TABLES" on page 1127

[1] The SYSSTAT.FUNCTIONS catalog view still exists for updating the statistics for functions and methods. This view, however, does not reflect any changes since DB2 Version 7.1.

SYSCAT.ATTRIBUTES

Each row represents an attribute that is defined for a user-defined structured data type. Includes inherited attributes of subtypes.

Table 102. SYSCAT.ATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the structured data type that includes the attribute.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the structured data type belongs. The null value if not a module structured data type.
TYPENAME	VARCHAR (128)		Unqualified name of the structured data type that includes the attribute.
ATTR_NAME	VARCHAR (128)		Attribute name.
ATTR_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of an attribute.
ATTR_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of an attribute belongs. The null value if not a module attribute.
ATTR_TYPENAME	VARCHAR (128)		Unqualified name of the data type of an attribute.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target row type. Applies to reference types only; null value otherwise.
TARGET_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target row type belongs. The null value if not a module row type. Applies to reference types only; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type. Applies to reference types only; null value otherwise.
SOURCE_TYPESCHEMA	VARCHAR (128)		For inherited attributes, the schema name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPESCHEMA.
SOURCE_TYPEMODULENAME	VARCHAR (128)	Y	For inherited attributes, the unqualified name of the module to which the data type with which the attribute was first defined belongs. For non-inherited attributes, this column is the same as TYPEMODULEID. The null value if not a module data type.
SOURCE_TYPENAME	VARCHAR (128)		For inherited attributes, the unqualified name of the data type with which the attribute was first defined. For non-inherited attributes, this column is the same as TYPENAME.
ORDINAL	SMALLINT		Position of the attribute in the definition of the structured data type, starting with 0.
LENGTH	INTEGER		Length of the attribute data type. 0 if the attribute is a user-defined type.

SYSCAT.ATTRIBUTES

Table 102. SYSCAT.ATTRIBUTES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SCALE	SMALLINT		Scale if the attribute data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the attribute data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the attribute; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the attribute; the null value otherwise.
LOGGED	CHAR (1)		Applies to LOB types only; blank otherwise. • N = Changes are not logged • Y = Changes are logged
COMPACT	CHAR (1)		Applies to LOB types only; blank otherwise. • N = Stored in non-compact format • Y = Stored in compact format
DL_FEATURES	CHAR (10)		This column is no longer used and will be removed in a future release.
JAVA_FIELDNAME	VARCHAR (256)	Y	Reserved for future use.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

SYSCAT.AUDITPOLICIES

Each row represents an audit policy.

Table 103. SYSCAT.AUDITPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
CREATE_TIME	TIMESTAMP		Time at which the audit policy was created.
ALTER_TIME	TIMESTAMP		Time at which the audit policy was last altered.
AUDITSTATUS	CHAR (1)		Status for the AUDIT category. • B = Both • F = Failure • N = None • S = Success
CONTEXTSTATUS	CHAR (1)		Status for the CONTEXT category. • B = Both • F = Failure • N = None • S = Success
VALIDATESTATUS	CHAR (1)		Status for the VALIDATE category. • B = Both • F = Failure • N = None • S = Success
CHECKINGSTATUS	CHAR (1)		Status for the CHECKING category. • B = Both • F = Failure • N = None • S = Success
SECMAINTSTATUS	CHAR (1)		Status for the SECMAINT category. • B = Both • F = Failure • N = None • S = Success
OBJMAINTSTATUS	CHAR (1)		Status for the OBJMAINT category. • B = Both • F = Failure • N = None • S = Success
SYSADMINSTATUS	CHAR (1)		Status for the SYSADMIN category. • B = Both • F = Failure • N = None • S = Success

SYSCAT.AUDITPOLICIES

Table 103. SYSCAT.AUDITPOLICIES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXECUTESTATUS	CHAR (1)		Status for the EXECUTE category. • B = Both • F = Failure • N = None • S = Success
EXECUTEWITHDATA	CHAR (1)		Host variables and parameter markers logged with EXECUTE category. • N = No • Y = Yes
ERRORTYPE	CHAR (1)		The audit error type. • A = Audit • N = Normal
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.AUDITUSE

Each row represents an audit policy that is associated with an object.

Table 104. SYSCAT.AUDITUSE Catalog View

Column Name	Data Type	Nullable	Description
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITPOLICYID	INTEGER		Identifier for the audit policy.
OBJECTTYPE	CHAR (1)		The type of object with which this audit policy is associated. • S = MQT • T = Table • g = Authority • i = Authorization ID • x = Trusted context • Blank = Database
SUBOBJECTTYPE	CHAR (1)		If OBJECTTYPE is 'i', this is the type that the authorization ID represents. • G = Group • R = Role • U = User • Blank = Not applicable
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object for which the audit policy is in use. OBJECTSCHEMA is null if OBJECTTYPE identifies an object to which a schema does not apply.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object for which this audit policy is in use.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.

SYSCAT.BUFFERPOOLDBPARTITIONS

Each row represents a combination of a buffer pool and a member, in which the size of the buffer pool on that member is different from its default size for other members in the same database partition group (as represented in SYSCAT.BUFFERPOOLS).

Table 105. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier.
DBPARTITIONNUM	SMALLINT		Member number.
NPAGES	INTEGER		Number of pages in this buffer pool on this member.

SYSCAT.BUFFERPOOLEXCEPTIONS

Each row represents a combination of a buffer pool and a member, in which the size of the buffer pool on that member is different from its default size for other members in the same database partition group (as represented in SYSBUFFERPOOLS).

Table 106. SYSCAT.BUFFERPOOLEXCEPTIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier.
MEMBER	SMALLINT		Member number.
NPAGES	INTEGER		The number of pages in this buffer pool on this member.

SYSCAT.BUFFERPOOLS

Each row represents the configuration of a buffer pool on one database partition group of a database, or on all database partitions of a database.

Table 107. SYSCAT.BUFFERPOOLS Catalog View

Column Name	Data Type	Nullable	Description
BPNAME	VARCHAR (128)		Name of the buffer pool.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool.
DBPGNAME	VARCHAR (128)	Y	Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).
NPAGES	INTEGER		Default number of pages in this buffer pool on database partitions in this database partition group.
PAGESIZE	INTEGER		Page size for this buffer pool on database partitions in this database partition group.
ESTORE	INTEGER		Always 'N'. Extended storage no longer applies.
NUMBLOCKPAGES	INTEGER		Number of pages of the buffer pool that are to be in a block-based area. A block-based area of the buffer pool is only used by prefetchers doing a sequential prefetch.
BLOCKSIZE	INTEGER		Number of pages in a *block*.
NGNAME[1]	VARCHAR (128)	Y	Name of the database partition group (the null value if the buffer pool exists on all database partitions in the database).

Note:

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.

SYSCAT.CASTFUNCTIONS

Each row represents a cast function, not including built-in cast functions.

Table 108. SYSCAT.CASTFUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FROM_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the parameter.
FROM_TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the data type of the parameter belongs. The null value if not a module data type.
FROM_TYPENAME	VARCHAR (128)		Name of the data type of the parameter.
FROM_TYPEMODULEID	INTEGER	Y	Identifier for the module to which the data type of the parameter belongs. The null value if not a module data type.
TO_TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the result after casting.
TO_TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the data type of the result after casting belongs. The null value if not a module data type.
TO_TYPENAME	VARCHAR (128)		Name of the data type of the result after casting.
TO_TYPEMODULEID	INTEGER	Y	Identifier for the module to which the data type of the result after casting belongs. The null value if not a module data type.
FUNCSCHEMA	VARCHAR (128)		Schema name of the function.
FUNCMODULENAME	VARCHAR (128)		Unqualified name of the module to which the function belongs. The null value if not a module function.
FUNCNAME	VARCHAR (128)		Unqualified name of the function.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
FUNCMODULEID	INTEGER	Y	Identifier for the module to which the function belongs. The null value if not a module function.
ASSIGN_FUNCTION	CHAR (1)		• N = Not an assignment function • Y = Implicit assignment function

SYSCAT.CHECKS

Each row represents a check constraint or a derived column in a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 109. SYSCAT.CHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Name of the table to which this constraint applies.
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined. Used in resolving functions that are part of this constraint. Functions that were created after the constraint was defined are not chosen.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
TYPE	CHAR (1)		Type of check constraint: • C = Check constraint • F = Functional dependency • O = Constraint is an object property • S = System-generated check constraint for a GENERATED ALWAYS column
FUNC_PATH	CLOB (2K)		SQL path in effect when the constraint was defined.
TEXT	CLOB (2M)		Text of the check condition or definition of the derived column.[1]
PERCENTVALID	SMALLINT		Number of rows for which the informational constraint is valid, expressed as a percentage of the total.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the constraint.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the constraint.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the constraint.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the constraint.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the constraint.

Table 109. SYSCAT.CHECKS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

Note:

1. In the catalog view, the text of the check condition is always shown in the database code page and can contain substitution characters. The check constraint will always be applied in the code page of the target table, and will not contain any substitution characters when applied. (The check constraint will be applied based on the original text in the code page of the target table, which might not include the substitution characters.)

2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.COLAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a column.

Table 110. SYSCAT.COLAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view on which the privilege is held.
TABNAME	VARCHAR (128)		Unqualified name of the table or view on which the privilege is held.
COLNAME	VARCHAR (128)		Name of the column to which this privilege applies.
COLNO	SMALLINT		Column number of this column within the table (starting with 0).
PRIVTYPE	CHAR (1)		• R = Reference privilege • U = Update privilege
GRANTABLE	CHAR (1)		• G = Privilege is grantable • N = Privilege is not grantable

Note:

1. Privileges can be granted by column, but can be revoked only on a table-wide basis.

SYSCAT.COLCHECKS

Each row represents a column that is referenced by a check constraint or by the definition of a materialized query table. For table hierarchies, each check constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 111. SYSCAT.COLCHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the check constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the referenced column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the referenced column.
COLNAME	VARCHAR (128)		Name of the column.
USAGE	CHAR (1)		• D = Column is the child in a functional dependency • P = Column is the parent in a functional dependency • R = Column is referenced in the check constraint • S = Column is a source in the system-generated column check constraint that supports a materialized query table • T = Column is a target in the system-generated column check constraint that supports a materialized query table

SYSCAT.COLDIST

Each row represents the *n*th most frequent value of some column, or the *n*th quantile (cumulative distribution) value of the column. Applies to columns of real tables only (not views). No statistics are recorded for inherited columns of typed tables.

Table 112. 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)		• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT		If TYPE = 'F', *n* in this column identifies the *n*th most frequent value. If TYPE = 'Q', *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)	Y	Data value as a character literal or a null value.
VALCOUNT[2]	BIGINT		If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT[3]	BIGINT	Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable).

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. VALCOUNT is estimated, regardless of the sampling method that is used for collecting statistics. VALCOUNT might not be estimated if the given column is a leading column of an index that is defined on the table.
3. DISTCOUNT is collected only for columns that are the first key column in an index.

SYSCAT.COLGROUPCOLS

Each row represents a column that makes up a column group.

Table 113. SYSCAT.COLGROUPCOLS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
COLNAME	VARCHAR (128)		Name of the column in the column group.
TABSCHEMA	VARCHAR (128)		Schema name of the table for the column in the column group.
TABNAME	VARCHAR (128)		Unqualified name of the table for the column in the column group.
ORDINAL	SMALLINT		Ordinal number of the column in the column group.

SYSCAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile value of the column group.

Table 114. SYSCAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		• F = Frequency value • Q = Quantile value
ORDINAL	SMALLINT		Ordinal number of the column in the column group.
SEQNO	SMALLINT		If TYPE = 'F', *n* in this column identifies the *n*th most frequent value. If TYPE = 'Q', *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)		Data value as a character literal or a null value.

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

SYSCAT.COLGROUPDISTCOUNTS

Each row represents the distribution statistics that apply to the nth most frequent value of a column group or the nth quantile of a column group.

Table 115. SYSCAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Identifier for the column group.
TYPE	CHAR (1)		• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT		Sequence number n representing the nth TYPE value.
VALCOUNT	BIGINT		If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).

SYSCAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 116. SYSCAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPSCHEMA	VARCHAR (128)		Schema name of the column group.
COLGROUPNAME	VARCHAR (128)		Unqualified name of the column group.
COLGROUPID	INTEGER		Identifier for the column group.
COLGROUPCARD	BIGINT		Cardinality of the column group.
NUMFREQ_VALUES	SMALLINT		Number of frequent values collected for the column group.
NUMQUANTILES	SMALLINT		Number of quantiles collected for the column group.

SYSCAT.COLIDENTATTRIBUTES

Each row represents an identity column that is defined for a table.

Table 117. SYSCAT.COLIDENTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table or view that contains the column.
TABNAME	VARCHAR (128)		Unqualified name of the table or view that contains the column.
COLNAME	VARCHAR (128)		Name of the column.
START	DECIMAL (31,0)	Y	Start value of the sequence. The null value if the sequence is an alias.
INCREMENT	DECIMAL (31,0)	Y	Increment value. The null value if the sequence is an alias.
MINVALUE	DECIMAL (31,0)	Y	Minimum value of the sequence. The null value if the sequence is an alias.
MAXVALUE	DECIMAL (31,0)	Y	Maximum value of the sequence. The null value if the sequence is an alias.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. • N = Sequence cannot cycle • Y = Sequence can cycle • Blank = Sequence is an alias.
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. • N = Sequence numbers are not required to be generated in order of request • Y = Sequence numbers must be generated in order of request • Blank = Sequence is an alias.
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
SEQID	INTEGER		Identifier for the sequence or alias.

SYSCAT.COLOPTIONS

Each row contains column-specific option values.

Table 118. SYSCAT.COLOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the nickname.
TABNAME	VARCHAR (128)		Nickname for the column for which options are set.
COLNAME	VARCHAR (128)		Local column name.
OPTION	VARCHAR (128)		Name of the column option.
SETTING	CLOB (32K)		Value.

SYSCAT.COLUMNS

Each row represents a column defined for a table, view, or nickname.

Table 119. 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 or number of digits of fractional seconds if the column type is TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
DEFAULT	CLOB (64K)	Y	Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. Can also be the keyword NULL. Values might be converted from what was specified as a default value. For example, date and time constants are shown in ISO format, cast-function names are qualified with schema names, and identifiers are delimited. Null value if a DEFAULT clause was not specified or the column is a view column.

SYSCAT.COLUMNS

Table 119. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
NULLS	CHAR (1)		Nullability attribute for the column. • N = Column is not nullable • Y = Column is nullable The value can be 'N' for a view column that is derived from an expression or function. Nevertheless, such a column allows null values when the statement using the view is processed with warnings for arithmetic errors.
CODEPAGE	SMALLINT		Code page used for data in this column; 0 if the column is defined as FOR BIT DATA or is not a string type.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; the null value otherwise.
LOGGED	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. • N = Column is not logged • Y = Column is logged
COMPACT	CHAR (1)		Applies only to columns whose type is LOB or distinct based on LOB; blank otherwise. • N = Column is not compacted • Y = Column is compacted in storage
COLCARD	BIGINT		Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY[1]	VARCHAR (254)	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY[1]	VARCHAR (254)	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.

Table 119. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVGCOLLEN	INTEGER		Average space in bytes when the column is stored in database memory or a temporary table. For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.
KEYSEQ	SMALLINT	Y	The column's numerical position within the table's primary key. The null value for columns of subtables and hierarchy tables.
PARTKEYSEQ	SMALLINT	Y	The column's numerical position within the table's distribution key; 0 or the null value if the column is not in the distribution key. The null value for columns of subtables and hierarchy tables.
NQUANTILES	SMALLINT		Number of quantile values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in SYSCAT.COLDIST for this column; -1 if statistics are not gathered; -2 for inherited columns and columns of hierarchy tables.
NUMNULLS	BIGINT		Number of null values in the column; -1 if statistics are not collected.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target row type, if the type of this column is REFERENCE; null value otherwise.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the target row type, if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table), if the type of this column is REFERENCE; null value otherwise.
SOURCE_TABSCHEMA	VARCHAR (128)	Y	For columns of typed tables or views, the schema name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABSCHEMA. The null value for columns of non-typed tables and views.

Table 119. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCE_TABNAME	VARCHAR (128)	Y	For columns of typed tables or views, the unqualified name of the table or view in which the column was first introduced. For non-inherited columns, this is the same as TABNAME. The null value for columns of non-typed tables and views.
DL_FEATURES	CHAR (10)	Y	This column is no longer used and will be removed in a future release.
SPECIAL_PROPS	CHAR (8)	Y	Applies to REFERENCE type columns only; blanks otherwise. Each byte position is defined as follows: • 1 = Object identifier (OID) column ('Y' for yes; 'N' for no) • 2 = User-generated or system-generated ('U' for user; 'S' for system) Bytes 3 through 8 are reserved for future use.
HIDDEN	CHAR (1)		Type of hidden column. • I = Column is defined as IMPLICITLY HIDDEN • S = System-managed hidden column • Blank = Column is not hidden
INLINE_LENGTH	INTEGER		Maximum size in bytes of the internal representation of an instance of an XML document, a structured type, or a LOB data type, that can be stored in the base table; 0 when not applicable.
PCTINLINED	SMALLINT		Percentage of inlined data for columns with VARCHAR, VARGRAPHIC, LOB, or XML data types. -1 if statistics have not been collected or the column data type does not support storing data outside the row. Also -1 for VARCHAR and VARGRAPHIC column if the table is organized by column or the table is organized by row and the row size of the table does not exceed the maximum record length for the page size of the table space.
IDENTITY	CHAR (1)		• N = Not an identity column • Y = Identity column
ROWCHANGETIMESTAMP	CHAR (1)		• N = Not a row change timestamp column • Y = Row change timestamp column
GENERATED	CHAR (1)		Type of generated column. • A = Column value is always generated • D = Column value is generated by default • Blank = Column is not generated
TEXT	CLOB (2M)	Y	For columns defined as generated as expression, this field contains the text of the generated column expression, starting with the keyword AS.

Table 119. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMPRESS	CHAR (1)		• O = Compress off • S = Compress system default values
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 the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.
IMPLICITVALUE[2]	VARCHAR (254)	Y	For a column that was added to a table after the table was created, stores the default value at the time the column was added. For a column that was defined when the table was created, stores the null value.
SECLABELNAME	VARCHAR (128)	Y	Name of the security label that is associated with the column if it is a protected column; the null value otherwise.
ROWBEGIN	CHAR (1)		• N = Not a row begin column • Y = Row begin column
ROWEND	CHAR (1)		• N = Not a row end column • Y = Row end column
TRANSACTIONSTARTID	CHAR (1)		• N = Not a transaction start ID column • Transaction start ID column
QUALIFIER	VARCHAR (128)	Y	Reserved for future use.
FUNC_PATH	CLOB (2K)	Y	Reserved for future use.
PCTENCODED	SMALLINT		Percentage of values that are encoded as a result of compression for a column in a column-organized table; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Table 119. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description

Note:

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

2. Attaching a data partition is allowed unless IMPLICITVALUE for a specific column is a non-null value for both the source column and the target column, and the values do not match. In this case, you must drop the source table and then re-create it. A column can have a non-null value in the IMPLICITVALUE field if one of the following conditions is met:

   • The column is created as the result of an ALTER TABLE...ADD COLUMN statement

   • The IMPLICITVALUE field is propagated from a source table during attach

   • The IMPLICITVALUE field is inherited from a source table during detach

   • The IMPLICITVALUE field is set during database upgrade from Version 8 to Version 9, where it is determined to be an added column, or might be an added column. If the database is not certain whether the column is added or not, it is treated as added. An added column is a column that was created as the result of an ALTER TABLE...ADD COLUMN statement.

   To avoid these inconsistencies during non-migration scenarios, it is recommended that you always create the tables that you are going to attach with all the columns already defined. That is, never use the ALTER TABLE statement to add columns to a table before attaching it.

SYSCAT.COLUSE

Each row represents a column that is referenced in the DIMENSIONS clause of a CREATE TABLE statement.

Table 120. SYSCAT.COLUSE Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
DIMENSION	SMALLINT		Dimension number, based on the reverse order of dimensions specified in the DIMENSIONS clause (initial position is 1). For a composite dimension, this value will be the same for each component of the dimension.
COLSEQ	SMALLINT		Numeric position of the column in the dimension to which it belongs (initial position is 1). The value is 1 for the single column in a noncomposite dimension.
TYPE	CHAR (1)		Type of dimension. • C = Clustering or multidimensional clustering • P = Partitioning

SYSCAT.CONDITIONS

Each row represents a condition defined in a module.

Table 121. SYSCAT.CONDITIONS Catalog View

Column Name	Data Type	Nullable	Description
CONDSCHEMA	VARCHAR (128)		Schema name of the condition.
CONDMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the condition belongs.
CONDNAME	VARCHAR (128)		Unqualified name of the condition.
CONDID	INTEGER		Identifier for the condition.
CONDMODULEID	INTEGER	Y	Identifier of the module to which the condition belongs.
SQLSTATE	CHAR (5)	Y	SQLSTATE value associated with the condition.
OWNER	VARCHAR (128)		Authorization ID of the owner of the condition.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the condition was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.CONSTDEP

Each row represents a dependency of a constraint on some other object. The constraint depends on the object of type BTYPE of name BNAME, so a change to the object affects the constraint.

Table 122. 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: • F = Routine • I = Index • R = User-defined structured type • u = Module alias
BSCHEMA	VARCHAR (128)		Schema name of the object on which the constraint depends.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which the constraint depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the constraint depends.

SYSCAT.CONTEXTATTRIBUTES

Each row represents a trusted context attribute.

Table 123. SYSCAT.CONTEXTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
ATTR_NAME	VARCHAR (128)		Name of the attribute. One of: • ADDRESS • ENCRYPTION
ATTR_VALUE	VARCHAR (128)		Value of the attribute.
ATTR_OPTIONS	VARCHAR (128)	Y	If ATTR_NAME is 'ADDRESS', specifies the level of encryption required for this specific address. A null value indicates that the global ENCRYPTION attribute applies.

SYSCAT.CONTEXTS

Each row represents a trusted context.

Table 124. SYSCAT.CONTEXTS Catalog View

Column Name	Data Type	Nullable	Description
CONTEXTNAME	VARCHAR (128)		Name of the trusted context.
CONTEXTID	INTEGER		Identifier for the trusted context.
SYSTEMAUTHID	VARCHAR (128)		The system authorization ID associated with the trusted context.
DEFAULTCONTEXTROLE	VARCHAR (128)	Y	The default role for the context.
CREATE_TIME	TIMESTAMP		Time at which the trusted context was created.
ALTER_TIME	TIMESTAMP		Time at which the trusted context was last altered.
ENABLED	CHAR (1)		Trusted context state. • N = Disabled • Y = Enabled
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.CONTROLDEP

Each row represents a dependency of a row permission or column mask on some other object.

Table 125. SYSCAT.CONTROLDEP Catalog View

Column Name	Data Type	Nullable	Description
DSCHEMA	VARCHAR (128)		Schema name of the row permission or column mask.
DNAME	VARCHAR (128)		Unqualified name of the row permission or column mask.
DTYPE	CHAR (1)		Type of the depending object. • y = Row Permission • 2 = Column Mask
BTYPE	CHAR (1)		Type of object on which there is a dependency: • A = Table alias • C = Column • F = Routine • H = Hierarchy table • I = Index • L = Detached table • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • s = Statistical Table • u = Module alias • v = Global variable
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name for the module of the object on which the control depends. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the control depends.
BCOLNAME	VARCHAR (128)	Y	If BTYPE = 'C', the column name on which there is a dependency; the null value otherwise.

SYSCAT.CONTROLS

Each row represents a row permission or column mask.

Table 126. SYSCAT.CONTROLS Catalog View

Column Name	Data Type	Nullable	Description
CONTROLSCHEMA	VARCHAR (128)		Schema of the row permission or column mask.
CONTROLNAME	VARCHAR (128)		Name of the row permission or column mask.
OWNER	VARCHAR (128)		Owner of the row permission or column mask.
OWNERTYPE	CHAR (1)		• S = System • U = User
TABSCHEMA	VARCHAR (128)		Schema of the table on which the row permission or column mask is defined.
TABNAME	VARCHAR (128)		Name of the table on which the row permission or column mask is defined.
COLNAME	VARCHAR (128)		Column name on which the column mask is defined. Blank if this is a row permission.
CONTROLID	INTEGER		Identifier of the control.
CONTROLTYPE	CHAR (1)		• C = Column mask • R = Row permission
ENFORCED	CHAR (1)		• A = All access
IMPLICIT	CHAR (1)		• N = The row permission was explicitly created or this is a column mask • Y = The row permission was implicitly created
ENABLE	CHAR (1)		• N = Not enabled • Y = Enabled
VALID	CHAR (1)		• N = The control is invalid • Y = The control is valid
RULETEXT	CLOB (2M)		The source text of the search condition or case expression portion of the CREATE PERMISSION or CREATE MASK statement.
TABCORRELATION	VARCHAR (128)		The correlation name of the table on which the row permission or column mask is defined. Blank if not specified.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the control was defined.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the control.

SYSCAT.CONTROLS

Table 126. SYSCAT.CONTROLS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the control.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for the ORDER BY clauses in the control.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for the ORDER BY clauses in the control.
CREATE_TIME	TIMESTAMP		Time at which the row permission or column mask was created.
ALTER_TIME	TIMESTAMP		Time at which the row permission or column mask was last altered.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.DATAPARTITIONEXPRESSION

Each row represents an expression for that part of the table partitioning key.

Table 127. SYSCAT.DATAPARTITIONEXPRESSION Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the partitioned table.
TABNAME	VARCHAR (128)		Unqualified name of the partitioned table.
DATAPARTITIONKEYSEQ	INTEGER		Expression key part sequence ID, starting from 1.
DATAPARTITIONEXPRESSION	CLOB (32K)		Expression for this entry in the sequence, in SQL syntax.
NULLSFIRST	CHAR (1)		• N = Null values in this expression compare high • Y = Null values in this expression compare low

SYSCAT.DATAPARTITIONS

Each row represents a data partition. Note that the data partition statistics represent one database partition if the table is created on multiple database partitions.

Table 128. 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.
TBSPACEID	INTEGER	Y	Identifier for the table space in which this data partition is stored. The null value when STATUS is 'I'.
PARTITIONOBJECTID	INTEGER	Y	Identifier for the data partition within the table space.
LONG_TBSPACEID	INTEGER	Y	Identifier for the table space in which long data is stored. The null value when STATUS is 'I'.
ACCESS_MODE	CHAR (1)		Access restriction state of the data partition. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access
STATUS	VARCHAR (32)		• A = Data partition is newly attached • D = Data partition is detached and detached dependents are to be incrementally maintained with respect to the content of this partition • I = Detached data partition whose entry in the catalog is maintained only during asynchronous index cleanup; rows with a STATUS value of 'I' are removed when all index records referring to the detached partition have been deleted • L = Data partition is logically detached • Empty string = Data partition is visible (normal status) Bytes 2 through 32 are reserved for future use.
SEQNO	INTEGER		Data partition sequence number (starting from 0).
LOWINCLUSIVE	CHAR (1)		• N = Low key value is not inclusive • Y = Low key value is inclusive

Table 128. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOWVALUE	VARCHAR (512)		Low key value (a string representation of an SQL value) for this data partition.
HIGHINCLUSIVE	CHAR (1)		• N = High key value is not inclusive • Y = High key value is inclusive
HIGHVALUE	VARCHAR (512)		High key value (a string representation of an SQL value) for this data partition.
CARD	BIGINT		Total number of rows in the data partition; -1 if statistics are not collected.
OVERFLOW	BIGINT		Total number of overflow records in the data partition; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the data partition exist; -1 if statistics are not collected.
FPAGES	BIGINT		Total number of pages in the data partition; -1 if statistics are not collected.
ACTIVE_BLOCKS	BIGINT		Total number of active blocks in the data partition, or -1. Applies to multidimensional clustering (MDC) tables only.
INDEX_TBSPACEID	INTEGER		Identifier for the table space which holds all partitioned indexes for this data partition.
AVGROWSIZE	SMALLINT		Average length (in bytes) of both compressed and uncompressed rows in this data partition; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Compressed rows as a percentage of the total number of rows in the data partition; -1 if statistics are not collected.
PCTPAGESAVED	SMALLINT		Approximate percentage of pages saved in the data partition as a result of row compression. This value includes overhead bytes for each user data row in the data partition, but does not include the space that is consumed by dictionary overhead; -1 if statistics are not collected.
AVGCOMPRESSEDROWSIZE	SMALLINT		Average length (in bytes) of compressed rows in this data partition; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		For compressed rows in the data partition, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. Null if statistics are not collected.

SYSCAT.DATAPARTITIONS

Table 128. SYSCAT.DATAPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LASTUSED	DATE		Date when the data partition was last used by any DML statement or the LOAD command. If the table is not partitioned, only the LASTUSED value in SYSCAT.TABLES is updated. This column is not updated when the data partition is used on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.

SYSCAT.DATATYPEDEP

Each row represents a dependency of a user-defined data type on some other object.

Table 129. SYSCAT.DATATYPEDEP Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the data type.
TYPEMODULENAME	VARCHAR (128)	Y	Module name of the data type.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module of the data type.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • G = Global temporary table • H = Hierarchy table • I = Index • N = Nickname • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Module name of the object on which there is a dependency.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent data type; the null value otherwise.

SYSCAT.DATATYPES

Each row represents a built-in or user-defined data type.

Table 130. SYSCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the user-defined type belongs. The null value if not a module user-defined type.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
OWNER	VARCHAR (128)		Authorization ID of the owner of the type.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SOURCESCHEMA	VARCHAR (128)	Y	For distinct types or array types, the schema name of the source data type. For user-defined structured types, the schema name of the built-in type of the reference representation type. Null for other data types.
SOURCEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the source data type belongs. The null value if not a module source data type.
SOURCENAME	VARCHAR (128)	Y	For distinct types or array types, the unqualified name of the source data type. For user-defined structured types, the unqualified built-in type name of the reference representation type. Null for other data types.
METATYPE	CHAR (1)		• A = User-defined array type • C = User-defined cursor type • F = User-defined row type • L = User-defined associative array type • R = User-defined structured type • S = System predefined type • T = User-defined distinct type
TYPERULES	CHAR(1)		Indicates the type rules for the user-defined type. • S = Strong typing • W = Weak typing • Blank = Not applicable
TYPEID	SMALLINT		Identifier for the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module to which the user-defined type belongs. The null value if not a module user-defined type.

Table 130. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SOURCETYPEID	SMALLINT	Y	Identifier for the source type (the null value for built-in types). For user-defined structured types, this is the identifier of the reference representation type.
SOURCEMODULEID	INTEGER	Y	Identifier for the module to which the source data type belongs. The null value if not a module source data type.
PUBLISHED	CHAR (1)		Indicates whether the module user-defined type can be referenced outside its module. • N = The module user-defined type is not published • Y = The module user-defined type is published • Blank = Not applicable
LENGTH	INTEGER		Maximum length of the type. 0 for built-in parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this is the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the built-in DECIMAL type; the number of digits of fractional seconds for distinct types based on the built-in TIMESTAMP type; 6 for the built-in TIMESTAMP type; 0 for all other types (including DECIMAL itself).
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		Database code page for string types, distinct types based on string types, or reference representation types; 0 otherwise.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the data type; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the data type; the null value otherwise.
ARRAY_LENGTH	INTEGER	Y	Maximum cardinality of the array. The null value if METATYPE is not 'A'.
ARRAYINDEXTYPESCHEMA	VARCHAR (128)	Y	Schema of the data type of the array index. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPENAME	VARCHAR (128)	Y	Name of the data type of the array index. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPEID	SMALLINT	Y	Identifier for the array index type. The null value if METATYPE is not 'L'.

SYSCAT.DATATYPES

Table 130. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ARRAYINDEXTYPELENGTH	INTEGER	Y	Maximum length of the array index data type. The null value if METATYPE is not 'L'.
ARRAYINDEXTYPE_ STRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string array index data type. Otherwise, the null value.
ARRAYINDEXTYPE_ STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string array index data type. Otherwise, the null value.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
VALID	CHAR (1)		• N = The data type is invalid • Y = The data type is valid
ATTRCOUNT	SMALLINT		Number of attributes in the data type.
INSTANTIABLE	CHAR (1)		• N = Type cannot be instantiated • Y = Type can be instantiated
WITH_FUNC_ACCESS	CHAR (1)		• N = Methods for this type cannot be invoked using function notation. • Y = All the methods for this type can be invoked using function notation.
FINAL	CHAR (1)		• N = The user-defined type can have subtypes. • Y = The user-defined type cannot have subtypes.
INLINE_LENGTH	INTEGER		Maximum length of a structured type that can be kept with a base table row; 0 otherwise.
NATURAL_ INLINE_LENGTH	INTEGER	Y	System-generated natural inline length of a structured type instance. The null value if this type is not a structured type.
JARSCHEMA	VARCHAR (128)	Y	Schema name of the JAR_ID that identifies the Jar file containing the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.
JAR_ID	VARCHAR (128)	Y	Identifier for the Jar file that contains the Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.
CLASS	VARCHAR (384)	Y	Java class that implements the SQL type. The null value if the EXTERNAL NAME clause is not specified.
SQLJ_REPRESENTATION	CHAR (1)	Y	SQLJ "representation_spec" of the Java class that implements the SQL type. The null value if the EXTERNAL NAME ... LANGUAGE JAVA REPRESENTATION SPEC clause is not specified. • D = SQL data • S = Serializable
ALTER_TIME	TIMESTAMP		Time at which the data type was last altered.

Table 130. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the type.
NULLS	CHAR (1)		Nullability attribute for the type. • N = Type is not nullable • Y = Type is nullable
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the type was created. The null value if no data type check constraint exists.
CONSTRAINT_TEXT	CLOB (64K)	Y	Predicate text which constrains the values allowed for the data type. The null value if no data type check constraint exists.
LAST_REGEN_TIME	TIMESTAMP		Time at which the data type check constraint was last regenerated. The same value as CREATE_TIME if no data type check constraint exists.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.DBAUTH

Each row represents a user, group, or role that has been granted one or more database-level authorities.

Table 131. SYSCAT.DBAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the authority.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the authority.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
BINDADDAUTH	CHAR (1)		Authority to create packages. • N = Not held • Y = Held
CONNECTAUTH	CHAR (1)		Authority to connect to the database. • N = Not held • Y = Held
CREATETABAUTH	CHAR (1)		Authority to create tables. • N = Not held • Y = Held
DBADMAUTH	CHAR (1)		DBADM authority. • N = Not held • Y = Held
EXTERNALROUTINEAUTH	CHAR (1)		Authority to create external routines. • N = Not held • Y = Held
IMPLSCHEMAAUTH	CHAR (1)		Authority to implicitly create schemas by creating objects in non-existent schemas. • N = Not held • Y = Held
LOADAUTH	CHAR (1)		Authority to use the DB2 load utility. • N = Not held • Y = Held
NOFENCEAUTH	CHAR (1)		Authority to create non-fenced user-defined functions. • N = Not held • Y = Held
QUIESCECONNECTAUTH	CHAR (1)		Authority to access the database when it is quiesced. • N = Not held • Y = Held
LIBRARYADMAUTH	CHAR (1)		Reserved for future use.

Table 131. SYSCAT.DBAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
SECURITYADMAUTH	CHAR (1)		Authority to administer database security. • N = Not held • Y = Held
SQLADMAUTH	CHAR (1)		Authority to monitor and tune SQL statements. • N = Not held • Y = Held
WLMADMAUTH	CHAR (1)		Authority to manage WLM objects. • N = Not held • Y = Held
EXPLAINAUTH	CHAR (1)		Authority to explain SQL statements without requiring actual privileges on the objects in the statement. • N = Not held • Y = Held
DATAACCESSAUTH	CHAR (1)		Authority to access data. • N = Not held • Y = Held
ACCESSCTRLAUTH	CHAR (1)		Authority to grant and revoke database object privileges. • N = Not held • Y = Held
CREATESECUREAUTH	CHAR (1)		Authority to create secure objects. • N = Not held • Y = Held

SYSCAT.DBPARTITIONGROUPDEF

Each row represents a database partition that is contained in a database partition group.

Table 132. 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. • A = The newly added database partition is not in the distribution map, but the containers for the table spaces in the database partition group have been created; the database partition is added to the distribution map when a redistribute database partition group operation has completed successfully. • D = The database partition will be dropped when a redistribute database partition group operation has completed successfully. • T = The newly added database partition is not in the distribution map, and it was added using the WITHOUT TABLESPACES clause; containers must be added to the table spaces in the database partition group. • Y = The database partition is in the distribution map.

SYSCAT.DBPARTITIONGROUPS

Each row represents a database partition group.

Table 133. SYSCAT.DBPARTITIONGROUPS Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR (128)		Name of the database partition group.
OWNER	VARCHAR (128)		Authorization ID of the owner of the database partition group.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
PMAP_ID	SMALLINT		Identifier for the distribution map in the SYSCAT.PARTITIONMAPS catalog view.
REDISTRIBUTE_PMAP_ID	SMALLINT		Identifier for the distribution map currently being used for redistribution; -1 if redistribution is currently not in progress.
CREATE_TIME	TIMESTAMP		Creation time of the database partition group.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the database partition group.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.EVENTMONITORS

Each row represents an event monitor.

Table 134. SYSCAT.EVENTMONITORS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.
OWNER	VARCHAR (128)		Authorization ID of the owner of the event monitor.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TARGET_TYPE	CHAR (1)		Type of target to which event data is written. • F = File • P = Pipe • T = Table • U = Unformatted event table
TARGET	VARCHAR (762)		Name of the target to which file or pipe event monitor data is written. For files, it can be either an absolute path name or a relative path name (relative to the database path for the database; this can be seen using the LIST ACTIVE DATABASES command). For pipes, it can be an absolute path name.
MAXFILES	INTEGER	Y	Maximum number of event files that this event monitor permits in an event path. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file).
MAXFILESIZE	INTEGER	Y	Maximum size (in 4K pages) that each event file can attain before the event monitor creates a new file. The null value if there is no maximum, or if TARGET_TYPE is not 'F' (file).
BUFFERSIZE	INTEGER	Y	Size of the buffer (in 4K pages) that is used by event monitors with file targets; null value otherwise.
IO_MODE	CHAR (1)	Y	Mode of file input/output (I/O). • B = Blocked • N = Not blocked • Null value = TARGET_TYPE is not 'F' (file) or 'T' (table)
WRITE_MODE	CHAR (1)	Y	Indicates how this event monitor handles existing event data when the monitor is activated. • A = Append • R = Replace • Null value = TARGET_TYPE is not 'F' (file)

Table 134. SYSCAT.EVENTMONITORS Catalog View (continued)

Column Name	Data Type	Nullable	Description
AUTOSTART	CHAR (1)		Indicates whether this event monitor is to be activated automatically when the database starts. • N = No • Y = Yes
DBPARTITIONNUM[1]	SMALLINT		This column is deprecated and will be removed in a future release. Replaced by MEMBER.
MONSCOPE	CHAR (1)		Monitoring scope. • G = Global • L = Local • T = Each database partition on which the table space exists • Blank = WRITE TO TABLE event monitor
EVMON_ACTIVATES	INTEGER		Number of times the event monitor has been activated.
NODENUM[1]	SMALLINT		This column is deprecated and will be removed in a future release. Replaced by MEMBER.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the event monitor.
VERSIONNUMBER	INTEGER		Version, release, and modification level in which the event monitor was created or last upgraded.
MEMBER	SMALLINT		Number of the member where the event monitor runs and logs events.
REMARKS	VARCHAR (254)	Y	Reserved for future use.

Note:

1. The NODENUM column is included for backwards compatibility. See DBPARTITIONNUM.
2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.EVENTS

Each row represents an event that is being monitored. An event monitor, in general, monitors multiple events.

Table 135. SYSCAT.EVENTS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor that is monitoring this event.
TYPE	VARCHAR (128)		Type of event being monitored. Possible values are: • ACTIVITIES • CHANGEHISTORY • CONNECTIONS • DATABASE • DEADLOCKS • DETAILDEADLOCKS • DLOCKWHIST • DLOCKWHISTAVAL • LOCKING • PKGCACHEBASE • PKGCACHEDETAILED • STATEMENTS • TABLES • TABLESPACES • THRESHOLDVIOLATIONS • TRANSACTIONS • STATISTICS • UOW
FILTER	CLOB (64K)	Y	Full text of the WHERE clause that applies to this event.

SYSCAT.EVENTTABLES

Each row represents the target table of an event monitor that writes to SQL tables.

Table 136. SYSCAT.EVENTTABLES Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR (128)		Name of the event monitor.
LOGICAL_GROUP	VARCHAR (128)		Name of the logical data group. Possible values are: • ACTIVITYHISTORY • BUFFERPOOL • CHANGESUMMARY • CONN • CONNHEADER • CONTROL • DATAVAL • DB • DBDBMCFG • DDLSTMTEXEC • DEADLOCK • DLCONN • DLLOCK • EVMONSTART • LOCKING • PKGCACHEBASE • PKGCACHEDETAILED • REGVAR • SCSTATS • STMT • STMTHIST • STMTVALS • SUBSECTION • TABLE • TABLESPACE • THRESHOLDVIOLATIONS • TXNCOMPLETION • UOW • UTILLOCATION • UTILPHASE • UTILSTART • UTILSTOP • WCSTATS • WLSTATS • XACT
TABSCHEMA	VARCHAR (128)		Schema name of the target table.
TABNAME	VARCHAR (128)		Unqualified name of the target table.

SYSCAT.EVENTTABLES

Table 136. SYSCAT.EVENTTABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PCTDEACTIVATE	SMALLINT		A percent value that specifies how full a DMS table space must be before an event monitor automatically deactivates. Set to 100 for SMS table spaces.
TABOPTIONS	VARCHAR (32)		String indicating the logical data group options for the target table. Each character in the string represents an option. Possible values are: • E = EXCLUDES • I = INCLUDES • T = TRUNC • Blank = Not applicable

SYSCAT.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 137. SYSCAT.FULLHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. • R = Between structured types • U = Between typed tables • W = Between typed views
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.
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 138. 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 139. SYSCAT.FUNCMAPPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping.
ORDINAL	SMALLINT		Position of the parameter.
LOCATION	CHAR (1)		Location of the parameter. • L = Local parameter • R = Remote parameter
OPTION	VARCHAR (128)		Name of the function mapping parameter option.
SETTING	VARCHAR (2048)		Value of the function mapping parameter option.

SYSCAT.FUNCMAPPINGS

Each row represents a function mapping.

Table 140. SYSCAT.FUNCMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR (128)		Name of the function mapping (might be system-generated).
FUNCSCHEMA	VARCHAR (128)	Y	Schema name of the function. If the null value, the function is assumed to be a built-in function.
FUNCNAME	VARCHAR (1024)	Y	Unqualified name of the user-defined or built-in function.
FUNCID	INTEGER	Y	Identifier for the function.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
OWNER	VARCHAR (128)		Authorization ID of the owner of the mapping. 'SYSIBM' indicates that this is a built-in function.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
WRAPNAME	VARCHAR (128)	Y	Wrapper to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Name of the data source.
SERVERTYPE	VARCHAR (30)	Y	Type of data source to which this mapping applies.
SERVERVERSION	VARCHAR (18)	Y	Version of the server type to which this mapping applies.
CREATE_TIME	TIMESTAMP		Time at which the mapping was created.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the mapping. 'SYSIBM' indicates that this is a built-in function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.HIERARCHIES

Each row represents the relationship between a subtable and its immediate supertable, a subtype and its immediate supertype, or a subview and its immediate superview. Only immediate hierarchical relationships are included in this view.

Table 141. SYSCAT.HIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR (1)		Relationship type. • R = Between structured types • U = Between typed tables • W = Between typed views
SUB_SCHEMA	VARCHAR (128)		Schema name of the subtype, subtable, or subview.
SUB_NAME	VARCHAR (128)		Unqualified name of the subtype, subtable, or subview.
SUPER_SCHEMA	VARCHAR (128)		Schema name of the supertype, supertable, or superview.
SUPER_NAME	VARCHAR (128)		Unqualified name of the supertype, supertable, or superview.
ROOT_SCHEMA	VARCHAR (128)		Schema name of the table, view, or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR (128)		Unqualified name of the table, view, or type that is at the root of the hierarchy.

SYSCAT.HISTOGRAMTEMPLATEBINS

Each row represents a histogram template bin.

Table 142. 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 143. SYSCAT.HISTOGRAMTEMPLATES Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATEID	INTEGER		Identifier for the histogram template.
TEMPLATENAME	VARCHAR (128)		Name of the histogram template.
CREATE_TIME	TIMESTAMP		Time at which the histogram template was created.
ALTER_TIME	TIMESTAMP		Time at which the histogram template was last altered.
NUMBINS	INTEGER		Number of bins in the histogram template, including the last bin that has an unbounded top value.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.HISTOGRAMTEMPLATEUSE

Each row represents a relationship between a workload management object that can use histogram templates and a histogram template.

Table 144. SYSCAT.HISTOGRAMTEMPLATEUSE Catalog View

Column Name	Data Type	Nullable	Description
TEMPLATENAME	VARCHAR (128)	Y	Name of the histogram template.
TEMPLATEID	INTEGER		Identifier for the histogram template.
HISTOGRAMTYPE	CHAR (1)		The type of information collected by histograms based on this template. • C = Activity estimated cost histogram • E = Activity execution time histogram • I = Activity interarrival time histogram • L = Activity life time histogram • Q = Activity queue time histogram • R = Request execution time histogram • U = Unit of work life time histogram
OBJECTTYPE	CHAR (1)		The type of WLM object. • b = Service class • k = Work action • w = Workload
OBJECTID	INTEGER		Identifier of the WLM object.
SERVICECLASSNAME	VARCHAR (128)	Y	Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	The name of the parent service class of the service subclass that uses the histogram template.
WORKACTIONNAME	VARCHAR (128)	Y	The name of the work action that uses the histogram template.
WORKACTIONSETNAME	VARCHAR (128)	Y	The name of the work action set containing the work action that uses the histogram template.
WORKLOADNAME	VARCHAR (128)	Y	The name of the workload that uses the histogram template.

SYSCAT.INDEXAUTH

Each row represents a user, group, or role that has been granted CONTROL privilege on an index.

Table 145. SYSCAT.INDEXAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held

SYSCAT.INDEXCOLUSE

Each row represents a column that participates in an index.

Table 146. SYSCAT.INDEXCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the index (initial position is 1).
COLORDER	CHAR (1)		Order of the values in this index column. Possible values are: • A = Ascending • D = Descending • I = INCLUDE column (ordering ignored) • R = Random
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the column; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the column; the null value otherwise.
VIRTUAL	CHAR (1)	Y	• N = Column exists in the table on which this index is defined. • S = Virtual index column found in statistical view associated with the index. The expression for this part of the index key is stored in the TEXT column. • Y = Virtual index column that does not exist in the table on which this index is defined.
TEXT	CLOB (64K)	Y	The expression text for this part of the index key. The null value if this part of the index key is not based on an expression.

SYSCAT.INDEXDEP

Each row represents a dependency of an index on some other object. The index depends on an object of type BTYPE and name BNAME, so a change to the object affects the index.

Table 147. 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.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

SYSCAT.INDEXDEP

Table 147. SYSCAT.INDEXDEP 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 the dependent index; the null value otherwise.

SYSCAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 148. SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OWNER	VARCHAR (128)		Authorization ID of the owner of the index.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)		Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in the next release. Use SYSCAT.INDEXCOLUSE for this information.
UNIQUERULE	CHAR (1)		Unique rule. • D = Permits duplicates • U = Unique • P = Implements primary key
MADE_UNIQUE	CHAR (1)		• N = Index remains as it was created. • Y = Index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index reverts to being non-unique.
COLCOUNT	SMALLINT		Number of columns in the key, plus the number of include columns, if any.
UNIQUE_COLCOUNT	SMALLINT		Number of columns required for a unique key. It is always <= COLCOUNT, and < COLCOUNT only if there are include columns; -1 if the index has no unique key (that is, it permits duplicates).

SYSCAT.INDEXES

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INDEXTYPE[5]	CHAR (4)		Type of index. • BLOK = Block index • CLUS = Clustering index (controls the physical placement of newly inserted rows) • CPMA = Page map index for a column-organized table • DIM = Dimension block index • RCT = Key sequence index for a range-clustered table • REG = Regular index • TEXT = Text index • XPTH = XML path index • XRGN = XML region index • XVIL = Index over XML column (logical) • XVIP = Index over XML column (physical)
ENTRYTYPE	CHAR (1)		• H = This row represents an index on a hierarchy table • L = This row represents a logical index on a typed table • Blank = This row represents an index on an untyped table
PCTFREE	SMALLINT		Percentage of each index page to be reserved during the initial building of the index. This space is available for data insertions after the index has been built.
IID	SMALLINT		Identifier for the index.
NLEAF	BIGINT		Number of leaf pages; -1 if statistics are not collected.
NLEVELS	SMALLINT		Number of index levels; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.
FULLKEYCARD	BIGINT		Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO[3]	SMALLINT		Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CLUSTERFACTOR[3]	DOUBLE		Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; 0 otherwise.

SYSTEM_REQUIRED	SMALLINT		• 1 if one or the other of the following conditions is met:
			  – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multidimensional clustering (MDC) table or an insert time clustering (ITC) table.
			  – This is the index on the object identifier (OID) column of a typed table.
			• 2 if both of the following conditions are met:
			  – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table or an ITC table.
			  – This is the index on the OID column of a typed table.
			• 0 otherwise.

CREATE_TIME	TIMESTAMP		Time when the index was created.
STATS_TIME	TIMESTAMP	Y	Last time that any change was made to the recorded statistics for this index. The null value if no statistics are available.
PAGE_FETCH_PAIRS[3]	VARCHAR (520)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
MINPCTUSED	SMALLINT		A non-zero integer value indicates that the index is enabled for online defragmentation, and represents the minimum percentage of used space on a page before a page merge can be attempted. A zero value indicates that no page merge is attempted.

SYSCAT.INDEXES

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
REVERSE_SCANS	CHAR (1)		• N = Index does not support reverse scans • Y = Index supports reverse scans
INTERNAL_FORMAT	SMALLINT		Possible values are: • 1 = Index does not have backward pointers • 2 or greater = Index has backward pointers • 6 = Index is a composite block index
COMPRESSION	CHAR (1)		Specifies whether index compression is activated • N = Not activated • Y = Activated
IESCHEMA	VARCHAR (128)	Y	Schema name of the index extension. The null value for ordinary indexes.
IENAME	VARCHAR (128)	Y	Unqualified name of the index extension. The null value for ordinary indexes.
IEARGUMENTS	CLOB (64K)	Y	External information of the parameter specified when the index is created. The null value for ordinary indexes.
INDEX_OBJECTID	INTEGER		Identifier for the index object.
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_FETCH_ PAGES[1,2]	DOUBLE		Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES[2]	DOUBLE		Average number of random table pages between sequential page accesses; -1 if not known.
AVERAGE_SEQUENCE_GAP[2]	DOUBLE		Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_FETCH_ GAP[1,2]	DOUBLE		Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVERAGE_SEQUENCE_PAGES[2]	DOUBLE		Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_FETCH_ PAGES[1,2]	DOUBLE		Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
TBSPACEID	INTEGER		Identifier for the index table space.
LEVEL2PCTFREE	SMALLINT		Percentage of each index level 2 page to be reserved during initial building of the index. This space is available for future inserts after the index has been built.
PAGESPLIT	CHAR (1)		Index page split behavior. • H = High • L = Low • S = Symmetric • Blank = Not applicable
AVGPARTITION_ CLUSTERRATIO[3]	SMALLINT		Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR[3]	DOUBLE		Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_FETCH_ PAIRS[3]	VARCHAR (520)		A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
PCTPAGESSAVED	SMALLINT		Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE		A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	BIGINT		Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.
AVGLEAFKEYSIZE	INTEGER		Average index key size for keys on leaf pages in the index.

SYSCAT.INDEXES

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVGNLEAFKEYSIZE	INTEGER		Average index key size for keys on non-leaf pages in the index.
OS_PTR_SIZE	INTEGER		Platform word size with which the index was created. • 32 = 32-bit • 64 = 64-bit
COLLECTSTATISTCS	CHAR (1)		Specifies how statistics were collected at index creation time. • D = Collect detailed index statistics • S = Collect sampled detailed index statistics • Y = Collect basic index statistics • Blank = Do not collect index statistics
DEFINER[4]	VARCHAR (128)		Authorization ID of the owner of the index.
LASTUSED	DATE		Date when the index was last used by any DML statement to perform a scan, or used to enforce referential integrity constraints. This column is not updated when the index is used on an HADR standby database, nor is it updated when rows are inserted into the table on which the index is defined. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
PERIODNAME	VARCHAR(128)	Y	Name of the period used to define this index.
PERIODPOLICY	CHAR (1)		If a period name was specified, the index uses this period policy. • N = Not applicable • O = Period overlaps not allowed
MADE_WITHOUTOVERLAPS	CHAR (1)		• N = Index remains as it was created. • Y = Index was converted to enforce WITHOUT OVERLAPS on the application period to support a primary or unique constraint. If the constraint is dropped, the index reverts to the original state.
NULLKEYS	CHAR (1)		• N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause)
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the index was defined with an expression in the key. The null value if the key does not include any expressions.

Table 148. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	VARCHAR(128)	Y	Schema name of the statistical view associated with the index key, if the key includes at least one expression. The null value if there are no expressions in the key.
VIEWNAME	VARCHAR(128)	Y	Unqualified name of the statistical view associated with the index key, if the key includes at least one expression. The null value if there are no expressions in the key.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. When using DMS table spaces, this statistic cannot be computed.

2. Prefetch statistics are not gathered during a LOAD...STATISTICS USE PROFILE, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.

3. AVGPARTITION_CLUSTERRATIO, AVGPARTITION_CLUSTERFACTOR, and AVGPARTITION_PAGE_FETCH_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE_FETCH_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.

4. The DEFINER column is included for backwards compatibility. See OWNER.

5. The XPTH, XRGN, and XVIP indexes are not recognized by any application programming interface that returns index metadata.

SYSCAT.INDEXEXPLOITRULES

Each row represents an index exploitation rule.

Table 149. SYSCAT.INDEXEXPLOITRULES Catalog View

Column Name	Data Type	Nullable	Description
FUNCID	INTEGER		Identifier for the function.
SPECID	SMALLINT		Number of the predicate specification.
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
RULEID	SMALLINT		Identifier for the exploitation rule.
SEARCHMETHODID	SMALLINT		Identifier for the search method in the specific index extension.
SEARCHKEY	VARCHAR (640)		Key used to exploit the index.
SEARCHARGUMENT	VARCHAR (2700)		Search arguments used to exploit the index.
EXACT	CHAR (1)		• N = Index lookup is not exact in terms of predicate evaluation • Y = Index lookup is exact in terms of predicate evaluation

SYSCAT.INDEXEXTENSIONDEP

Each row represents a dependency of an index extension on some other object. The index extension depends on the object of type BTYPE of name BNAME, so a change to the object affects the index extension.

Table 150. SYSCAT.INDEXEXTENSIONDEP Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

SYSCAT.INDEXEXTENSIONDEP

Table 150. SYSCAT.INDEXEXTENSIONDEP 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 the dependent index extension; the null value otherwise.

SYSCAT.INDEXEXTENSIONMETHODS

Each row represents a search method. An index extension can contain more than one search method.

Table 151. SYSCAT.INDEXEXTENSIONMETHODS Catalog View

Column Name	Data Type	Nullable	Description
METHODNAME	VARCHAR (128)		Name of the search method.
METHODID	SMALLINT		Number of the method in the index extension.
IESCHEMA	VARCHAR (128)		Schema name of the index extension on which this method is defined.
IENAME	VARCHAR (128)		Unqualified name of the index extension on which this method is defined.
RANGEFUNCSCHEMA	VARCHAR (128)		Schema name of the range-through function.
RANGEFUNCNAME	VARCHAR (128)		Unqualified name of the range-through function.
RANGESPECIFICNAME	VARCHAR (128)		Function-specific name of the range-through function.
FILTERFUNCSCHEMA	VARCHAR (128)	Y	Schema name of the filter function.
FILTERFUNCNAME	VARCHAR (128)	Y	Unqualified name of the filter function.
FILTERSPECIFICNAME	VARCHAR (128)	Y	Function-specific name of the filter function.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.INDEXEXTENSIONPARMS

Each row represents an index extension instance parameter or source key column.

Table 152. SYSCAT.INDEXEXTENSIONPARMS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
ORDINAL	SMALLINT		Sequence number of the parameter or key column.
PARMNAME	VARCHAR (128)		Name of the parameter or key column.
TYPESCHEMA	VARCHAR (128)		Schema name of the data type of the parameter or key column.
TYPENAME	VARCHAR (128)		Unqualified name of the data type of the parameter or key column.
LENGTH	INTEGER		Data type length of the parameter or key column.
SCALE	SMALLINT		Data type scale of the parameter or key column; 0 if not applicable.
PARMTYPE	CHAR (1)		• K = Source key column • P = Index extension instance parameter
CODEPAGE	SMALLINT		Code page of the index extension instance parameter; 0 if not a string type.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the parameter; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; the null value otherwise.

SYSCAT.INDEXEXTENSIONS

Each row represents an index extension.

Table 153. SYSCAT.INDEXEXTENSIONS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR (128)		Schema name of the index extension.
IENAME	VARCHAR (128)		Unqualified name of the index extension.
OWNER	VARCHAR (128)		Authorization ID of the owner of the index extension.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the index extension was defined.
KEYGENFUNCSCHEMA	VARCHAR (128)		Schema name of the key generation function.
KEYGENFUNCNAME	VARCHAR (128)		Unqualified name of the key generation function.
KEYGENSPECIFICNAME	VARCHAR (128)		Name of the key generation function instance (might be system-generated).
TEXT	CLOB (2M)		Full text of the CREATE INDEX EXTENSION statement.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the index extension.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.INDEXOPTIONS

Each row represents an index-specific option value.

Table 154. SYSCAT.INDEXOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
OPTION	VARCHAR (128)		Name of the index option.
SETTING	VARCHAR (2048)		Value of the index option.

SYSCAT.INDEXPARTITIONS

Each row represents a partitioned index piece located on one data partition. Note that the index partition statistics represent one database partition if the table is created on multiple database partitions.

Table 155. SYSCAT.INDEXPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the index.
INDNAME	VARCHAR (128)		Unqualified name of the index.
TABSCHEMA	VARCHAR (128)		Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)		Unqualified name of the table or nickname on which the index is defined.
IID	SMALLINT		Identifier for the index.
INDPARTITIONTBSPACEID	INTEGER		Identifier for the index partition table space.
INDPARTITIONOBJECTID	INTEGER		Identifier for the index partition object.
DATAPARTITIONID	INTEGER		This corresponds to the DATAPARTITIONID found in the SYSCAT.DATAPARTITIONS view.
INDCARD	BIGINT		Cardinality of the index partition. This might be different from the cardinality of the corresponding data partition for partitioned indexes that do not have a one-to-one relationship between the data partition rows and the index entries.
NLEAF	BIGINT		Number of leaf pages in the index partition; -1 if statistics are not collected.
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages in the index partition that have all of their row identifiers (RIDs) or block identifiers (BIDs) marked deleted.
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition; -1 if not known.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index partition that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
FULLKEYCARD	BIGINT		Number of distinct full-key values in the index partition; -1 if statistics are not collected.
NLEVELS	SMALLINT		Number of index levels in the index partition; -1 if statistics are not collected.
CLUSTERRATIO	SMALLINT		Degree of data clustering with the index partition; -1 in either of the following situations: • Statistics are not collected • Detailed index statistics are collected. In this situation, CLUSTERFACTOR will be used instead.

SYSCAT.INDEXPARTITIONS

Table 155. SYSCAT.INDEXPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CLUSTERFACTOR	DOUBLE		Finer measurement of the degree of clustering; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index key; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index key; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index key; -1 if statistics are not collected, or if not applicable.
AVGLEAFKEYSIZE	INTEGER		Average index key size for keys on leaf pages in the index partition; -1 if statistics are not collected.
AVGNLEAFKEYSIZE	INTEGER		Average index key size for keys on non-leaf pages in the index partition; -1 if statistics are not collected.
PCTFREE	SMALLINT		Percentage of each index page to be reserved during the initial building of the index partition. This space is available for data insertions after the index partition has been built.
PAGE_FETCH_PAIRS	VARCHAR (520)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the data partition with this index using that hypothetical buffer. Zero-length string if not data is available.
SEQUENTIAL_PAGES	BIGINT		Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index partition, expressed as a percent (integer between 0 and 100); -1 if statistics are not collected.
AVERAGE_SEQUENCE_GAP	DOUBLE		Gap between index page sequences within the index partition. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.

Table 155. SYSCAT.INDEXPARTITIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
AVERAGE_SEQUENCE_ FETCH_GAP	DOUBLE		Gap between table page sequences when fetching using the index partition. Detected through a scan of index leaf pages, each gap represents the average number of data partition pages that must be randomly fetched between sequences of data partition pages; -1 if not known.
AVERAGE_SEQUENCE_ PAGES	DOUBLE		Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.
AVERAGE_SEQUENCE_ FETCH_PAGES	DOUBLE		Average number of data partition pages that are accessible in sequence (that is, the number of data partition pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
AVERAGE_RANDOM_PAGES	DOUBLE		Average number of random data partition pages between sequential page accesses; -1 if not known.
AVERAGE_RANDOM_FETCH_ PAGES	DOUBLE		Average number of random data partition pages between sequential page accesses when fetching using the index partition; -1 if not known.
STATS_TIME	TIMESTAMP	Y	Last time that any change was made to the recorded statistics for this index partition. The null value if no statistics are available.
COMPRESSION	CHAR (1)		Specifies whether index compression is activated • N = Not activated • Y = Activated
PCTPAGESSAVED	SMALLINT		Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.

SYSCAT.INDEXXMLPATTERNS

Each row represents a pattern clause in an index over an XML column.

Table 156. SYSCAT.INDEXXMLPATTERNS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR (128)		Schema name of the logical index.
INDNAME	VARCHAR (128)		Unqualified name of the logical index.
PINDNAME	VARCHAR (128)		Unqualified name of the physical index.
PINDID	SMALLINT		Identifier for the physical index.
TYPEMODEL	CHAR (1)		• Q = SQL DATA TYPE (Ignore invalid values) • R = SQL DATA TYPE (Reject invalid values)
DATATYPE	VARCHAR (128)		Name of the data type.
HASHED	CHAR (1)		Indicates whether or not the value is hashed. • N = Not hashed • Y = Hashed
LENGTH	SMALLINT		Length if DATATYPE = 'VARCHAR' and HASHED = 'N'; precision if DATATYPE = 'DECIMAL'; 0 otherwise.
SCALE	SMALLINT		Scale if DATATYPE = 'DECIMAL'; 0 otherwise.
PATTERNID	SMALLINT		Identifier for the pattern.
PATTERN	CLOB (2M)	Y	Definition of the pattern.

Note:

1. When indexes over XML columns are created, logical indexes that use XML pattern information are created, resulting in the creation of physical B-tree indexes with key columns generated by the database manager to support the logical indexes. A physical index is created to support the data type that is specified in the xmltype-clause of the CREATE INDEX statement.

SYSCAT.INVALIDOBJECTS

Each row represents an invalid object.

Table 157. SYSCAT.INVALIDOBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object being created or revalidated.
OBJECTMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object being created or revalidated belongs. The null value if the object does not belong to a module.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object being created or revalidated. For routines (OBJECTTYPE = 'F'), this is the specific name.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
OBJECTTYPE	CHAR (1)		Type of the object being created or revalidated. Possible values are: • B = Trigger • F = Routine • R = User-defined data type • V = View • v = Global variable • y = Row permission • 2 = Column mask • 3 = Usage list
SQLCODE	INTEGER	Y	SQLCODE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.
SQLSTATE	CHAR (5)	Y	SQLSTATE returned in CREATE with errors or revalidation. The null value if the object has never been revalidated.
ERRORMESSAGE	VARCHAR (4000)	Y	Short text for the message associated with SQLCODE. The null value if the object has never been revalidated.
LINENUMBER	INTEGER	Y	Line number where the error occurred in compiled objects. The null value if the object is not a compiled object.
INVALIDATE_TIME	TIMESTAMP		Time at which the object was last invalidated.
LAST_REGEN_TIME	TIMESTAMP	Y	Time at which the object was last revalidated. The null value if the object has never been revalidated.

SYSCAT.KEYCOLUSE

Each row represents a column that participates in a key defined by a unique, primary key, or foreign key constraint.

Table 158. SYSCAT.KEYCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table containing the column.
TABNAME	VARCHAR (128)		Unqualified name of the table containing the column.
COLNAME	VARCHAR (128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key for the constraint (initial position is 1). If a constraint uses an existing index, this value is the numeric position of the column in the index.

SYSCAT.MEMBERSUBSETATTRS

Each row represents a member subset attribute.

Table 159. SYSCAT.MEMBERSUBSETATTRS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETID	INTEGER		Subset identifier.
ATTRID	SMALLINT		Attribute identifier. • 1 = DBALIAS
ATTRVALUE	VARCHAR(1000)	Y	Value of attribute.

SYSCAT.MEMBERSUBSETMEMBERS

Each row represents a member that is associated with a member subset.

Table 160. SYSCAT.MEMBERSUBSETMEMBERS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETID	INTEGER		Subset identifier.
MEMBER	SMALLINT		Member ID as defined in db2nodes.cfg.

SYSCAT.MEMBERSUBSETS

Each row represents a member subset.

Table 161. SYSCAT.MEMBERSUBSETS Catalog View

Column Name	Data Type	Nullable	Description
SUBSETNAME	VARCHAR(128)		Name of subset.
SUBSETID	INTEGER		Subset identifier.
CREATE_TIME	TIMESTAMP		Time at which subset was defined.
ALTER_TIME	TIMESTAMP		Time at which subset was last altered.
ENABLED	CHAR(1)		State of the member subset. • N = subset is currently disabled • Y = subset is currently enabled
MEMBERPRIORITYBASIS	CHAR(1)		Basis for member priorities: • E = Equal priorities • L = Member load
INCLUSIVESUBSET	CHAR(1)		Inclusiveness of member subset: • N = The member subset is not inclusive • Y = The member subset is inclusive
ALTERNATESERVER	CHAR(1)		Alternate server's inclusion in the member subsets server list: • N = The alternate server is not included in the member subsets server list • Y = The alternate server is included in the member subsets server list
CATALOGDATABASEALIAS	CHAR(1)		Database alias catalogued for member subset: • N = No database alias cataloged explicitly for this member subset • Y = Database alias cataloged explicitly for this member subset
REMARKS	VARCHAR(254)	Y	User provided comments, or the null value.

SYSCAT.MODULEAUTH

Each row represents a user, group, or role that has been granted a privilege on a module.

Table 162. SYSCAT.MODULEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
MODULEID	INTEGER		Identifier for the module to which this privilege applies.
MODULESCHEMA	VARCHAR (128)		Schema name of the module to which this privilege applies.
MODULENAME	VARCHAR (128)		Unqualified name of the module to which this privilege applies.
EXECUTEAUTH	CHAR (1)		Privilege to execute objects in the identified module. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.MODULEOBJECTS

Each row represents a function, procedure, global variable, condition, or user-defined type that belongs to a module.

Table 163. SYSCAT.MODULEOBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTSCHEMA	VARCHAR (128)	N	Schema name of the module.
OBJECTMODULENAME	VARCHAR (128)	N	Unqualified name of the module to which the object belongs.
OBJECTNAME	VARCHAR (128)	N	Unqualified name of the object.
OBJECTTYPE	VARCHAR (9)	N	• CONDITION = The object is a condition • FUNCTION = The object is a function • PROCEDURE = The object is a procedure • TYPE = The object is a data type • VARIABLE = The object is a variable
PUBLISHED	CHAR (1)	N	Indicates whether the object can be referenced outside its module. • N = The object is not published • Y = The object is published
SPECIFICNAME	VARCHAR (128)	N	Routine specific name if OBJECTTYPE is 'FUNCTION', 'METHOD' or 'PROCEDURE'; the null value otherwise.
USERDEFINED	CHAR (1)	N	Indicates whether the object is generated by the system or defined by a user. • N = The object is system generated • Y = The object is defined by a user

SYSCAT.MODULES

Each row represents a module.

Table 164. SYSCAT.MODULES Catalog View

Column Name	Data Type	Nullable	Description
MODULESCHEMA	VARCHAR (128)		Schema name of the module.
MODULENAME	VARCHAR (128)		Unqualified name of the module.
MODULEID	INTEGER		Identifier for the module.
DIALECT	VARCHAR (10)		The source dialect of the SQL module. Possible values are: • DB2 SQL PL • PL/SQL • Blank = Not applicable for an alias
OWNER	VARCHAR (128)		Authorization ID of the owner of the module.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
MODULETYPE	CHAR (1)		Type of module. • A = Alias • M = Module • P = PL/SQL package
BASE_MODULESCHEMA	VARCHAR (128)	Y	If MODULETYPE is 'A', contains the schema name of the module or alias that is referenced by this alias; the null value otherwise.
BASE_MODULENAME	VARCHAR (128)	Y	If MODULETYPE is 'A', contains the unqualified name of the module or alias that is referenced by this alias; the null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the module was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.NAMEMAPPINGS

Each row represents the mapping between a "logical" object (typed table or view and its columns and indexes, including inherited columns) and the corresponding "implementation" object (hierarchy table or hierarchy view and its columns and indexes) that implements the logical object.

Table 165. SYSCAT.NAMEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE	CHAR (1)		• C = Column • I = Index • U = Typed table
LOGICAL_SCHEMA	VARCHAR (128)		Schema name of the logical object.
LOGICAL_NAME	VARCHAR (128)		Unqualified name of the logical object.
LOGICAL_COLNAME	VARCHAR (128)	Y	Name of the logical column if TYPE = 'C'; null value otherwise.
IMPL_SCHEMA	VARCHAR (128)		Schema name of the implementation object that implements the logical object.
IMPL_NAME	VARCHAR (128)		Unqualified name of the implementation object that implements the logical object.
IMPL_COLNAME	VARCHAR (128)	Y	Name of the implementation column if TYPE = 'C'; null value otherwise.

SYSCAT.NICKNAMES

Each row represents a nickname.

Table 166. SYSCAT.NICKNAMES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the nickname.
TABNAME	VARCHAR (128)		Unqualified name of the nickname.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
STATUS	CHAR (1)		Status of the object. • C = Set integrity pending • N = Normal • X = Inoperative
CREATE_TIME	TIMESTAMP		Time at which the object was created.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected.
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).
TABLEID	SMALLINT		Internal logical object identifier.
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.
CARD	BIGINT		Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the nickname exist; -1 if statistics are not gathered.
FPAGES	BIGINT		Total number of pages; -1 if statistics are not gathered.
OVERFLOW	BIGINT		Total number of overflow records; -1 if statistics are not gathered.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.
SELFREFS	SMALLINT	Y	Number of self-referencing referential constraints for this object; that is, the number of referential constraints in which this object is both a parent and a dependent.
KEYCOLUMNS	SMALLINT	Y	Number of columns in the primary key.
KEYINDEXID	SMALLINT	Y	Index identifier for the primary key index; 0 or the null value if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique key constraints (other than the primary key constraint) defined on this object.

Table 166. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CHECKCOUNT	SMALLINT		Number of check constraints defined on this object.
DATACAPTURE	CHAR (1)		• L = Nickname participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns • N = Nickname does not participate in data replication • Y = Nickname participates in data replication
CONST_CHECKED	CHAR (32)		• Byte 1 represents foreign key constraint. • Byte 2 represents check constraint. • Byte 5 represents materialized query table. • Byte 6 represents generated column. • Byte 7 represents staging table. • Byte 8 represents data partitioning constraint. • Other bytes are reserved for future use. Possible values are: • F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table. • N = Not checked • U = Checked by user • W = Was in 'U' state when the table was placed in set integrity pending state • Y = Checked by system
PARTITION_MODE	CHAR (1)		Reserved for future use.
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
ACCESS_MODE	CHAR (1)		Access restriction state of the object. These states only apply to objects that are in set integrity pending state or to objects that were processed by a SET INTEGRITY statement. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access
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.

SYSCAT.NICKNAMES

Table 166. SYSCAT.NICKNAMES Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_SCHEMA	VARCHAR (128)	Y	Schema name of the specific data source object (such as a table or a view) for which the nickname was created.
SERVERNAME	VARCHAR (128)	Y	Name of the data source that contains the table or view for which the nickname was created.
REMOTE_TYPE	CHAR (1)	Y	Type of object at the data source. • A = Alias • N = Nickname • S = Materialized query table • T = Table (untyped) • V = View (untyped)
CACHINGALLOWED	VARCHAR (1)		• N = Caching is not allowed • Y = Caching is allowed
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.PACKAGEAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a package.

Table 167. SYSCAT.PACKAGEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held
BINDAUTH	CHAR (1)		Privilege to bind the package. • G = Held and grantable • N = Not held • Y = Held
EXECUTEAUTH	CHAR (1)		Privilege to execute the package. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.PACKAGEDEP

Each row represents a dependency of a package on some other object. The package depends on the object of type BTYPE of name BNAME, so a change to the object affects the package.

Table 168. SYSCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BINDER	VARCHAR (128)		Binder of the package.
BINDERTYPE	CHAR (1)		• U = Binder is an individual user
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • D = Server definition • F = Routine • G = Global temporary table • I = Index • M = Function mapping • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • P = Page size • Q = Sequence object • R = User-defined data type • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view • Z = XSR object • m = Module • n = Database partition group • q = Sequence alias • u = Module alias • v = Global variable • 4 = Application-period temporal table • 5 = System-period temporal table
BSCHEMA	VARCHAR (128)		Schema name of an object on which the package depends.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of an object on which the package depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

Table 168. SYSCAT.PACKAGEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
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).
VARAUTH	SMALLINT	Y	If BTYPE is 'v', encodes the privileges that are required by this package (READ or WRITE).
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
PKGVERSION	VARCHAR (64)	Y	Version identifier for the package.

Note:

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.

SYSCAT.PACKAGES

Each row represents a package that has been created by binding an application program.

Table 169. SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
BOUNDBY	VARCHAR (128)		Authorization ID of the binder and owner of the package.
BOUNDBYTYPE	CHAR (1)		• U = The binder and owner is an individual user
OWNER	VARCHAR (128)		Authorization ID of the binder and owner of the package.
OWNERTYPE	CHAR (1)		• U = The binder and owner is an individual user
DEFAULT_SCHEMA	VARCHAR (128)		Default schema name used for unqualified names in static SQL statements.
VALID[1]	CHAR (1)		• N = Needs rebinding • V = Validate at run time • X = Package is inoperative because some function instance on which it depends has been dropped; explicit rebind is needed • Y = Valid
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
TOTAL_SECT	SMALLINT		Number of sections in the package.
FORMAT	CHAR (1)		Date and time format associated with the package. • 0 = Format associated with the territory code of the client • 1 = USA • 2 = EUR • 3 = ISO • 4 = JIS • 5 = LOCAL
ISOLATION	CHAR (2)	Y	Isolation level. • CS = Cursor Stability • RR = Repeatable Read • RS = Read Stability • UR = Uncommitted Read

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CONCURRENTACCESSRESOLUTION	CHAR (1)	Y	The value of the CONCURRENTACCESSRESOLUTION bind option: • U = USE CURRENTLY COMMITTED • W = WAIT FOR OUTCOME • Blank = Not specified
BLOCKING	CHAR (1)	Y	Cursor blocking option. • B = Block all cursors • N = No blocking • U = Block unambiguous cursors
INSERT_BUF	CHAR (1)		Setting of the INSERT bind option (applies to partitioned database systems). • N = Inserts are not buffered • Y = Inserts are buffered at the coordinator member to minimize traffic among members
LANG_LEVEL	CHAR (1)	Y	Setting of the LANGLEVEL bind option. • 0 = SAA1 • 1 = MIA • 2 = SQL92E
FUNC_PATH	CLOB (2K)		SQL path in effect when the package was bound.
QUERYOPT	INTEGER		Optimization class under which this package was bound. Used for rebind operations.
EXPLAIN_LEVEL	CHAR (1)		Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option. • P = Package selection level • Blank = No Explain requested
EXPLAIN_MODE	CHAR (1)		Value of the EXPLAIN bind option. • A = ALL • N = No • R = REOPT • Y = Yes
EXPLAIN_SNAPSHOT	CHAR (1)		Value of the EXPLSNAP bind option. • A = ALL • N = No • R = REOPT • Y = Yes
SQLWARN	CHAR (1)		Indicates whether or not positive SQLCODEs resulting from dynamic SQL statements are returned to the application. • N = No, they are suppressed • Y = Yes

SYSCAT.PACKAGES

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SQLMATHWARN	CHAR (1)		Value of the *dft_sqlmathwarn* database configuration parameter at bind time. Indicates whether arithmetic and retrieval conversion errors return warnings and null values (indicator -2), allowing query processing to continue whenever possible. • N = No, errors are returned • Y = Yes, warnings are returned
CREATE_TIME	TIMESTAMP		Time at which the package was first bound.
EXPLICIT_BIND_TIME	TIMESTAMP		Time at which this package was last changed by: • BIND • REBIND (explicit)
LAST_BIND_TIME	TIMESTAMP		Time at which the package was last changed by: • BIND • REBIND (explicit) • REBIND (implicit)
ALTER_TIME	TIMESTAMP		Time at which this package was last changed by: • BIND • REBIND (explicit) • REBIND (implicit) • ALTER PACKAGE
CODEPAGE	SMALLINT		Application code page at bind time; -1 if not known.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the package.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the package.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the package.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the package.
DEGREE	CHAR (5)		Degree of intrapartition parallelism that was specified when the package was bound. • 1 = No parallelism • 2-32767 = User-specified limit • ANY = Degree determined by the system (no limit specified)
MULTINODE_PLANS	CHAR (1)		• N = Package was not bound in a partitioned database environment • Y = Package was bound in a partitioned database environment

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INTRA_PARALLEL	CHAR (1)		Use of intrapartition parallelism by static SQL statements within the package.
			• F = One or more static SQL statements in this package can use intrapartition parallelism; this parallelism has been disabled for use on a system that is not configured for intrapartition parallelism
			• N = No static SQL statement uses intrapartition parallelism
			• Y = One or more static SQL statements in the package use intrapartition parallelism
VALIDATE	CHAR (1)		Indicates whether validity checking can be deferred until run time.
			• B = All checking must be performed at bind time
			• R = Validation of tables, views, and privileges that do not exist at bind time is performed at run time

SYSCAT.PACKAGES

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DYNAMICRULES	CHAR (1)		• B = BIND; dynamic SQL statements are executed with DYNAMICRULES BIND behavior • D = DEFINERBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior • E = DEFINERRUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with DEFINE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior • H = INVOKEBIND; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with BIND behavior • I = INVOKERUN; when the package is run within a routine context, dynamic SQL statements in the package are executed with INVOKE behavior; when the package is not run within a routine context, dynamic SQL statements in the package are executed with RUN behavior • R = RUN; dynamic SQL statements are executed with RUN behavior; this is the default
SQLERROR	CHAR (1)		SQLERROR option on the most recent subcommand that bound or rebound the package. • C = CONTINUE; creates a package, even if errors occur while binding SQL statements • N = NOPACKAGE; does not create a package or a bind file if an error occurs
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.

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FEDERATED	CHAR (1)		• N = FEDERATED bind or prep option is turned off • U = FEDERATED bind or prep option was not specified • Y = FEDERATED bind or prep option is turned on
TRANSFORMGROUP	VARCHAR (1024)	Y	Value of the TRANSFORM GROUP bind option; the null value if a transform group is not specified.
REOPTVAR	CHAR (1)		Indicates whether the access path is determined again at execution time using input variable values. • A = Access path is reoptimized for every OPEN or EXECUTE request • N = Access path is determined at bind time • O = Access path is reoptimized only at the first OPEN or EXECUTE request; it is subsequently cached
OS_PTR_SIZE	INTEGER		Word size for the platform on which the package was created. • 32 = Package is a 32-bit package • 64 = Package is a 64-bit package
PKGVERSION	VARCHAR (64)		Version identifier for the package.
STATICREADONLY	CHAR (1)		Indicates whether or not static cursors will be treated as READ ONLY. Possible values are: • I = Any static cursor that does not contain the FOR UPDATE clause is considered READ ONLY and INSENSITIVE • N = Static cursors take on the attributes that would normally be generated for the given statement text and the setting of the LANGLEVEL precompile option • Y = Any static cursor that does not contain the FOR UPDATE or the FOR READ ONLY clause is considered READ ONLY
FEDERATED_ASYNCHRONY	INTEGER		Indicates the limit on asynchrony (the number of ATQs in the plan) as a bind option when the package was bound. • 0 = No asynchrony • n = User-specified limit (32 767 maximum) • -1 = Degree of asynchrony determined by the system • -2 = Degree of asynchrony not specified For a non-federated system, the value is 0.

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ANONBLOCK	CHAR (1)		• N = The package is not associated with an anonymous block • Y = The package is associated with an anonymous block
OPTPROFILESCHEMA	VARCHAR (128)	Y	Value of the optimization profile schema specified as part of the OPTPROFILE bind option.
OPTPROFILENAME	VARCHAR (128)	Y	Value of the optimization profile name specified as part of the OPTPROFILE bind option.
PKGID	BIGINT		Identifier for the package.
DBPARTITIONNUM	SMALLINT		Number of the database partition where the package was bound.
DEFINER[2]	VARCHAR (128)		Authorization ID of the binder and owner of the package.
PKG_CREATE_TIME[3]	TIMESTAMP		Time at which the package was first bound.
APREUSE	CHAR (1)		• N = The query compiler will not attempt to reuse access plans • Y = The access plans in this package should be reused, meaning that at rebind time the query compiler will attempt to choose plans like the ones currently in the package
EXTENDEDINDICATOR	CHAR (1)		• N = Extended indicator variable values are not recognized • Y = Extended indicator variable values are recognized
LASTUSED	DATE		Date when any statement in the package was last executed. This column is not updated for a package associated with an anonymous block. This column is not updated when a statement in the package is executed on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
BUSTIMESENSITIVE	CHAR (1)		• N = Statements that reference an application period temporal table (ATT) will not be affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register • Y = Statements that reference an ATT will be affected by the value of the CURRENT TEMPORAL BUSINESS_TIME special register

Table 169. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SYSTIMESENSITIVE	CHAR (1)		• N = Statements that reference a system period temporal table (STT) will not be affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register • Y = Statements that reference an STT will be affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register
KEEPDYNAMIC	CHAR (1)		Specifies whether dynamic SQL statements are kept after commit or rollback. • N = Inactive dynamic SQL statements need to be prepared again after commit or rollback • Y = Dynamic SQL statements are kept across transactions
STATICASDYNAMIC	CHAR (1)		• N = All static SQL statements in the package are compiled at bind time, using static SQL semantics • Y = All static SQL statements in the package are compiled at execution time, using dynamic SQL semantics
MEMBER	SMALLINT		Number of the member where the package was bound.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. If a function instance with dependencies is dropped, the package is put into an "inoperative" state, and it must be explicitly rebound. If any other object with dependencies is dropped, the package is put into an "invalid" state, and the system will attempt to rebind the package automatically when it is first referenced.

2. The DEFINER column is included for backwards compatibility. See OWNER.

3. The PKG_CREATE_TIME column is included for backwards compatibility. See CREATE_TIME.

SYSCAT.PARTITIONMAPS

Each row represents a distribution map that is used to distribute the rows of a table among the database partitions in a database partition group, based on hashing the table's distribution.

Table 170. SYSCAT.PARTITIONMAPS Catalog View

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT		Identifier for the distribution map.
PARTITIONMAP	BLOB (65536)		Distribution map, a vector of 32768 two-byte integers for a multiple partition database partition group. For a single partition database partition group, there is one entry denoting the partition number of the single partition.

SYSCAT.PASSTHRUAUTH

Each row represents a user, group, or role that has been granted pass-through authorization to query a data source.

Table 171. SYSCAT.PASSTHRUAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SERVERNAME	VARCHAR (128)		Name of the data source to which authorization is being granted.

SYSCAT.PERIODS

Each row represents the definition of a period for use with a temporal table.

Table 172. SYSCAT.PERIODS Catalog View

Column Name	Data Type	Nullable	Description
PERIODNAME	VARCHAR (128)		Name of the period.
TABSCHEMA	VARCHAR (128)		Schema name of the table.
TABNAME	VARCHAR (128)		Unqualified name of the table.
BEGINCOLNAME	VARCHAR (128)		Period begin column name.
ENDCOLNAME	VARCHAR (128)		Period end column name.
PERIODTYPE	CHAR (1)		Type of period. • A = Application period • S = System period
HISTORYTABSCHEMA	VARCHAR (128)		Schema name of the history table.
HISTORYTABNAME	VARCHAR (128)		Unqualified name of the history table.

SYSCAT.PREDICATESPECS

Each row represents a predicate specification.

Table 173. 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 174. SYSCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the dependent table.
TABNAME	VARCHAR (128)		Unqualified name of the dependent table.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
REFKEYNAME	VARCHAR (128)		Name of the parent key.
REFTABSCHEMA	VARCHAR (128)		Schema name of the parent table.
REFTABNAME	VARCHAR (128)		Unqualified name of the parent table.
COLCOUNT	SMALLINT		Number of columns in the foreign key.
DELETERULE	CHAR (1)		Delete rule. • A = NO ACTION • C = CASCADE • N = SET NULL • R = RESTRICT
UPDATERULE	CHAR (1)		Update rule. • A = NO ACTION • R = RESTRICT
CREATE_TIME	TIMESTAMP		Time at which the constraint was defined.
FK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.
PK_COLNAMES	VARCHAR (640)		This column is no longer used and will be removed in a future release. Use SYSCAT.KEYCOLUSE for this information.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the constraint.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.ROLEAUTH

Each row represents a role granted to a user, group, role, or PUBLIC.

Table 175. SYSCAT.ROLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted the role.
GRANTORTYPE	CHAR (1)		• U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Authorization ID to which the role was granted.
GRANTEETYPE	CHAR (1)		• G = The grantee is a group • R = The grantee is a role • U = The grantee is an individual user
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
ADMIN	CHAR (1)		Privilege to grant or revoke the role to or from others, or to comment on the role. • N = Not held • Y = Held

SYSCAT.ROLES

Each row represents a role.

Table 176. SYSCAT.ROLES Catalog View

Column Name	Data Type	Nullable	Description
ROLENAME	VARCHAR (128)		Name of the role.
ROLEID	INTEGER		Identifier for the role.
CREATE_TIME	TIMESTAMP		Time when the role was created.
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.ROUTINEAUTH

Each row represents a user, group, or role that has been granted EXECUTE privilege on either a particular routine (function, method, or procedure) in the database that is not defined in a module or all routines in a particular schema in the database that are not defined in a module.

Table 177. SYSCAT.ROUTINEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege. 'SYSIBM' if the privilege was granted by the system.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SCHEMA	VARCHAR (128)		Schema name of the routine.
SPECIFICNAME	VARCHAR (128)	Y	Specific name of the routine. If SPECIFICNAME is the null value and ROUTINETYPE is not 'M', the privilege applies to all routines of the type specified in ROUTINETYPE in the schema specified in SCHEMA. If SPECIFICNAME is the null value and ROUTINETYPE is 'M', the privilege applies to all methods for the subject type specified by TYPENAME in the schema specified by TYPESCHEMA. If SPECIFICNAME is the null value, ROUTINETYPE is 'M', and both TYPENAME and TYPESCHEMA are null values, the privilege applies to all methods for all types in the schema.
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the type for the method. The null value if ROUTINETYPE is not 'M'.
TYPENAME	VARCHAR (128)	Y	Unqualified name of the type for the method. The null value if ROUTINETYPE is not 'M'. If TYPENAME is the null value and ROUTINETYPE is 'M', the privilege applies to all methods for any subject type if they are in the schema specified by SCHEMA.
ROUTINETYPE	CHAR (1)		Type of the routine. • F = Function • M = Method • P = Procedure
EXECUTEAUTH	CHAR (1)		Privilege to execute the routine. • G = Held and grantable • N = Not held • Y = Held
GRANT_TIME	TIMESTAMP		Time at which the privilege was granted.

SYSCAT.ROUTINEDEP

Each row represents a dependency of a routine on some other object. The routine depends on the object of type BTYPE of name BNAME, so a change to the object affects the routine.

Table 178. SYSCAT.ROUTINEDEP Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine that has dependencies on another object.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module.
SPECIFICNAME	VARCHAR (128)		Specific name of the routine that has dependencies on another object.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module of the object that has dependencies on another object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.

Table 178. SYSCAT.ROUTINEDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.
TABAUTH	SMALLINT	Y	If BTYPE = 'O', 'S', 'T', 'U', 'V', 'W', or 'v', encodes the privileges on the table or view that are required by the dependent routine; the 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 179. SYSCAT.ROUTINEOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
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 180. SYSCAT.ROUTINEPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ORDINAL	SMALLINT		Position of the parameter within the routine signature.
OPTION	VARCHAR (128)		Name of the federated routine option.
SETTING	VARCHAR (2048)		Value of the federated routine option.

SYSCAT.ROUTINEPARMS

Each row represents a parameter or the result of a routine defined in SYSCAT.ROUTINES.

Table 181. SYSCAT.ROUTINEPARMS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)	Y	Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)	Y	Unqualified name of the routine.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module to which the routine belongs. The null value if not a module routine.
SPECIFICNAME	VARCHAR (128)	Y	Name of the routine instance (might be system-generated).
PARMNAME	VARCHAR (128)	Y	Name of the parameter or result column, or the null value if no name exists.
ROWTYPE	CHAR (1)		• B = Both input and output parameter • C = Result after casting • O = Output parameter • P = Input parameter • R = Result before casting
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 if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of the parameter or result belongs. The null value if not a module data type.
TYPENAME	VARCHAR (128)	Y	Unqualified name of the data type.
LOCATOR	CHAR (1)		• N = Paramater or result is not passed in the form of a locator • Y = Paramater or result is passed in the form of a locator
LENGTH[1]	INTEGER		Length of the parameter or result; 0 if the parameter or result is a user-defined data type.

Table 181. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SCALE[1]	SMALLINT		Scale if the parameter or result data type is DECIMAL; the number of digits of fractional seconds if the parameter or result data type is TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		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; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the parameter; the null value otherwise.
CAST_FUNCSCHEMA	VARCHAR (128)	Y	Schema name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.
CAST_FUNCSPECIFIC	VARCHAR (128)	Y	Unqualified name of the function used to cast an argument or a result. Applies to sourced and external functions; the null value otherwise.
TARGET_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the target type if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
TARGET_TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.
TARGET_TYPENAME	VARCHAR (128)	Y	Unqualified name of the module to which the target type belongs if the type of the parameter or result is REFERENCE. The null value if the type of the parameter or result is not REFERENCE or if the target type is not a module data type.
SCOPE_TABSCHEMA	VARCHAR (128)	Y	Schema name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.
SCOPE_TABNAME	VARCHAR (128)	Y	Unqualified name of the scope (target table) if the parameter type is REFERENCE; null value otherwise.
TRANSFORMGRPNAME	VARCHAR (128)	Y	Name of the transform group for a structured type parameter or result.

SYSCAT.ROUTINEPARMS

Table 181. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
DEFAULT	CLOB (64K)	Y	Expression used to calculate the default value of the parameter. The null value if DEFAULT clause was not specified for the parameter.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

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 182. SYSCAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
ROUTINETYPE	CHAR (1)		Type of routine. • F = Function • M = Method • P = Procedure
OWNER	VARCHAR (128)		Authorization ID of the owner of the routine.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER		Identifier for the routine.
ROUTINEMODULEID	INTEGER	Y	Identifier for the module to which the routine belongs. The null value if not a module routine.
RETURN_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the return type for a scalar function or method.
RETURN_TYPEMODULE	VARCHAR (128)	Y	The module name of the return type; the null value if the return type does not belong to any module.
RETURN_TYPENAME	VARCHAR (128)	Y	Unqualified name of the return type for a scalar function or method.
ORIGIN	CHAR (1)		• B = Built-in • E = User-defined, external • M = Template function • F = Federated procedure • Q = SQL-bodied[1] • R = System-generated SQL-bodied routine • S = System-generated • T = System-generated transform function (not directly invokable) • U = User-defined, based on a source

SYSCAT.ROUTINES

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FUNCTIONTYPE	CHAR (1)		• C = Column or aggregate • R = Row • S = Scalar • T = Table • Blank = Procedure
PARM_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', 'Q', or 'R'.
DIALECT	VARCHAR (10)		The source dialect of the SQL routine body: • DB2 SQL PL • PL/SQL • Blank = Not an SQL routine
SOURCESCHEMA	VARCHAR (128)	Y	If ORIGIN = 'U' and the source function is a user-defined function, contains the schema name of the specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'SYSIBM'. The null value if ORIGIN is not 'U'.
SOURCEMODULENAME	VARCHAR (128)	Y	Contains the module name of the specific name of the source function if ORIGIN = 'U' and the source function is a user-defined function defined in a module; the null value otherwise.
SOURCESPECIFIC	VARCHAR (128)	Y	If ORIGIN = 'U' and the source function is a user-defined function, contains the unqualified specific name of the source function. If ORIGIN = 'U' and the source function is a built-in function, contains the value 'N/A for built-in'. The null value if ORIGIN is not 'U'.
PUBLISHED	CHAR (1)		Inidicates whether the module routine can be invoked outside its module. • N = The module routine is not published • Y = The module routine is published • Blank = Not applicable
DETERMINISTIC	CHAR (1)		• N = Results are not deterministic (same parameters might give different results in different routine calls) • Y = Results are deterministic • Blank = ORIGIN is not 'E', 'F', 'Q', or 'R'

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXTERNAL_ACTION	CHAR (1)		• E = Function has external side-effects (therefore, the number of invocations is important) • N = No side-effects • Blank = ORIGIN is not 'E', 'F', 'Q', or 'R'
NULLCALL	CHAR (1)		• N = RETURNS NULL ON NULL INPUT (function result is implicitly the null value if one or more operands are null) • Y = CALLED ON NULL INPUT • Blank = ORIGIN is not 'E', 'Q', or 'R'
CAST_FUNCTION	CHAR (1)		• N = Not a cast function • Y = Cast function • Blank = ROUTINETYPE is not 'F'
ASSIGN_FUNCTION	CHAR (1)		• N = Not an assignment function • Y = Implicit assignment function • Blank = ROUTINETYPE is not 'F'
SCRATCHPAD	CHAR (1)		• N = Routine has no scratchpad • Y = Routine has a scratchpad • Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P'
SCRATCHPAD_LENGTH	SMALLINT		Size (in bytes) of the scratchpad for the routine. • -1 = LANGUAGE is 'OLEDB' and SCRATCHPAD is 'Y' • 0 = SCRATCHPAD is not 'Y'
FINALCALL	CHAR (1)		• N = No final call is made • Y = Final call is made to this routine at the runtime end-of-statement • Blank = ORIGIN is not 'E' or ROUTINETYPE is 'P'
PARALLEL	CHAR (1)		• N = Routine cannot be executed in parallel • Y = Routine can be executed in parallel • Blank = ORIGIN is not 'E'

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PARAMETER_STYLE	CHAR (8)		Parameter style that was declared when the routine was created. Possible values are: • DB2DARI • DB2GENRL • DB2SQL • GENERAL • GNRLNULL • JAVA • SQL • Blanks if ORIGIN is not 'E'
FENCED	CHAR (1)		• N = Not fenced • Y = Fenced • Blank = ORIGIN is not 'E'
SQL_DATA_ACCESS	CHAR (1)		Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine. • C = Contains SQL (simple expressions with no subqueries only) • M = Contains SQL statements that modify data • N = Does not contain SQL statements • R = Contains read-only SQL statements • Blank = ORIGIN is not 'E', 'F', 'Q', or 'R'
DBINFO	CHAR (1)		Indicates whether a DBINFO parameter is passed to an external routine. • N = DBINFO is not passed • Y = DBINFO is passed • Blank = ORIGIN is not 'E'
PROGRAMTYPE	CHAR (1)		Indicates how the external routine is invoked. • M = Main • S = Subroutine • Blank = ORIGIN is 'F'
COMMIT_ON_RETURN	CHAR (1)		Indicates whether the transaction is committed on successful return from this procedure. • N = The unit of work is not committed • Y = The unit of work is committed • Blank = ROUTINETYPE is not 'P'

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
AUTONOMOUS	CHAR (1)		Indicates whether or not the routine executes autonomously. • N = Routine does not execute autonomously from invoking transaction • Y = Routine executes autonomously from invoking transaction • Blank = ROUTINETYPE is not 'P'
RESULT_SETS	SMALLINT		Estimated maximum number of result sets.
SPEC_REG	CHAR (1)		Indicates the special registers values that are used when the routine is called. • I = Inherited special registers • Blank = PARAMETER_STYLE is 'DB2DARI' or ORIGIN is not 'E', 'Q', or 'R'
FEDERATED	CHAR (1)		Indicates whether or not federated objects can be accessed from the routine. • Y = Federated objects can be accessed • Blank = ORIGIN is not 'F'
THREADSAFE	CHAR (1)		Indicates whether or not the routine can run in the same process as other routines. • N = Routine is not threadsafe • Y = Routine is threadsafe • Blank = ORIGIN is not 'E'
VALID	CHAR (1)		Applies to LANGUAGE = 'SQL' and routines having parameters with default; blank otherwise. • N = Routine needs rebinding • X = Routine is inoperative and must be recreated • Y = Routine is valid
MODULEROUTINEIMPLEMENTED	CHAR (1)		• N = Module routine body is not implemented • Y = Module routine body is implemented • Blank = ROUTINEMODULENAME is null value
METHODIMPLEMENTED	CHAR (1)		• N = Method body is not implemented • Y = Method body is implemented • Blank = ROUTINETYPE is not 'M' or ROUTINEMODULENAME is not the null value
METHODEFFECT	CHAR (2)		• CN = Constructor method • MU = Mutator method • OB = Observer method • Blanks = Not a system method

SYSCAT.ROUTINES

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPE_PRESERVING	CHAR (1)		• N = Return type is the declared return type of the method • Y = Return type is governed by a "type-preserving" parameter; all system-generated mutator methods are type-preserving • Blank = ROUTINETYPE is not 'M'
WITH_FUNC_ACCESS	CHAR (1)		• N = This method cannot be invoked by using functional notation • Y = This method can be invoked by using functional notation; that is, the "WITH FUNCTION ACCESS" attribute is specified • Blank = ROUTINETYPE is not 'M'
OVERRIDDEN_METHODID	INTEGER	Y	Identifier for the overridden method when the OVERRIDING option is specified for a user-defined method. The null value if ROUTINETYPE is not 'M'.
SUBJECT_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the subject type for the user-defined method. The null value if ROUTINETYPE is not 'M'.
SUBJECT_TYPENAME	VARCHAR (128)	Y	Unqualified name of the subject type for the user-defined method. The null value if ROUTINETYPE is not 'M'.
CLASS	VARCHAR (384)	Y	For LANGUAGE JAVA, CLR, or OLE, this is the class that implements this routine; null value otherwise.
JAR_ID	VARCHAR (128)	Y	For LANGUAGE JAVA, this is the JAR_ID of the installed jar file that implements this routine if a jar file was specified at routine creation time; null value otherwise. For LANGUAGE CLR, this is the assembly file that implements this routine.
JARSCHEMA	VARCHAR (128)	Y	For LANGUAGE JAVA when a JAR_ID is present, this is the schema name of the jar file that implements this routine; null value otherwise.
JAR_SIGNATURE	VARCHAR (2048)	Y	For LANGUAGE JAVA, this is the method signature of this routine's specified Java method. For LANGUAGE CLR, this is a reference field for this CLR routine. Null value otherwise.
CREATE_TIME	TIMESTAMP		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 in effect when the routine was defined. The null value if LANGUAGE is not 'SQL' and no parameters have defaults.

Table 182. SYSCAT.ROUTINES 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.
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.
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[2]	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	Internal identifier for compiled SQL routines. Otherwise the null value.
TEXT_BODY_OFFSET	INTEGER		If LANGUAGE = 'SQL', the offset to the start of the compiled SQL routine body in the full text of the CREATE statement; -1 if LANGUAGE is not 'SQL' or the SQL routine is not compiled.

SYSCAT.ROUTINES

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
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. • N = A new savepoint level is not initiated when the routine is invoked; the routine uses the existing savepoint level • Y = A new savepoint level is initiated when the routine is invoked • Blank = Not applicable
DEBUG_MODE[3]	VARCHAR (8)		Indicates whether the routine can be debugged using the debugger that is integrated with the database. • DISALLOW = Routine is not debuggable • ALLOW = Routine is debuggable, and can participate in a client debug session with the integrated debugger • DISABLE = Routine is not debuggable, and this setting cannot be altered without dropping and recreating the routine • Blank = Routine type is not currently supported by the integrated debugger
TRACE_LEVEL	VARCHAR (1)	Y	Reserved for future use.
DIAGNOSTIC_LEVEL	VARCHAR (1)	Y	Reserved for future use.
CHECKOUT_USERID	VARCHAR (128)	Y	ID of the user who performed a checkout of the object; the null value if the object is not checked out.
PRECOMPILE_OPTIONS	VARCHAR (1024)	Y	The precompile and bind options that were in effect when the compiled SQL routine was created. The null value if LANGUAGE is not 'SQL' or if the SQL routine is not compiled.
COMPILE_OPTIONS	VARCHAR (1024)	Y	The value of the SQL_CCFLAGS special register that was in effect when the compiled SQL routine was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled SQL routine. The null value if LANGUAGE is not 'SQL' or if the SQL routine is not compiled.

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
EXECUTION_CONTROL	CHAR (1)		Execution control mode of a common language runtime (CLR) routine. Possible values are: • N = Network • R = Fileread • S = Safe • U = Unsafe • W = Filewrite • Blank = LANGUAGE is not 'CLR'
CODEPAGE	SMALLINT		Routine code page, which specifies the default code page used for all character parameter types, result types, and local variables within the routine body.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the routine.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the routine.
COLLATIONSCHEMA_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: • A = ASCII • U = UNICODE • Blank = PARAMETER CCSID clause was not specified
LAST_REGEN_TIME	TIMESTAMP		Time at which the SQL routine packed descriptor was last regenerated.
INHERITLOCKREQUEST	CHAR (1)		• N = This function or method cannot be invoked in the context of an SQL statement that includes a lock-request-clause as part of a specified isolation-clause • Y = This function or method inherits the isolation level of the invoking statement; it also inherits the specified lock-request-clause • Blank = LANGUAGE is not 'SQL' or ROUTINETYPE is 'P'
DEFINER[4]	VARCHAR (128)		Authorization ID of the owner of the routine.
SECURE	CHAR (1)		Indicates whether the function is secure for row and column access control • N = Not secure • Y = Secure • Blank = ROUTINETYPE is not 'F'

Table 182. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. For SQL procedures created before Version 8.2 and upgraded to Version 9, 'E' (instead of 'Q').

2. During database upgrade, the SELECTIVITY column will be set to -1 in the packed descriptor and system catalogs for all user-defined routines. For a user-defined predicate, the selectivity in the system catalog will be -1. In this case, the selectivity value used by the optimizer is 0.01.

3. 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.

4. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.ROUTINESFEDERATED

Each row represents a federated procedure.

Table 183. SYSCAT.ROUTINESFEDERATED Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
ROUTINENAME	VARCHAR (128)		Unqualified name of the routine.
ROUTINETYPE	CHAR (1)		Type of routine. • P = Procedure
OWNER	VARCHAR (128)		Authorization ID of the owner of the routine.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
ROUTINEID	INTEGER		Identifier for the routine.
PARM_COUNT	SMALLINT		Number of routine parameters.
DETERMINISTIC	CHAR (1)		• N = Results are not deterministic (same parameters might give different results in different routine calls) • Y = Results are deterministic
EXTERNAL_ACTION	CHAR (1)		• E = Routine has external side-effects (therefore, the number of invocations is important) • N = No side-effects
SQL_DATA_ACCESS	CHAR (1)		Indicates what type of SQL statements, if any, the database manager should assume is contained in the routine. • C = Contains SQL (simple expressions with no subqueries only) • M = Contains SQL statements that modify data • N = Does not contain SQL statements • R = Contains read-only SQL statements
COMMIT_ON_RETURN	CHAR (1)		Indicates whether the transaction is committed on successful return from this procedure. • N = The unit of work is not committed • Y = The unit of work is committed • Blank = ROUTINETYPE is not 'P'
RESULT_SETS	SMALLINT		Estimated maximum number of result sets.
CREATE_TIME	TIMESTAMP		Time at which the routine was created.
ALTER_TIME	TIMESTAMP		Time at which the routine was last altered.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.

SYSCAT.ROUTINESFEDERATED

Table 183. SYSCAT.ROUTINESFEDERATED Catalog View (continued)

Column Name	Data Type	Nullable	Description
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 the null value.

SYSCAT.ROWFIELDS

Each row represents a field that is defined for a user-defined row data type.

Table 184. SYSCAT.ROWFIELDS Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR (128)		Schema name of the row data type that includes the field.
TYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the row data type belongs. The null value if not a module row data type.
TYPENAME	VARCHAR (128)		Unqualified name of the row data type that includes the field.
FIELDNAME	VARCHAR (128)		Field name.
FIELDTYPESCHEMA	VARCHAR (128)		Schema name of the data type of the field.
FIELDTYPEMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the data type of the field belongs. The null value if the field data type is not a module user-defined data type.
FIELDTYPENAME	VARCHAR (128)		Unqualified name of the data type of the field.
ORDINAL	SMALLINT		Position of the field in the definition of the row data type, starting with 0.
LENGTH	INTEGER		Length of the field data type. For decimal types, contains the precision.
SCALE	SMALLINT		For decimal types, contains the scale of the field data type; for timestamp types, contains the timestamp precision of the field data type; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		For string types, denotes the code page; 0 indicates FOR BIT DATA; 0 for non-string types.
COLLATIONSCHEMA	VARCHAR (128)	Y	For string types, the schema name of the collation for the field; the null value otherwise.
COLLATIONNAME	VARCHAR (128)	Y	For string types, the unqualified name of the collation for the field; the null value otherwise.
NULLS	CHAR (1)		Reserved for future use.
QUALIFIER	VARCHAR (128)	Y	Reserved for future use.
FUNC_PATH	CLOB (2K)	Y	Reserved for future use.
DEFAULT	CLOB (64K)	Y	Reserved for future use.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

SYSCAT.SCHEMAAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a schema.

Table 185. SYSCAT.SCHEMAAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SCHEMANAME	VARCHAR (128)		Name of the schema to which this privilege applies.
ALTERINAUTH	CHAR (1)		Privilege to alter or comment on objects in the named schema. • G = Held and grantable • N = Not held • Y = Held
CREATEINAUTH	CHAR (1)		Privilege to create objects in the named schema. • G = Held and grantable • N = Not held • Y = Held
DROPINAUTH	CHAR (1)		Privilege to drop objects from the named schema. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.SCHEMATA

Each row represents a schema.

Table 186. SYSCAT.SCHEMATA Catalog View

Column Name	Data Type	Nullable	Description
SCHEMANAME	VARCHAR (128)		Name of the schema.
OWNER	VARCHAR (128)		Authorization ID of the owner of the schema.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
DEFINER	VARCHAR (128)		Authorization ID of the definer of the schema or authorization ID of the owner of the schema if the ownership of the schema has been transferred.
DEFINERTYPE	CHAR (1)		• S = The definer is the system • U = The definer is an individual user
CREATE_TIME	TIMESTAMP		Time at which the schema was created.
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)		Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
DATACAPTURE	CHAR (1)		Indicates the default data capture setting for new tables that are created within this schema. • N = New tables do not participate in data capture • Y = New tables participate in data capture, including replication of all columns
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SCPREFTBSPACES

Each row represents a preferred system temporary table space for the service class.

Table 187. SYSCAT.SCPREFTBSPACES Catalog View

Column Name	Data Type	Nullable	Description
SERVICECLASSNAME	VARCHAR (128)		Name of the service class.
PARENTSERVICECLASSNAME	VARCHAR (128)		Service class name of the parent service superclass.
TBSPACE	VARCHAR (128)		Name of the table space.
SERVICECLASSID	SMALLINT		Identifier for the service class.
PARENTSERVICECLASSID	SMALLINT		Identifier for the parent service class for the service class. 0 if the service class is a super service class.
TBSPACEID	INTEGER		Identifier for the table space.
DATATYPE	CHAR (1)		Type of data that can be stored in this table space. • A = All types of permanent data; regular table space • L = All types of permanent data; large table space • T = System temporary tables only • U = Created temporary tables or declared temporary tables only

SYSCAT.SECURITYLABELACCESS

Each row represents a security label that was granted to the database authorization ID.

Table 188. SYSCAT.SECURITYLABELACCESS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the security label.
GRANTEE	VARCHAR (128)		Holder of the security label.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SECLABELID	INTEGER		Identifier for the security label. For the name of the security label, select the SECLABELNAME column for the corresponding SECLABELID value in the SYSCAT.SECURITYLABELS catalog view.
SECPOLICYID	INTEGER		Identifier for the security policy that is associated with the security label. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSTYPE	CHAR (1)		• B = Both read and write access • R = Read access • W = Write access
GRANT_TIME	TIMESTAMP		Time at which the security label was granted.

SYSCAT.SECURITYLABELCOMPONENTELEMENTS

Each row represents an element value for a security label component.

Table 189. SYSCAT.SECURITYLABELCOMPONENTELEMENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPID	INTEGER		Identifier for the security label component.
ELEMENTVALUE	VARCHAR (32)		Element value for the security label component.
ELEMENTVALUEENCODING	CHAR (8) FOR BIT DATA		Encoded form of the element value.
PARENTELEMENTVALUE	VARCHAR (32)	Y	Name of the parent of an element for tree components; the null value for set and array components, and for the ROOT node of a tree component.

SYSCAT.SECURITYLABELCOMPONENTS

Each row represents a security label component.

Table 190. SYSCAT.SECURITYLABELCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
COMPNAME	VARCHAR (128)		Name of the security label component.
COMPID	INTEGER		Identifier for the security label component.
COMPTYPE	CHAR (1)		Security label component type. • A = Array • S = Set • T = Tree
NUMELEMENTS	INTEGER		Number of elements in the security label component.
CREATE_TIME	TIMESTAMP		Time at which the security label component was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SECURITYLABELS

Each row represents a security label.

Table 191. 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 the null value.

SYSCAT.SECURITYPOLICIES

Each row represents a security policy.

Table 192. SYSCAT.SECURITYPOLICIES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYNAME	VARCHAR (128)		Name of the security policy.
SECPOLICYID	INTEGER		Identifier for the security policy.
NUMSECLABELCOMP	INTEGER		Number of security label components in the security policy.
RWSECLABELREL	CHAR (1)		Relationship between the security labels for read and write access granted to the same authorization ID. • S = The security label for write access granted to a user is a subset of the security label for read access granted to that same user
NOTAUTHWRITESECLABEL	CHAR (1)		Action to take when a user is not authorized to write the security label that is specified in the INSERT or UPDATE statement. • O = Override • R = Restrict
CREATE_TIME	TIMESTAMP		Time at which the security policy was created.
GROUPAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a group will be used or ignored. • I = Ignored • U = Used
ROLEAUTHS	CHAR (1)		Indicates if authorizations of security labels and exemptions granted to an authorization ID that represents a role will be used or ignored. • I = Ignored • U = Used
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SECURITYPOLICYCOMPONENTRULES

Each row represents the read and write access rules for a security label component of the security policy.

Table 193. SYSCAT.SECURITYPOLICYCOMPONENTRULES Catalog View

Column Name	Data Type	Nullable	Description
SECPOLICYID	INTEGER		Identifier for the security policy.
COMPID	INTEGER		Identifier for the security label component of the security policy.
ORDINAL	INTEGER		Position of the security label component as it appears in the security policy, starting with 1.
READACCESSRULENAME	VARCHAR (128)		Name of the read access rule that is associated with the security label component.
READACCESSRULETEXT	VARCHAR (512)		Text of the read access rule that is associated with the security label component.
WRITEACCESSRULENAME	VARCHAR (128)		Name of the write access rule that is associated with the security label component.
WRITEACCESSRULETEXT	VARCHAR (512)		Text of the write access rule that is associated with the security label component.

SYSCAT.SECURITYPOLICYEXEMPTIONS

Each row represents a security policy exemption that was granted to a database authorization ID.

Table 194. SYSCAT.SECURITYPOLICYEXEMPTIONS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the exemption.
GRANTEE	VARCHAR (128)		Holder of the exemption.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SECPOLICYID	INTEGER		Identifier for the security policy for which the exemption was granted. For the name of the security policy, select the SECPOLICYNAME column for the corresponding SECPOLICYID value in the SYSCAT.SECURITYPOLICIES catalog view.
ACCESSRULENAME	VARCHAR (128)		Name of the access rule for which the exemption was granted.
ACCESSTYPE	CHAR (1)		Type of access to which the rule applies. • R = Read access • W = Write access
ORDINAL	INTEGER		Position of the security label component in the security policy to which the rule applies.
ACTIONALLOWED	CHAR (1)		If the rule is DB2LBACWRITEARRAY, then: • D = Write down • U = Write up Blank otherwise.
GRANT_TIME	TIMESTAMP		Time at which the exemption was granted.

SYSCAT.SEQUENCEAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a sequence.

Table 195. SYSCAT.SEQUENCEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of a privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of a privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
SEQSCHEMA	VARCHAR (128)		Schema name of the sequence.
SEQNAME	VARCHAR (128)		Unqualified name of the sequence.
ALTERAUTH	CHAR (1)		Privilege to alter the sequence. • G = Held and grantable • N = Not held • Y = Held
USAGEAUTH	CHAR (1)		Privilege to reference the sequence. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.SEQUENCES

Each row represents a sequence or alias.

Table 196. 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[1]	VARCHAR (128)		Authorization ID of the owner of the sequence.
DEFINERTYPE	CHAR (1)		• S = The definer is the system • U = The definer is an individual user
OWNER	VARCHAR (128)		Authorization ID of the owner of the sequence.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SEQID	INTEGER		Identifier for the sequence or alias.
SEQTYPE	CHAR (1)		Type of sequence. • A = Alias • I = Identity sequence • S = Sequence
BASE_SEQSCHEMA	VARCHAR (128)	Y	If SEQTYPE is 'A', contains the schema name of the sequence or alias that is referenced by this alias; the null value otherwise.
BASE_SEQNAME	VARCHAR (128)	Y	If SEQTYPE is 'A', contains the unqualified name of the sequence or alias that is referenced by this alias; the null value otherwise.
INCREMENT	DECIMAL (31,0)	Y	Increment value. The null value if the sequence is an alias.
START	DECIMAL (31,0)	Y	Start value of the sequence. The null value if the sequence is an alias.
MAXVALUE	DECIMAL (31,0)	Y	Maximum value of the sequence. The null value if the sequence is an alias.
MINVALUE	DECIMAL (31,0)	Y	Minimum value of the sequence. The null value if the sequence is an alias.
NEXTCACHEFIRSTVALUE	DECIMAL (31,0)	Y	The first value available to be assigned in the next cache block. If no caching, the next value available to be assigned.
CYCLE	CHAR (1)		Indicates whether or not the sequence can continue to generate values after reaching its maximum or minimum value. • N = Sequence cannot cycle • Y = Sequence can cycle • Blank = Sequence is an alias.

SYSCAT.SEQUENCES

Table 196. SYSCAT.SEQUENCES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CACHE	INTEGER		Number of sequence values to pre-allocate in memory for faster access. 0 indicates that values of the sequence are not to be preallocated. In a partitioned database, this value applies to each database partition. -1 if the sequence is an alias.
ORDER	CHAR (1)		Indicates whether or not the sequence numbers must be generated in order of request. • N = Sequence numbers are not required to be generated in order of request • Y = Sequence numbers must be generated in order of request • Blank = Sequence is an alias.
DATATYPEID	INTEGER		For built-in types, the internal identifier of the built-in type. For distinct types, the internal identifier of the distinct type. 0 if the sequence is an alias.
SOURCETYPEID	INTEGER		For a built-in type or if the sequence is an alias, this has a value of 0. For a distinct type, this is the internal identifier of the built-in type that is the source type for the distinct type.
CREATE_TIME	TIMESTAMP		Time at which the sequence was created.
ALTER_TIME	TIMESTAMP		Time at which the sequence was last altered.
PRECISION	SMALLINT		Precision of the data type of the sequence. Possible values are: • 5 = SMALLINT • 10 = INTEGER • 19 = BIGINT For DECIMAL, it is the precision of the specified DECIMAL data type. 0 if the sequence is an alias.
ORIGIN	CHAR (1)		Origin of the sequence. • S = System-generated sequence • U = User-generated sequence
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.SERVEROPTIONS

Each row represents a server-specific option value.

Table 197. SYSCAT.SERVEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)	Y	Name of the wrapper.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
CREATE_TIME	TIMESTAMP		Time at which the entry was created.
OPTION	VARCHAR (128)		Name of the server option.
SETTING	VARCHAR (2048)		Value of the server option.
SERVEROPTIONKEY	VARCHAR (18)		Uniquely identifies a row.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SERVERS

Each row represents a data source.

Table 198. SYSCAT.SERVERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
SERVERNAME	VARCHAR (128)		Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.SERVICECLASSES

Each row represents a service class.

Table 199. 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.
ENABLED	CHAR (1)		State of the service class. • N = Disabled • Y = Enabled
AGENTPRIORITY	SMALLINT		This column is no longer used and will be removed in a future release.
PREFETCHPRIORITY	CHAR (1)		Prefetch priority of the agents in the service class. • H = High • L = Low • M = Medium • Blank = not set
MAXDEGREE	SMALLINT	Y	Reserved for future use.
BUFFERPOOLPRIORITY	CHAR (1)		Bufferpool priority of the agents in the service class • H = High • L = Low • M = Medium • Blank = Not set
INBOUNDCORRELATOR	VARCHAR (128)	Y	For future use.
OUTBOUNDCORRELATOR	VARCHAR (128)	Y	String used to associate the service class with an operating system workload manager service class.
COLLECTAGGACTDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate activity data • E = Collect extended aggregate activity data • N = None
COLLECTAGGREQDATA	CHAR (1)		Specifies what aggregate request data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate request data • N = None

SYSCAT.SERVICECLASSES

Table 199. SYSCAT.SERVICECLASSES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members
COLLECTREQMETRICS	CHAR (1)		Specifies the monitoring level for requests submitted by a connection that is associated with the service superclass. • B = Collect base request metrics • E = Collect extended request metrics • N = None
CPUSHARES	INTEGER		The number of CPU shares allocated to this service class.
CPUSHARETYPE	CHAR (1)		Specifies the type of CPU shares. • S = Soft shares • H = Hard shares
CPULIMIT	SMALLINT		The maximum percentage of the CPU resource that can be allocated to the service class; -1 if there is no CPU limit.
SORTMEMORYPRIORITY	CHAR (1)		Reserved for future use.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for future use.
COLLECTAGGUOWDATA	CHAR (1)		Specifies what aggregate unit of work data should be captured for the service class by the applicable event monitor. • B = Collect base aggregate unit of work data • N = None
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.STATEMENTS

Each row represents an SQL statement in a package.

Table 200. SYSCAT.STATEMENTS Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR (128)		Schema name of the package.
PKGNAME	VARCHAR (128)		Unqualified name of the package.
STMTNO	INTEGER		Line number of the SQL statement in the source module of the application program.
SECTNO	SMALLINT		Number of the package section containing the SQL statement.
SEQNO	INTEGER		Always 1.
TEXT	CLOB (2M)		Text of the SQL statement.
UNIQUE_ID	CHAR (8) FOR BIT DATA		Identifier for a specific package when multiple packages having the same name exist.
VERSION	VARCHAR (64)	Y	Version identifier for the package.

SYSCAT.STOGROUPS

Each row represents a storage group object.

Table 201. SYSCAT.STOGROUPS Catalog View

Column Name	Data Type	Nullable	Description
SGNAME	VARCHAR (128)		Name of the storage group.
SGID	INTEGER		Identifier for the storage group.
OWNER	VARCHAR (128)		Authorization ID of the owner of the storage group.
CREATE_TIME	TIMESTAMP		Time at which the storage group was created.
DEFAULTSG	CHAR (1)		Indicates whether the storage group is the default storage group. • N = NO • Y = YES
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds (average for the storage paths in this storage group); 0 indicates value is not defined (assigned only by upgrade processing).
DEVICEREADRATE	DOUBLE		Read transfer rate of the device, in megabytes per second (average for the storage paths in this storage group); 0 indicates value is not defined (assigned only by upgrade processing).
WRITEOVERHEAD	DOUBLE	Y	Reserved for future use.
DEVICEWRITERATE	DOUBLE	Y	Reserved for future use.
DATATAG	SMALLINT		Tag to identify data stored in this storage group. Valid user-specified range is 1 through 9; 0 indicates no data tag is specified.
CACHINGTIER	SMALLINT		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.STATEMENTTEXTS

Each row represents a user-provided SQL statement for statement thresholds.

Table 202. SYSCAT.STATEMENTTEXTS Catalog View

Column Name	Data Type	Nullable	Description
TEXTID	INTEGER		Identifier for the SQL statement.
TEXT	CLOB (2M)		Text of the SQL statement.

SYSCAT.SURROGATEAUTHIDS

Each row represents a user or a group that has been granted SETSESSIONUSER privilege on a user or PUBLIC.

Table 203. SYSCAT.SURROGATEAUTHIDS Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Authorization ID that granted TRUSTEDID the ability to act as a surrogate. When the TRUSTEDID represents a trusted context object, this field represents the authorization ID that created or altered the trusted context object.
TRUSTEDID	VARCHAR (128)		Identifier for the entity that is trusted to act as a surrogate.
TRUSTEDIDTYPE	CHAR (1)		• C = Trusted context • G = Group • U = User
SURROGATEAUTHID	VARCHAR (128)		Surrogate authorization ID that can be assumed by TRUSTEDID. 'PUBLIC' indicates that TRUSTEDID can assume any authorization ID.
SURROGATEAUTHIDTYPE	CHAR (1)		• G = Group • U = User
AUTHENTICATE	CHAR (1)		• N = No authentication is required • Y = Authentication token is required with the authorization ID to authenticate the user before the authorization ID can be assumed • Blank = TRUSTEDIDTYPE is not 'C'
CONTEXTROLE	VARCHAR (128)	Y	A specific role to be assigned to the assumed authorization ID, which supercedes the default role, if any, that is defined for the trusted context. Null value when TRUSTEDIDTYPE is not 'C'.
GRANT_TIME	TIMESTAMP		Time at which the grant was made .

SYSCAT.TABAUTH

Each row represents a user, group, or role that has been granted one or more privileges on a table or view.

Table 204. SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view.
TABNAME	VARCHAR (128)		Unqualified name of the table or view.
CONTROLAUTH	CHAR (1)		CONTROL privilege. • N = Not held • Y = Held but not grantable
ALTERAUTH	CHAR (1)		Privilege to alter the table; allow a parent table to this table to drop its primary key or unique constraint; allow a table to become a materialized query table that references this table or view in the materialized query; or allow a table that references this table or view in its materialized query to no longer be a materialized query table. • G = Held and grantable • N = Not held • Y = Held
DELETEAUTH	CHAR (1)		Privilege to delete rows from a table or updatable view. • G = Held and grantable • N = Not held • Y = Held
INDEXAUTH	CHAR (1)		Privilege to create an index on a table. • G = Held and grantable • N = Not held • Y = Held
INSERTAUTH	CHAR (1)		Privilege to insert rows into a table or updatable view, or to run the import utility against a table or view. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.TABAUTH

Table 204. SYSCAT.TABAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFAUTH	CHAR (1)		Privilege to create and drop a foreign key referencing a table as the parent. • G = Held and grantable • N = Not held • Y = Held
SELECTAUTH	CHAR (1)		Privilege to retrieve rows from a table or view, create views on a table, or to run the export utility against a table or view. • G = Held and grantable • N = Not held • Y = Held
UPDATEAUTH	CHAR (1)		Privilege to run the UPDATE statement against a table or updatable view. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY. For table hierarchies, each constraint is recorded only at the level of the hierarchy where the constraint was created.

Table 205. SYSCAT.TABCONST Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR (128)		Name of the constraint.
TABSCHEMA	VARCHAR (128)		Schema name of the table to which this constraint applies.
TABNAME	VARCHAR (128)		Unqualified name of the table to which this constraint applies.
OWNER	VARCHAR (128)		Authorization ID of the owner of the constraint.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TYPE	CHAR (1)		Indicates the constraint type. • F = Foreign key • I = Functional dependency • K = Check • P = Primary key • U = Unique
ENFORCED	CHAR (1)		• N = Do not enforce constraint • Y = Enforce constraint
TRUSTED	CHAR (1)		If ENFORCED = 'N', specifies whether the data can be trusted to conform to the constraint. • N = Not trusted • Y = Trusted • Blank = Not applicable
CHECKEXISTINGDATA	CHAR (1)		• D = Defer checking any existing data • I = Immediately check existing data • N = Never check existing data
ENABLEQUERYOPT	CHAR (1)		• N = Query optimization is disabled • Y = Query optimization is enabled
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the constraint.
PERIODNAME	VARCHAR (128)	Y	Name of the period used to define this constraint.
PERIODPOLICY	CHAR (1)		If a period name was specified, the constraint uses this period policy. • N = Not applicable • O = Period overlaps not allowed
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABDEP

Each row represents a dependency of a view or a materialized query table on some other object.

The view or materialized query table depends on the object of type BTYPE of name BNAME, so a change to the object affects the view or materialized query table. Also encodes how privileges on views depend on privileges on underlying tables and views.

Table 206. 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. • S = Materialized query table • T = Table (staging only) • V = View (untyped) • W = Typed view • 7 = Synopsis table
OWNER	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.
OWNERTYPE	CHAR (1)		• U = The owner is an individual user
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • I = Index, if recording dependency on a base table • G = Global temporary table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • R = User-defined structured type • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view • Z = XSR object • m = Module • u = Module alias • v = Global variable
BSCHEMA	VARCHAR (128)		Schema name of the object on which the view or materialized query table depends.

Table 206. SYSCAT.TABDEP Catalog View (continued)

Column Name	Data Type	Nullable	Description
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which there is a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which the view or materialized query table depends.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which the view or materialized query table depends.
TABAUTH	SMALLINT	Y	If BTYPE is 'N', 'O', 'S', 'T', 'U', 'V', or 'W', encodes the privileges on the underlying table or view on which this view or materialized query table depends; the null value otherwise.
VARAUTH	SMALLINT	Y	If BTYPE is 'v', encodes the privileges on the underlying global variable on which this view or materialized query table depends; the null value otherwise.
DEFINER[1]	VARCHAR (128)		Authorization ID of the creator of the view or materialized query table.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABDETACHEDDEP

Each row represents a detached dependency between a detached dependent table and a detached table.

Table 207. 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 208. SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of the object.
TABNAME	VARCHAR (128)		Unqualified name of the object.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TYPE	CHAR (1)		Type of object. • A = Alias • G = Created temporary table • H = Hierarchy table • L = Detached table • N = Nickname • S = Materialized query table • T = Table (untyped) • U = Typed table • V = View (untyped) • W = Typed view
STATUS	CHAR (1)		Status of the object. • C = Set integrity pending • N = Normal • X = Inoperative
BASE_TABSCHEMA	VARCHAR (128)	Y	If TYPE = 'A', contains the schema name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
BASE_TABNAME	VARCHAR (128)	Y	If TYPE = 'A', contains the unqualified name of the table, view, alias, or nickname that is referenced by this alias; null value otherwise.
ROWTYPESCHEMA	VARCHAR (128)	Y	Schema name of the row type for this table, if applicable; null value otherwise.
ROWTYPENAME	VARCHAR (128)	Y	Unqualified name of the row type for this table, if applicable; null value otherwise.
CREATE_TIME	TIMESTAMP		Time at which the object was created.
ALTER_TIME	TIMESTAMP		Time at which the object was last altered.
INVALIDATE_TIME	TIMESTAMP		Time at which the object was last invalidated.
STATS_TIME	TIMESTAMP	Y	Time at which any change was last made to recorded statistics for this object. The null value if statistics are not collected.

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLCOUNT	SMALLINT		Number of columns, including inherited columns (if any).
TABLEID	SMALLINT		Internal logical object identifier.
TBSPACEID	SMALLINT		Internal logical identifier for the primary table space for this object.
CARD	BIGINT		Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
MPAGES	BIGINT		Total number of pages for table metadata. Non-zero only for a table that is organized by column; -1 for a view, an alias, or if statistics are not collected; -2 for subtables or hierarchy tables.
FPAGES	BIGINT		Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
OVERFLOW	BIGINT		Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
TBSPACE	VARCHAR (128)	Y	Name of the primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. The null value for aliases, views, and partitioned tables.
INDEX_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all indexes created on this table. The null value for aliases, views, and partitioned tables, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR (128)	Y	Name of the table space that holds all long data (LONG or LOB column types) for this table. The null value for aliases, views, and partitioned tables, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Y	Number of parent tables for this object; that is, the number of referential constraints in which this object is a dependent.
CHILDREN	SMALLINT	Y	Number of dependent tables for this object; that is, the number of referential constraints in which this object is a parent.

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
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.
DATACAPTURE	CHAR (1)		• L = Table participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns • N = Table does not participate in data replication • Y = Table participates in data replication, excluding replication of LONG VARCHAR and LONG VARGRAPHIC columns
CONST_CHECKED	CHAR (32)		• Byte 1 represents foreign key constraint. • Byte 2 represents check constraint. • Byte 5 represents materialized query table. • Byte 6 represents generated column. • Byte 7 represents staging table. • Byte 8 represents data partitioning constraint. • Other bytes are reserved for future use. Possible values are: • F = In byte 5, the materialized query table cannot be refreshed incrementally. In byte 7, the content of the staging table is incomplete and cannot be used for incremental refresh of the associated materialized query table. • N = Not checked • U = Checked by user • W = Was in 'U' state when the table was placed in set integrity pending state • Y = Checked by system

SYSCAT.TABLES

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT	Y	Identifier for the distribution map that is currently in use by this table (the null value for aliases or views).
PARTITION_MODE	CHAR (1)		Indicates how data is distributed among database partitions in a partitioned database system. • H = Hashing • R = Replicated across database partitions • Blank = No database partitioning
LOG_ATTRIBUTE	CHAR (1)		• Always 0. This column is no longer used.
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts.
APPEND_MODE	CHAR (1)		Controls how rows are inserted into pages. • N = New rows are inserted into existing spaces, if available • Y = New rows are appended to the end of the data
REFRESH	CHAR (1)		Refresh mode. • D = Deferred • I = Immediate • O = Once • Blank = Not a materialized query table
REFRESH_TIME	TIMESTAMP	Y	For REFRESH = 'D' or 'O', time at which the data was last refreshed (REFRESH TABLE statement); null value otherwise.
LOCKSIZE	CHAR (1)		Indicates the preferred lock granularity for tables that are accessed by data manipulation language (DML) statements. Applies to tables only. Possible values are: • I = Block insert • R = Row • T = Table • Blank = Not applicable
VOLATILE	CHAR (1)		• C = Cardinality of the table is volatile • Blank = Not applicable
ROW_FORMAT	CHAR (1)		Not used.

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
PROPERTY	VARCHAR (32)		Properties for a table. A single blank indicates that the table has no properties. The following is position within string, value, and meaning: • 1, Y = User maintained materialized query table • 2, Y = Staging table • 3, Y = Propagate immediate • 11, Y = Nickname that will not be cached • 13, Y = Statistical view • 19, Y = Statistical view for an index with an expression-based key • 20, Y = Column-organized table • 21, Y = Synopsis table • 23, Y = Shadow table (materialized query table maintained by replication)
STATISTICS_PROFILE	CLOB (10M)	Y	RUNSTATS command used to register a statistical profile for the object.
COMPRESSION	CHAR (1)		• B = Both value and row compression are enabled • N = No compression is enabled; a row format that does not support compression is used • R = Row compression is enabled; a row format that supports compression might be used • V = Value compression is enabled; a row format that supports compression is used • Blank = Not applicable
ROWCOMPMODE	CHAR (1)		Row compression mode for the table. • A = ADAPTIVE • S = STATIC • Blank = Row compression is not enabled
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 processed by a SET INTEGRITY statement. Possible values are: • D = No data movement • F = Full access • N = No access • R = Read-only access

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
CLUSTERED	CHAR (1)	Y	• T = Table is clustered by insert time • Y = Table is clustered by dimensions (even if only by one dimension) • Null value = Table is not clustered by dimensions or insert time
ACTIVE_BLOCKS	BIGINT		Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables or insert time clustering (ITC) tables only.
DROPRULE	CHAR (1)		• N = No rule • R = Restrict rule applies on drop
MAXFREESPACESEARCH	SMALLINT		Reserved for future use.
AVGCOMPRESSEDROWSIZE	SMALLINT		Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.
AVGROWSIZE	SMALLINT		Average length (in bytes) of both compressed and uncompressed rows in this table; -1 if statistics are not collected.
PCTROWSCOMPRESSED	REAL		Compressed rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
LOGINDEXBUILD	VARCHAR (3)	Y	Level of logging that is to be performed during create, re-create, or reorganize index operations on the table. • OFF = Index build operations on the table will be logged minimally • ON = Index build operations on the table will be logged completely • Null value = Value of the *logindexbuild* database configuration parameter will be used to determine whether or not index build operations are to be completely logged
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.
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.

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the table.
ENCODING_SCHEME	CHAR (1)		• A = CCSID ASCII was specified • U = CCSID UNICODE was specified • Blank = CCSID clause was not specified
PCTPAGESSAVED	SMALLINT		The approximate percentage of pages saved in a row-organized table as a result of row compression. For a column-organized table, the estimate is based on the number of data pages needed to store the table in uncompressed row organization.-1 if statistics are not collected.
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)		• B = Both column- and row-level granularity • C = Column-level granularity • R = Row-level granularity • Blank = Non-protected table
AUDITPOLICYID	INTEGER	Y	Identifier for the audit policy.
AUDITPOLICYNAME	VARCHAR (128)	Y	Name of the audit policy.
AUDITEXCEPTIONENABLED	CHAR (1)		Reserved for future use.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the table, view, alias, or nickname.
ONCOMMIT	CHAR (1)		Specifies the action taken on the created temporary table when a COMMIT operation is performed. • D = Delete rows • P = Preserve rows • Blank = Table is not a created temporary table
LOGGED	CHAR (1)		Specifies whether the created temporary table is logged. • N = Not logged • Y = Logged • Blank = Table is not a created temporary table

Table 208. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
ONROLLBACK	CHAR (1)		Specifies the action taken on the created temporary table when a ROLLBACK operation is performed. • D = Delete rows • P = Preserve rows • Blank = Table is not a created temporary table
LASTUSED	DATE		Date when the table was last used by any DML statement or the LOAD command. This column is not updated for an alias, created temporary table, nickname, or view. This column is not updated when the table is used on an HADR standby database. The default value is '0001-01-01'. This value is updated asynchronously not more than once within a 24 hour period and might not reflect usage within the last 15 minutes.
CONTROL	CHAR (1)		Access control that is enforced for the table • B = Both row and column • C = Column • R = Row • Blank = No access control
TEMPORALTYPE	CHAR (1)		Type of temporal table. • A = Application-period temporal table • B = Bitemporal table • N = Not a temporal table • S = System-period temporal table
TABLEORG	CHAR(1)		• C = Column-organized table • R = Row-organized table
EXTENDED_ROW_SIZE	CHAR(1)		Indicates whether the row size of a table that is organized by row exceeds the maximum record length for the page size of the table space in which it is defined. • N = Row size does not exceed the maximum record length for the page size • Y = Row size exceeds the maximum record length for the page size • blank = Not applicable
PCTEXTENDEDROWS	REAL		Extended rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABLESPACES

Each row represents a table space.

Table 209. SYSCAT.TABLESPACES Catalog View

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR (128)		Name of the table space.
OWNER	VARCHAR (128)		Authorization ID of the owner of the table space.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the table space was created.
TBSPACEID	INTEGER		Identifier for the table space.
TBSPACETYPE	CHAR (1)		Type of table space. • D = Database-managed space • S = System-managed space
DATATYPE	CHAR (1)		Type of data that can be stored in this table space. • A = All types of permanent data; regular table space • L = All types of permanent data; large table space • T = System temporary tables only • U = Created temporary tables or declared temporary tables only
EXTENTSIZE	INTEGER		Size of each extent, in pages of size PAGESIZE. This many pages are written to one container in the table space before switching to the next container.
PREFETCHSIZE	INTEGER		Number of pages of size PAGESIZE to be read when prefetching is performed; -1 when AUTOMATIC.
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds (average for the containers in this table space); -1 when value is inherited from the storage group that the table space uses.
TRANSFERRATE	DOUBLE		Time to read one page of size PAGESIZE into the buffer (average for the containers in this table space); -1 when value is inherited from the storage group that the table space uses.
WRITEOVERHEAD	DOUBLE	Y	Reserved for future use.
WRITETRANSFERRATE	DOUBLE	Y	Reserved for future use.
PAGESIZE	INTEGER		Size (in bytes) of pages in this table space.
DBPGNAME	VARCHAR (128)		Name of the database partition group that is associated with this table space.
BUFFERPOOLID	INTEGER		Identifier for the buffer pool that is used by this table space (1 indicates the default buffer pool).

SYSCAT.TABLESPACES

Table 209. SYSCAT.TABLESPACES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DROP_RECOVERY	CHAR (1)		Indicates whether or not tables in this table space can be recovered after a drop table operation. • N = Tables are not recoverable • Y = Tables are recoverable
NGNAME[1]	VARCHAR (128)		Name of the database partition group that is associated with this table space.
DEFINER[2]	VARCHAR (128)		Authorization ID of the owner of the table space.
DATATAG	SMALLINT		A tag to identify data stored in this table space. Valid user-specified range is 1 through 9; 0 indicates no data tag specified; -1 indicates value is inherited from the storage group that the table space uses.
SGNAME	VARCHAR (128)	Y	Name of the storage group the table space is using; null value when the table space is not using automatic storage.
SGID	INTEGER		Identifier of the storage group the table space is using; -1 when the table space is not using automatic storage.
EFFECTIVEPREFETCHSIZE	INTEGER		Value of the effective prefetch size when PREFETCHSIZE is set to -1 (AUTOMATIC); otherwise same as PREFETCHSIZE.
CACHINGTIER	SMALLINT		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The NGNAME column is included for backwards compatibility. See DBPGNAME.
2. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.TABOPTIONS

Each row represents an option that is associated with a remote table.

Table 210. SYSCAT.TABOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR (128)		Schema name of a table, view, alias, or nickname.
TABNAME	VARCHAR (128)		Unqualified name of a table, view, alias, or nickname.
OPTION	VARCHAR (128)		Name of the table option.
SETTING	CLOB (32K)		Value of the table option.

SYSCAT.TBSPACEAUTH

Each row represents a user, group, or role that has been granted the USE privilege on a particular table space in the database.

Table 211. SYSCAT.TBSPACEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
TBSPACE	VARCHAR (128)		Name of the table space.
USEAUTH	CHAR (1)		Privilege to create tables within the table space. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.THRESHOLDS

Each row represents a threshold.

Table 212. SYSCAT.THRESHOLDS Catalog View

Column Name	Data Type	Nullable	Description
THRESHOLDNAME	VARCHAR (128)		Name of the threshold.
THRESHOLDID	INTEGER		Identifier for the threshold.
ORIGIN	CHAR (1)		Origin of the threshold. • U = Threshold was created by a user • W = Threshold was created through a work action set
THRESHOLDCLASS	CHAR (1)		Classification of the threshold. • A = Aggregate threshold • C = Activity threshold
THRESHOLDPREDICATE	VARCHAR (15)		Type of the threshold. Possible values are: • AGGTEMPSPACE • CONCDBC • CONCWCN • CONCWOC • CONNIDLETIME • CPUTIME • CPUTIMEINSC • DATATAGINSC • DATATAGNOTINSC • DBCONN • ESTSQLCOST • ROWSREAD • ROWSREADINSC • ROWSRET • SCCONN • TEMPSPACE • TOTALTIME • UOWTOTALTIME
THRESHOLDPREDICATEID	SMALLINT		Identifier for the threshold predicate.
DOMAIN	CHAR (2)		Domain of the threshold. • DB = Database • SB = Service subclass • SP = Service superclass • WA = Work action set • WD = Workload definition • SQ = SQL statement
DOMAINID	INTEGER		Identifier for the object with which the threshold is associated. This can be a service class, work action, workload unique ID, or SQL statement. If this is a database threshold, this value is 0.

SYSCAT.THRESHOLDS

Table 212. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ENFORCEMENT	CHAR (1)		Scope of enforcement for the threshold. • D = Database • P = Member • W = Workload occurrence
QUEUING	CHAR (1)		• N = The threshold is not queueing • Y = The threshold is queueing
MAXVALUE	BIGINT		Upper bound specified by the threshold. If THRESHOLDPREDICATE is 'DATATAGINSC' or 'DATATAGNOTINSC', this value encodes one or more data tags.
DATATAGLIST	VARCHAR (256)	Y	If THRESHOLDPREDICATE is 'DATATAGINSC' or 'DATATAGNOTINSC', this value represents one or more data tags as a comma separated list. Otherwise, the null value.
QUEUESIZE	INTEGER		If QUEUEING is 'Y', the size of the queue. -1 otherwise.
OVERFLOWPERCENT	SMALLINT		Reserved for future use.
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members
EXECUTION	CHAR (1)		Indicates the execution action taken after a threshold has been exceeded. • C = Execution continues • F = Application is forced off the system • R = Execution is remapped to a different service subclass • S = Execution stops
REMAPSCID	SMALLINT		Target service subclass ID of the REMAP ACTIVITY action.
VIOLATIONRECORDLOGGED	CHAR (1)		Indicates whether a record is written to the event monitor upon threshold violation. • N = No • Y = Yes

Table 212. SYSCAT.THRESHOLDS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CHECKINTERVAL	INTEGER		The interval, in seconds, in which the threshold condition is checked if THRESHOLDPREDICATE is: • 'CPUTIME' • 'CPUTIMEINSC' • 'ROWSREAD' • 'ROWSREADINSC' Otherwise, -1.
ENABLED	CHAR (1)		• N = This threshold is disabled. • Y = This threshold is enabled.
CREATE_TIME	TIMESTAMP		Time at which the threshold was created.
ALTER_TIME	TIMESTAMP		Time at which the threshold was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.TRANSFORMS

Each row represents the functions that handle transformations between a user-defined type and a base SQL type, or the reverse.

Table 213. SYSCAT.TRANSFORMS Catalog View

Column Name	Data Type	Nullable	Description
TYPEID	SMALLINT		Identifier for the data type.
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
GROUPNAME	VARCHAR (128)		Name of the transform group.
FUNCID	INTEGER		Identifier for the routine.
FUNCSCHEMA	VARCHAR (128)		Schema name of the routine if ROUTINEMODULEID is null; otherwise schema name of the module to which the routine belongs.
FUNCNAME	VARCHAR (128)		Unqualified name of the routine.
SPECIFICNAME	VARCHAR (128)		Name of the routine instance (might be system-generated).
TRANSFORMTYPE	VARCHAR (8)		• 'FROM SQL' = Transform function transforms a structured type from SQL • 'TO SQL' = Transform function transforms a structured type to SQL
FORMAT	CHAR (1)		Format produced by the FROM SQL transform. • S = Structured data type • U = User-defined
MAXLENGTH	INTEGER	Y	Maximum length (in bytes) of output from the FROM SQL transform; the null value for TO SQL transforms.
ORIGIN	CHAR (1)		Source of this group of transforms. • O = Original transform group • R = Redefined transform group
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.TRIGDEP

Each row represents a dependency of a trigger on some other object. The trigger depends on the object of type BTYPE of name BNAME, so a change to the object affects the trigger.

Table 214. 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: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

SYSCAT.TRIGDEP

Table 214. SYSCAT.TRIGDEP 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.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 215. SYSCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR (128)		Schema name of the trigger.
TRIGNAME	VARCHAR (128)		Unqualified name of the trigger.
OWNER	VARCHAR (128)		Authorization ID of the owner of the trigger.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
TABSCHEMA	VARCHAR (128)		Schema name of the table or view to which this trigger applies.
TABNAME	VARCHAR (128)		Unqualified name of the table or view to which this trigger applies.
TRIGTIME	CHAR (1)		Time at which triggered actions are applied to the base table, relative to the event that fired the trigger. • A = Trigger is applied after the event • B = Trigger is applied before the event • I = Trigger is applied instead of the event
TRIGEVENT	CHAR (1)		Event that fires the trigger. • D = Delete event • I = Insert event • M = Multiple events • U = Update event
EVENTUPDATE	CHAR (1)		Indicates whether an update event fires the trigger. • N = No • Y = Yes
EVENTDELETE	CHAR (1)		Indicates whether a delete event fires the trigger. • N = No • Y = Yes
EVENTINSERT	CHAR (1)		Indicates whether an insert event fires the trigger. • N = No • Y = Yes
GRANULARITY	CHAR (1)		Trigger is executed once per: • R = Row • S = Statement
VALID	CHAR (1)		• N = Trigger is invalid • X = Trigger is inoperative and must be re-created • Y = Trigger is valid

SYSCAT.TRIGGERS

Table 215. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the trigger was defined.
TEXT	CLOB (2M)		Full text of the CREATE TRIGGER statement, exactly as typed.
LAST_REGEN_TIME	TIMESTAMP		Time at which the packed descriptor for the trigger was last regenerated.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the trigger.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the trigger.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the trigger.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the trigger.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the trigger.
SECURE	CHAR (1)		Indicates whether the trigger is secure for row and column access control • N = Not secure • Y = Secure
ALTER_TIME	TIMESTAMP		Time at which the trigger was last altered.
DEBUG_MODE	VARCHAR (8)		Indicates whether the trigger can be debugged using the DB2 debugger. • DISALLOW = Trigger is not debuggable • ALLOW = Trigger is debuggable, and can participate in a client debug session with the DB2 debugger • DISABLE = Trigger is not debuggable, and this setting cannot be altered without dropping and re-creating the trigger • Blank = Trigger type is not currently supported by the DB2 debugger
ENABLED	CHAR (1)		Reserved for future use.
LIB_ID	INTEGER	Y	Internal identifier for compiled SQL triggers. Otherwise, the null value.
PRECOMPILE_OPTIONS	VARCHAR(1024)	Y	The precompile and bind options that were in effect when the compiled trigger was created. The null value if the trigger is not compiled.

Table 215. SYSCAT.TRIGGERS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COMPILE_OPTIONS	VARCHAR(1024)	Y	The value of the SQL_CCFLAGS special register that was in effect when the compiled trigger was created and inquiry directives were present. An empty string if no inquiry directives were present in the compiled trigger. The null value if the trigger is not compiled.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for compatibility with earlier versions. See OWNER.

SYSCAT.TYPEMAPPINGS

Each row represents a data type mapping between a locally-defined data type and a data source data type. There are two mapping types (mapping directions):

forward type 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 216. 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. • F = Forward type mapping • R = Reverse type mapping
TYPESCHEMA	VARCHAR (128)	Y	Schema name of the local type in a data type mapping; the null value for built-in types.
TYPENAME	VARCHAR (128)		Unqualified name of the local type in a data type mapping.
TYPEID	SMALLINT		Identifier for the data type.
SOURCETYPEID	SMALLINT		Identifier for the source type.
OWNER	VARCHAR (128)		Authorization ID of the owner of the type mapping. 'SYSIBM' indicates a built-in type mapping.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
LENGTH	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.
SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.
LOWER_LEN	INTEGER	Y	Minimum length or precision of the local data type in this mapping. If the null value, the system determines the minimum length or precision. For character types, represents the minimum number of bytes.
UPPER_LEN	INTEGER	Y	Maximum length or precision of the local data type in this mapping. If the null value, the system determines the maximum length or precision. For character types, represents the maximum number of bytes.

Table 216. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a local decimal value or the minimum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the minimum number.
UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a local decimal value or the maximum number of digits of fractional seconds of a local TIMESTAMP value in this mapping. If the null value, the system determines the maximum number.
S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a local decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
BIT_DATA	CHAR (1)	Y	Indicates whether or not this character type is for bit data. Possible values are: • N = This type is not for bit data • Y = This type is for bit data • Null value = This is not a character data type, or the system determines the bit data attribute
WRAPNAME	VARCHAR (128)	Y	Data access protocol (wrapper) to which this mapping applies.
SERVERNAME	VARCHAR (128)	Y	Uppercase name of the server.
SERVERTYPE	VARCHAR (30)	Y	Type of server.
SERVERVERSION	VARCHAR (18)	Y	Server version.
REMOTE_TYPESCHEMA	VARCHAR (128)	Y	Schema name of the data source data type.
REMOTE_TYPENAME	VARCHAR (128)		Unqualified name of the data source data type.
REMOTE_META_TYPE	CHAR (1)	Y	Indicates whether this remote type is a system built-in type or a distinct type. • S = System built-in type • T = Distinct type
REMOTE_LOWER_LEN	INTEGER	Y	Minimum length or precision of the remote data type in this mapping, or the null value. For character types, represents the minimum number of characters (not bytes). For binary types, represents the minimum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.

SYSCAT.TYPEMAPPINGS

Table 216. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_UPPER_LEN	INTEGER	Y	Maximum length or precision of the remote data type in this mapping, or the null value. For character types, represents the maximum number of characters (not bytes). For binary types, represents the maximum number of bytes. A value of -1 indicates that the default length or precision is used, or that the remote type does not have a length or precision.
REMOTE_LOWER_SCALE	SMALLINT	Y	Minimum number of digits in the fractional part of a remote decimal value or the minimum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.
REMOTE_UPPER_SCALE	SMALLINT	Y	Maximum number of digits in the fractional part of a remote decimal value or the maximum number of digits of fractional seconds of a remote TIMESTAMP value in this mapping, or the null value.
REMOTE_S_OPR_P	CHAR (2)	Y	Relationship between the scale and precision of a remote decimal value in this mapping. Basic comparison operators (=, <, >, <=, >=, <>) can be used. A null value indicates that no specific relationship is required.
REMOTE_BIT_DATA	CHAR (1)	Y	Indicates whether or not this remote character type is for bit data. Possible values are: • N = This type is not for bit data • Y = This type is for bit data • Null value = This is not a character data type, or the system determines the bit data attribute
USER_DEFINED	CHAR (1)		Indicates whether or not the mapping is user-defined. The value is always 'Y'; that is, the mapping is always user-defined.
CREATE_TIME	TIMESTAMP		Time at which this mapping was created.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the type mapping. 'SYSIBM' indicates a built-in type mapping.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.USAGELISTS

Each row represents a usage list for a table or index object.

Table 217. SYSCAT.USAGELISTS Catalog View

Column Name	Data Type	Nullable	Description
USAGELISTSCHEMA	VARCHAR (128)		Schema of the usage list.
USAGELISTNAME	VARCHAR (128)		Name of the usage list.
USAGELISTID	INTEGER		Identifier for the usage list.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the object for which the usage list is defined.
OBJECTNAME	VARCHAR (128)		Unqualified name of the object for which the usage list is defined.
OBJECTTYPE	CHAR (1)		The type of the object for which this usage list is defined. • I = Index • T = Table
STATUS	CHAR (1)		The status of the usage list. • I = Invalid • V = Valid
MAXLISTSIZE	INTEGER		The maximum number of entries in the usage list.
WHENFULL	CHAR (1)		Action to be performed when the usage list is full. • D = Deactivate collection • W = Wrap
AUTOSTART	CHAR (1)		Indicates whether this usage list is to be activated automatically when the database starts. • N = Manual start • Y = Autostart
ACTIVEDURATION	INTEGER		Reserved for future use.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.USEROPTIONS

Each row represents a server-specific user option value.

Table 218. SYSCAT.USEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
AUTHID	VARCHAR (128)		Local authorization ID, in uppercase characters.
AUTHIDTYPE	CHAR (1)		• U = Grantee is an individual user
SERVERNAME	VARCHAR (128)		Name of the server on which the user is defined.
OPTION	VARCHAR (128)		Name of the user option.
SETTING	VARCHAR (2048)		Value of the user option.

SYSCAT.VARIABLEAUTH

Each row represents a user, group, or role that has been granted one or more privileges by a specific grantor on a global variable in the database that is not defined in a module.

Table 219. SYSCAT.VARIABLEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
VARSCHEMA	VARCHAR (128)		Schema name of the global variable if VARMODULEID is null; otherwise schema name of the module to which the global variable belongs.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARID	INTEGER		Identifier for the global variable.
READAUTH	CHAR (1)		Privilege to read the global variable. • G = Held and grantable • N = Not held • Y = Held
WRITEAUTH	CHAR (1)		Privilege to write the global variable. • G = Held and grantable • N = Not held • Y = Held

SYSCAT.VARIABLEDEP

Each row represents a dependency of a global variable on some other object. The global variable depends on the object of type BTYPE of name BNAME, so a change to the object affects the global variable.

Table 220. SYSCAT.VARIABLEDEP Catalog View

Column Name	Data Type	Nullable	Description
VARSCHEMA	VARCHAR (128)		Schema name of the global variable that has dependencies on another object.
VARMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the global variable belongs. The null value if not a module variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable that has dependencies on another object.
VARMODULEID	INTEGER	Y	Identifier for the module of the object that has dependencies on another object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • F = Routine • G = Global temporary table • H = Hierarchy table • I = Index • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

Table 220. SYSCAT.VARIABLEDEP 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 the dependent global variable; the null value otherwise.

SYSCAT.VARIABLES

Each row represents a global variable.

Table 221. SYSCAT.VARIABLES Catalog View

Column Name	Data Type	Nullable	Description
VARSCHEMA	VARCHAR (128)		Schema name of the global variable if VARMODULEID is null; otherwise schema name of the module to which the global variable belongs.
VARMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the global variable belongs. The null value if not a module variable.
VARNAME	VARCHAR (128)		Unqualified name of the global variable.
VARMODULEID	INTEGER	Y	Identifier for the module to which the global variable belongs. The null value if not a module variable.
VARID	INTEGER		Identifier for the global variable.
OWNER	VARCHAR (128)		Authorization ID of the owner of the global variable.
OWNERTYPE	CHAR (1)		• U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the global variable was created.
LAST_REGEN_TIME	TIMESTAMP		Time at which the default expression was last regenerated.
VALID	CHAR (1)		• N = The global variable is invalid • Y = The global variable is valid
PUBLISHED	CHAR (1)		Indicates whether the module variable can be referenced outside its module. • N = The module variable is not published • Y = The module variable is published • Blank = Not applicable
TYPESCHEMA	VARCHAR (128)		Schema name of the data type if TYPEMODULEID is null; otherwise schema name of the module to which the data type belongs.
TYPEMODULENAME	VARCHAR (128)		Unqualified name of the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.
TYPENAME	VARCHAR (128)		Unqualified name of the data type.
TYPEMODULEID	INTEGER	Y	Identifier for the module to which the variable data type belongs. The null value if the variable data type does not belong to a module.
LENGTH	INTEGER		Maximum length of the global variable.

Table 221. SYSCAT.VARIABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SCALE	SMALLINT		Scale if the global variable data type is DECIMAL or distinct type based on DECIMAL; the number of digits of fractional seconds if the global variable data type is TIMESTAMP or distinct type based on TIMESTAMP; 0 otherwise.
TYPESTRINGUNITS	VARCHAR (11)	Y	In a Unicode database, the string units that apply to a character string or graphic string data type. Otherwise, the null value.
STRINGUNITSLENGTH	INTEGER	Y	In a Unicode database, the declared number of string units for a character string or graphic string data type. Otherwise, the null value.
CODEPAGE	SMALLINT		Code page of the global variable.
COLLATIONSCHEMA	VARCHAR (128)		Schema name of the collation for the variable.
COLLATIONNAME	VARCHAR (128)		Unqualified name of the collation for the variable.
COLLATIONSCHEMA_ORDERBY	VARCHAR (128)		Schema name of the collation for ORDER BY clauses in the variable.
COLLATIONNAME_ORDERBY	VARCHAR (128)		Unqualified name of the collation for ORDER BY clauses in the variable.
SCOPE	CHAR (1)		Scope of the global variable. • D = Database • S = Session
DEFAULT	CLOB (64K)	Y	Expression used to calculate the initial value of the global variable when first referenced.
QUALIFIER	VARCHAR (128)	Y	Value of the default schema at the time the variable was defined.
FUNC_PATH	CLOB (2K)	Y	SQL path in effect when the variable was defined.
NULLS	CHAR (1)		Reserved for future use.
READONLY	CHAR (1)		• C = Read-only because the global variable is defined with a CONSTANT clause • N = Not read-only • S = Read-only because the global variable value is maintained by the database manager
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.VIEWS

Each row represents a view or materialized query table.

Table 222. SYSCAT.VIEWS Catalog View

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	VARCHAR (128)		Schema name of the view or materialized query table.
VIEWNAME	VARCHAR (128)		Unqualified name of the view or materialized query table.
OWNER	VARCHAR (128)		Authorization ID of the owner of the view or materialized query table.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
SEQNO	SMALLINT		Always 1.
VIEWCHECK	CHAR (1)		Type of view checking. • C = Cascaded check option • L = Local check option • N = No check option or is a materialized query table
READONLY	CHAR (1)		• N = View can be updated by users with appropriate authorization or is a materialized query table • Y = View is read-only because of its definition
VALID	CHAR (1)		• N = View or materialized query table definition is invalid • X = View or materialized query table definition is inoperative and must be recreated • Y = View or materialized query table definition is valid
QUALIFIER	VARCHAR (128)		Value of the default schema at the time of object definition. Used to complete any unqualified references.
FUNC_PATH	CLOB (2K)		SQL path in effect when the view or materialized query table was defined.
TEXT	CLOB (2M)		Full text of the view or materialized query table CREATE statement, exactly as typed.
DEFINER[1]	VARCHAR (128)		Authorization ID of the owner of the view or materialized query table.
ENVSTRINGUNITS	VARCHAR (11)		Default string units when the object was created.

Note:

1. The DEFINER column is included for backwards compatibility. See OWNER.

SYSCAT.WORKACTIONS

Each row represents a work action that is defined for a work action set.

Table 223. 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)		• N = This work action is disabled. • Y = This work action is enabled.

Table 223. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
ACTIONTYPE	CHAR (1)		The type of action performed on each DB2 activity that matches the attributes in the work class within scope.

- B = Collect basic aggregate activity data, specifiable only for work action sets that apply to service classes or workloads.
- C = Allow any DB2 activity under the associated work class to execute and increment the work class counter.
- D = Collect activity data with details at the coordinating member of the activity.
- E = Collect extended aggregate activity data, specifiable only for work action sets that apply to service classes or workloads.
- F = Collect activity data with details, section, and values at the coordinating member of the activity.
- G = Collect activity details and section at the coordinating member of the activity and collect activity data at all members.
- H = Collect activity details, section, and values at the coordinating member of the activity and collect activity data at all members.
- M = Map to a service subclass, specifiable only for work action sets that apply to service classes.
- P = Prevent the execution of any DB2 activity under the work class with which this work action is associated.
- S = Collect activity data with details and section at the coordinating member of the activity.
- T = The action represents a threshold, specifiable only for work action sets that are associated with a database or a workload.
- U = Map all activities with a nesting level of zero and all activities nested under these activities to a service subclass, specifiable only for work action sets that apply to service classes.
- V = Collect activity data with details and values at the coordinating member.
- W = Collect activity data without details at the coordinating member.
- X = Collect activity data with details at the coordinating member and collect activity data at all members.
- Y = Collect activity data with details and values at the coordinating member and collect activity data at all members.
- Z = Collect activity data without details at all members.

Table 223. SYSCAT.WORKACTIONS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFOBJECTID	INTEGER	Y	If ACTIONTYPE is 'M' (map) or 'N' (map nested), this value is set to the ID of the service subclass to which the DB2 activity is mapped. If ACTIONTYPE is 'T' (threshold), this value is set to the ID of the threshold to be used. For all other actions, this value is NULL.
REFOBJECTTYPE	VARCHAR (30)		If the ACTIONTYPE is 'M' or 'N', this value is set to 'SERVICE CLASS'; if the ACTIONTYPE is 'T', this value is 'THRESHOLD'; the null value otherwise.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for for future use.

SYSCAT.WORKACTIONSETS

Each row represents a work action set.

Table 224. SYSCAT.WORKACTIONSETS Catalog View

Column Name	Data Type	Nullable	Description
ACTIONSETNAME	VARCHAR (128)		Name of the work action set.
ACTIONSETID	INTEGER		Identifier for the work action set.
WORKCLASSSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSSETID	INTEGER		The identifier of the work class set that is to be mapped to the object specified by the OBJECTID. This column refers to WORKCLASSSETID in the SYSCAT.WORKCLASSSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work action set was created.
ALTER_TIME	TIMESTAMP		Time at which the work action set was last altered.
ENABLED	CHAR (1)		• N = This work action set is disabled. • Y = This work action set is enabled.
OBJECTTYPE	CHAR (1)		• b = Service superclass • w = Workload • Blank = Database
OBJECTNAME	VARCHAR (128)	Y	Name of the service class or workload.
OBJECTID	INTEGER		The identifier of the object to which the work class set (specified by the WORKCLASSSETID) is mapped. If the OBJECTTYPE is 'b', the OBJECTID is the ID of the service superclass. If the OBJECTTYPE is 'w', the OBJECTID is the ID of the workload. If the OBJECTTYPE is blank, the OBJECTID is -1.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WORKCLASSATTRIBUTES

Each row represents an attribute in the definition of a work class.

Table 225. SYSCAT.WORKCLASSATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSNAME	VARCHAR (128)		Name of the work class.
WORKCLASSSETNAME	VARCHAR (128)		Name of the work class set.
WORKCLASSID	INTEGER		Identifier for the work class.
WORKCLASSSETID	INTEGER		Identifier for the work class set.
TYPE	VARCHAR (30)		The type of work class attribute. Possible values are: • WORK TYPE • TIMERONCOST • CARDINALITY • DATA TAG • ROUTINE SCHEMA
VALUE1	DOUBLE	Y	If the TYPE is WORK TYPE, the type of database activity using one of the following values. • 1 = ALL • 2 = READ • 3 = WRITE • 4 = CALL • 5 = DML • 6 = DDL • 7 = LOAD If the TYPE is 'TIMERONCOST' or 'CARDINALITY', the low value in the range. If the TYPE is 'DATA TAG', this value represents the tag that the estimated data tag list must contain. Otherwise, the null value.
VALUE2	DOUBLE	Y	If the TYPE is 'TIMERONCOST' or 'CARDINALITY', the high value in the range. -1 is used to indicate no upper bound. Otherwise, the null value.
VALUE3	VARCHAR (128)	Y	If the TYPE is 'ROUTINE SCHEMA', the schema name of the procedures that are called from the CALL statement. Otherwise, the null value.

SYSCAT.WORKCLASSES

Each row represents a work class defined for a work class set.

Table 226. SYSCAT.WORKCLASSES Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSNAME	VARCHAR (128)		Name of the work class.
WORKCLASSSETNAME	VARCHAR (128)	Y	Name of the work class set.
WORKCLASSID	INTEGER		Identifier for the work class.
WORKCLASSSETID	INTEGER		Identifier for the work class set to which this work class belongs. This column refers to the WORKCLASSSETID column in the SYSCAT.WORKCLASSSETS view.
CREATE_TIME	TIMESTAMP		Time at which the work class was created.
ALTER_TIME	TIMESTAMP		Time at which the work class was last altered.
EVALUATIONORDER	SMALLINT		Uniquely identifies the evaluation order used for choosing a work class within a work class set.

SYSCAT.WORKCLASSSETS

Each row represents a work class set.

Table 227. SYSCAT.WORKCLASSSETS Catalog View

Column Name	Data Type	Nullable	Description
WORKCLASSSETNAME	VARCHAR (128)		Name of the work class set.
WORKCLASSSETID	INTEGER		Identifier for the work class set.
CREATE_TIME	TIMESTAMP		Time at which the work class set was created.
ALTER_TIME	TIMESTAMP		Time at which the work class set was last altered.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WORKLOADAUTH

Each row represents a user, group, or role that has been granted USAGE privilege on a workload.

Table 228. 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)		• U = Grantee is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
USAGEAUTH	CHAR (1)		Indicates whether grantee holds USAGE privilege on the workload. • N = Not held • Y = Held

SYSCAT.WORKLOADCONNATTR

Each row represents a connection attribute in the definition of a workload.

Table 229. SYSCAT.WORKLOADCONNATTR Catalog View

Column Name	Data Type	Nullable	Description
WORKLOADID	INTEGER		Identifier for the workload.
WORKLOADNAME	VARCHAR (128)		Name of the workload.
CONNATTRTYPE	VARCHAR (30)		Type of the connection attribute. • 1 = APPLNAME • 2 = SYSTEM_USER • 3 = SESSION_USER • 4 = SESSION_USER GROUP • 5 = SESSION_USER ROLE • 6 = CURRENT CLIENT_USERID • 7 = CURRENT CLIENT_APPLNAME • 8 = CURRENT CLIENT_WRKSTNNAME • 9 = CURRENT CLIENT_ACCTNG • 10 = ADDRESS
CONNATTRVALUE	VARCHAR (1000)		Value of the connection attribute.

SYSCAT.WORKLOADS

Each row represents a workload.

Table 230. 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)		• N = This workload is disabled. • Y = This workload is enabled.
ALLOWACCESS	CHAR (1)		• N = A UOW associated with this workload will be rejected. • Y = A unit of work (UOW) associated with this workload can access the database.
MAXDEGREE	SMALLINT		Maximum degree of parallelism for the workload. The valid values are: 1 to 32767, and -1. If MAXIMUM DEGREE is DEFAULT, the value is -1.
SERVICECLASSNAME	VARCHAR (128)		Name of the service subclass to which a unit of work (associated with this workload) is assigned.
PARENTSERVICECLASSNAME	VARCHAR (128)	Y	Name of the service superclass to which a unit of work (associated with this workload) is assigned.
COLLECTAGGACTDATA	CHAR (1)		Specifies what aggregate activity data should be captured for the workload by the applicable event monitor. • B = Collect base aggregate activity data • E = Collect extended aggregate activity data • N = None
COLLECTACTDATA	CHAR (1)		Specifies what activity data should be collected by the applicable event monitor. • D = Activity data with details • N = None • S = Activity data with details and section environment • V = Activity data with details and values. Applies when the COLLECT column is set to 'C' • W = Activity data without details • X = Activity data with details, section environment, and values
COLLECTACTPARTITION	CHAR (1)		Specifies where activity data is collected. • C = Coordinator member of the activity • D = All members

Table 230. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTDEADLOCK	CHAR (1)		Specifies that deadlock events should be collect by the applicable event monitor. • H = Collect deadlock data with past activities only • V = Collect deadlock data with past activities and values • W = Collect deadlock data without past activities and values
COLLECTLOCKTIMEOUT	CHAR (1)		Specifies that lock timeout events should be collect by the applicable event monitor. • H = Collect lock timeout data with past activities only • N = Do not not collect lock timeout data • V = Collect lock timeout data with past activities and values • W = Collect lock timeout data without past activities and values
COLLECTLOCKWAIT	CHAR (1)		Specifies that lock wait events should be collect by the applicable event monitor. • H = Collect lock wait data with past activities only • N = Do not not collect lock wait data • V = Collect lock wait data with past activities and values • W = Collect lock wait data without past activities and values
LOCKWAITVALUE	INTEGER		Specifies the time in milliseconds a lock should wait before a lock event is collected by the applicable event monitor; 0 when COLLECTLOCKWAIT = 'N'
COLLECTACTMETRICS	CHAR (1)		Specifies the monitoring level for activities submitted by an occurrence of the workload. • B = Collect base activity metrics • E = Collect extended activity metrics • N = None

SYSCAT.WORKLOADS

Table 230. SYSCAT.WORKLOADS Catalog View (continued)

Column Name	Data Type	Nullable	Description
COLLECTUOWDATAOPTIONS	VARCHAR (32)		Specifies what unit of work data is collected by the applicable event monitor. The first position in the string represents whether the collection of unit of work data is enabled. • B = Enabled and collect base unit of work data • N = Not enabled Starting from second position, each position in the string represents a specific extended option: • 2 = Package Reference List • 3 = Executable ID list Each position representing an extended option is then set to one of the following values: • Y = Extended option is included • N = Extended option is not included
COLLECTUOWDATA	CHAR (1)		Specifies what unit of work data should be collected by the applicable event monitor. • B = Collect base unit of work data • N = None • P = Collect base unit of work data and the package list This column is deprecated. Information for the column is available from COLLECTUOWDATAOPTIONS.
EXTERNALNAME	VARCHAR (128)	Y	Reserved for future use.
SECTIONACTUALSOPTIONS	VARCHAR (32)		Specifies what section actuals are collected during the execution of a section. The first position in the string represents the whether the collection of section actuals is enabled. • B = Enabled and collect basic operator cardinality counts and statistics for each object referenced by the section (DML statements only). • N = Not enabled. The second position is always 'N' and reserved for for future use.
COLLECTAGGUOWDATA	CHAR (1)		Specifies what aggregate unit of work data should be captured for the workload by the applicable event monitor. • B = Collect base aggregate unit of work data • N = None
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.WRAPOPTIONS

Each row represents a wrapper-specific option.

Table 231. 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 232. SYSCAT.WRAPPERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR (128)		Name of the wrapper.
WRAPTYPE	CHAR (1)		Type of wrapper. • N = Non-relational • R = Relational
WRAPVERSION	INTEGER		Version of the wrapper.
LIBRARY	VARCHAR (255)		Name of the file that contains the code used to communicate with the data sources that are associated with this wrapper.
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSCAT.XDBMAPGRAPHS

Each row represents a schema graph for an XDB map (XSR object).

Table 233. 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 234. SYSCAT.XDBMAPSHREDTREES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
SCHEMAGRAPHID	INTEGER		Schema graph identifier, which is unique within an XDB map identifier.
SHREDTREEID	INTEGER		Shred tree identifier, which is unique within an XDB map identifier.
MAPPINGDESCRIPTION	CLOB (1M)	Y	Diagnostic mapping information.

SYSCAT.XMLSTRINGS

Each row represents a single string and its unique string ID, used to condense structural XML data. The string is provided in both UTF-8 encoding and database code page encoding.

Table 235. SYSCAT.XMLSTRINGS Catalog View

Column Name	Data Type	Nullable	Description
STRINGID	INTEGER		Unique string ID.
STRING	VARCHAR (1001)		The string represented in the database code page.
STRING_UTF8	VARCHAR (1001)		The string in UTF-8 encoding (as stored in the catalog table).

SYSCAT.XSROBJECTAUTH

Each row represents a user, group, or role that has been granted the USAGE privilege on a particular XSR object.

Table 236. SYSCAT.XSROBJECTAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR (128)		Grantor of the privilege.
GRANTORTYPE	CHAR (1)		• S = Grantor is the system • U = Grantor is an individual user
GRANTEE	VARCHAR (128)		Holder of the privilege.
GRANTEETYPE	CHAR (1)		• G = Grantee is a group • R = Grantee is a role • U = Grantee is an individual user
OBJECTID	BIGINT		Identifier for the XSR object.
USAGEAUTH	CHAR (1)		Privilege to use the XSR object and its components. • N = Not held • Y = Held

SYSCAT.XSROBJECTCOMPONENTS

Each row represents an XSR object component.

Table 237. SYSCAT.XSROBJECTCOMPONENTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
COMPONENTID	BIGINT		Unique generated identifier for an XSR object component.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location.
COMPONENT	BLOB (30M)		External representation of the component.
CREATE_TIME	TIMESTAMP		Time at which the XSR object component was registered.
STATUS	CHAR (1)		Registration status. • C = Complete • I = Incomplete

SYSCAT.XSROBJECTDEP

Each row represents a dependency of an XSR object on some other object. The XSR object depends on the object of type BTYPE of name BNAME, so a change to the object affects the XSR object.

Table 238. SYSCAT.XSROBJECTDEP Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
BTYPE	CHAR (1)		Type of object on which there is a dependency. Possible values are: • A = Table alias • B = Trigger • C = Column • F = Routine • G = Global temporary table • H = Hierachy table • I = Index • K = Package • L = Detached table • N = Nickname • O = Privilege dependency on all subtables or subviews in a table or view hierarchy • Q = Sequence • R = User-defined data type • S = Materialized query table • T = Table (not typed) • U = Typed table • V = View (not typed) • W = Typed view • X = Index extension • Z = XSR object • m = Module • q = Sequence alias • u = Module alias • v = Global variable • * = Anchored to the row of a base table
BSCHEMA	VARCHAR (128)		Schema name of the object on which there is a dependency.
BMODULENAME	VARCHAR (128)	Y	Unqualified name of the module to which the object on which a dependency belongs. The null value if not a module object.
BNAME	VARCHAR (128)		Unqualified name of the object on which there is a dependency. For routines (BTYPE = 'F'), this is the specific name.
BMODULEID	INTEGER	Y	Identifier for the module of the object on which there is a dependency.

Table 238. 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.XSROBJECTDETAILS

Each row represents an XML schema repository object.

Table 239. SYSCAT.XSROBJECTDETAILS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XML schema object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XML schema object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XML schema object.
GRAMMAR	BLOB (127M)	Y	Binary representation of the grammar for the XML schema object.
PROPERTIES	BLOB (4190000)	Y	Properties document for the XML schema object.

SYSCAT.XSROBJECTHIERARCHIES

Each row represents the hierarchical relationship between an XSR object and its components.

Table 240. SYSCAT.XSROBJECTHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Identifier for an XSR object.
COMPONENTID	BIGINT		Identifier for an XSR component.
HTYPE	CHAR (1)		Hierarchy type. • D = Document • N = Top-level namespace • P = Primary document
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the component's target namespace.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the component's schema location.

SYSCAT.XSROBJECTS

Each row represents an XML schema repository object.

Table 241. SYSCAT.XSROBJECTS Catalog View

Column Name	Data Type	Nullable	Description
OBJECTID	BIGINT		Unique generated identifier for an XSR object.
OBJECTSCHEMA	VARCHAR (128)		Schema name of the XSR object.
OBJECTNAME	VARCHAR (128)		Unqualified name of the XSR object.
TARGETNAMESPACE	VARCHAR (1001)	Y	String identifier for the target namespace, or public identifier.
SCHEMALOCATION	VARCHAR (1001)	Y	String identifier for the schema location, or system identifier.
OBJECTINFO	XML	Y	Metadata document.
OBJECTTYPE	CHAR (1)		XSR object type. • D = DTD • E = External entity • S = XML schema
OWNER	VARCHAR (128)		Authorization ID of the owner of the XSR object.
OWNERTYPE	CHAR (1)		• S = The owner is the system • U = The owner is an individual user
CREATE_TIME	TIMESTAMP		Time at which the object was registered.
ALTER_TIME	TIMESTAMP		Time at which the object was last updated (replaced).
STATUS	CHAR (1)		Registration status. • C = Complete • I = Incomplete • R = Replace • T = Temporary
DECOMPOSITION	CHAR (1)		Indicates whether or not decomposition (shredding) is enabled on this XSR object. • N = Not enabled • X = Inoperative • Y = Enabled
REMARKS	VARCHAR (254)	Y	User-provided comments, or the null value.

SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with DB2 for z/OS.

Table 242. 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 243. SYSSTAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table to which the statistics apply.
TABNAME	VARCHAR (128)			Unqualified name of the table to which the statistics apply.
COLNAME	VARCHAR (128)			Name of the column to which the statistics apply.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT			If TYPE = 'F', *n* in this column identifies the *n*th most frequent value. If TYPE = 'Q', *n* in this column identifies the *n*th quantile value.
COLVALUE[1]	VARCHAR (254)	Y	Y	Data value as a character literal or a null value.
VALCOUNT	BIGINT		Y	If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT[2]	BIGINT	Y	Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE (the null value if unavailable).

Note:

1. In the catalog view, the value of COLVALUE is always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.
2. DISTCOUNT is collected only for columns that are the first key column in an index.

SYSSTAT.COLGROUPDIST

Each row represents the value of the column in a column group that makes up the nth most frequent value of the column group or the nth quantile value of the column group.

Table 244. SYSSTAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
ORDINAL	SMALLINT			Ordinal number of the column in the column group.
SEQNO	SMALLINT			If TYPE = 'F', n in this column identifies the nth most frequent value. If TYPE = 'Q', n in this column identifies the nth 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 nth most frequent value of a column group or the nth quantile of a column group.

Table 245. SYSSTAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Updat- able	Description
COLGROUPID	INTEGER			Identifier for the column group.
TYPE	CHAR (1)			• F = Frequency value • Q = Quantile value
SEQNO	SMALLINT			Sequence number n representing the nth TYPE value.
VALCOUNT	BIGINT		Y	If TYPE = 'F', VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = 'Q', VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT		Y	If TYPE = 'Q', this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (the null value if unavailable).

SYSSTAT.COLGROUPS

Each row represents a column group and statistics that apply to the entire column group.

Table 246. 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 247. SYSSTAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
TABSCHEMA	VARCHAR (128)			Schema name of the table, view, or nickname that contains the column.
TABNAME	VARCHAR (128)			Unqualified name of the table, view, or nickname that contains the column.
COLNAME	VARCHAR (128)			Name of the column.
COLCARD	BIGINT		Y	Number of distinct values in the column; -1 if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
HIGH2KEY[1]	VARCHAR (254)	Y	Y	Second-highest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
LOW2KEY[1]	VARCHAR (254)	Y	Y	Second-lowest data value. Representation of numeric data changed to character literals. Empty if statistics are not collected. Empty for inherited columns and columns of hierarchy tables.
AVGCOLLEN	INTEGER		Y	Average space in bytes when the column is stored in database memory or a temporary table. For LOB data types that are not inlined, LONG data types, and XML documents, the value used to calculate the average column length is the length of the data descriptor. An extra byte is required if the column is nullable; -1 if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables. Note: The average space required to store the column on disk may be different than the value represented by this statistic.
NUMNULLS	BIGINT		Y	Number of null values in the column; -1 if statistics are not collected.
PCTINLINED	SMALLINT			Percentage of inlined data for columns with VARCHAR, VARGRAPHIC, LOB, or XML data types. -1 if statistics have not been collected or the column data type does not support storing data outside the row. Also -1 for VARCHAR and VARGRAPHIC column if the table is organized by column or the table is organized by row and the row size of the table does not exceed the maximum record length for the page size of the table space.

Table 247. SYSSTAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
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.
PCTENCODED	SMALLINT		Y	Percentage of values that are encoded as a result of compression for a column in a column-organized table; -1 if the table is not organized by column or if statistics are not collected; -2 for inherited columns and columns of hierarchy tables.
AVGCOLLENCHAR	INTEGER		Y	Average number of characters (based on the collation in effect for the column) required for the column; -1 if the data type of the column is long, LOB, or XML or if statistics have not been collected; -2 for inherited columns and columns of hierarchy tables.

Note:

1. In the catalog view, the values of HIGH2KEY and LOW2KEY are always shown in the database code page and can contain substitution characters. However, the statistics are gathered internally in the code page of the column's table, and will therefore use actual column values when applied during query optimization.

SYSSTAT.INDEXES

Each row represents an index. Indexes on typed tables are represented by two rows: one for the "logical index" on the typed table, and one for the "H-index" on the hierarchy table.

Table 248. SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
INDSCHEMA	VARCHAR (128)			Schema name of the index.
INDNAME	VARCHAR (128)			Unqualified name of the index.
TABSCHEMA	VARCHAR (128)			Schema name of the table or nickname on which the index is defined.
TABNAME	VARCHAR (128)			Unqualified name of the table or nickname on which the index is defined.
COLNAMES	VARCHAR (640)			This column is no longer used and will be removed in the next release. Use SYSCAT.INDEXCOLUSE for this information.
NLEAF	BIGINT		Y	Number of leaf pages; -1 if statistics are not collected.
NLEVELS	SMALLINT		Y	Number of index levels; -1 if statistics are not collected.
FIRSTKEYCARD	BIGINT		Y	Number of distinct first-key values; -1 if statistics are not collected.
FIRST2KEYCARD	BIGINT		Y	Number of distinct keys using the first two columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST3KEYCARD	BIGINT		Y	Number of distinct keys using the first three columns of the index; -1 if statistics are not collected, or if not applicable.
FIRST4KEYCARD	BIGINT		Y	Number of distinct keys using the first four columns of the index; -1 if statistics are not collected, or if not applicable.
FULLKEYCARD	BIGINT		Y	Number of distinct full-key values; -1 if statistics are not collected.
CLUSTERRATIO[4]	SMALLINT		Y	Degree of data clustering with the index; -1 if statistics are not collected or if detailed index statistics are collected (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR[4]	DOUBLE		Y	Finer measurement of the degree of clustering; -1 if statistics are not collected or if the index is defined on a nickname.
SEQUENTIAL_PAGES	BIGINT		Y	Number of leaf pages located on disk in index key order with few or no large gaps between them; -1 if statistics are not collected.

Table 248. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat- able	Description
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.
PAGE_FETCH_PAIRS[4]	VARCHAR (520)		Y	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. Zero-length string if no data is available.
NUMRIDS[4]	BIGINT		Y	Total number of row identifiers (RIDs) or block identifiers (BIDs) in the index; -1 if not known.
NUMRIDS_DELETED[4]	BIGINT		Y	Total number of row identifiers (or block identifiers) in the index that are marked deleted, excluding those identifiers on leaf pages on which all the identifiers are marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Y	Total number of index leaf pages that have all of their row identifiers (or block identifiers) marked deleted.
AVERAGE_RANDOM_ FETCH_PAGES[1,2,4]	DOUBLE		Y	Average number of random table pages between sequential page accesses when fetching using the index; -1 if not known.
AVERAGE_RANDOM_ PAGES[2]	DOUBLE		Y	Average number of random table pages between sequential page accesses; -1 if not known.
AVERAGE_SEQUENCE_ GAP[2]	DOUBLE		Y	Gap between index page sequences. Detected through a scan of index leaf pages, each gap represents the average number of index pages that must be randomly fetched between sequences of index pages; -1 if not known.
AVERAGE_SEQUENCE_ FETCH_GAP[1,2,4]	DOUBLE		Y	Gap between table page sequences when fetching using the index. Detected through a scan of index leaf pages, each gap represents the average number of table pages that must be randomly fetched between sequences of table pages; -1 if not known.
AVERAGE_SEQUENCE_ PAGES[2]	DOUBLE		Y	Average number of index pages that are accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); -1 if not known.

SYSSTAT.INDEXES

Table 248. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updatable	Description
AVERAGE_SEQUENCE_ FETCH_PAGES[1,2,4]	DOUBLE		Y	Average number of table pages that are accessible in sequence (that is, the number of table pages that the prefetchers would detect as being in sequence) when fetching using the index; -1 if not known.
AVGPARTITION_ CLUSTERRATIO[3,4]	SMALLINT		Y	Degree of data clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if detailed statistics are collected (in which case AVGPARTITION_ CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR[3,4]	DOUBLE		Y	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not partitioned, if statistics are not collected, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS[3,4]	VARCHAR (520)		Y	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE		Y	A statistic measuring the "clustering" of the index keys with regard to data partitions. It is a number between 0 and 1, with 1 representing perfect clustering and 0 representing no clustering.
INDCARD	BIGINT		Y	Cardinality of the index. This might be different from the cardinality of the table for indexes that do not have a one-to-one relationship between the table rows and the index entries.
PCTPAGESSAVED	SMALLINT			Approximate percentage of pages saved in the index as a result of index compression. -1 if statistics are not collected.
AVGLEAFKEYSIZE	INTEGER		Y	Average index key size for keys on leaf pages in the index.
AVGNLEAFKEYSIZE	INTEGER		Y	Average index key size for keys on non-leaf pages in the index.

Table 248. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Updat- able	Description

Note:

1. When using DMS table spaces, this statistic cannot be computed.

2. Prefetch statistics are not gathered during a LOAD...STATISTICS USE PROFILE, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.

3. AVGPARTITION_CLUSTERRATIO, AVGPARTITION_CLUSTERFACTOR, and AVGPARTITION_PAGE_FETCH_PAIRS measure the degree of clustering within a single data partition (local clustering). CLUSTERRATIO, CLUSTERFACTOR, and PAGE_FETCH_PAIRS measure the degree of clustering in the entire table (global clustering). Global clustering and local clustering values can diverge significantly if the table partitioning key is not a prefix of the index key, or when the table partitioning key and the index key are logically independent of each other.

4. This statistic cannot be updated if the index type is 'XPTH' (an XML path index).

5. Because logical indexes on an XML column do not have statistics, the SYSSTAT.INDEXES catalog view excludes rows whose index type is 'XVIL'.

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 249. SYSSTAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Updat-able	Description
ROUTINESCHEMA	VARCHAR (128)			Schema name of the routine if ROUTINEMODULENAME is null; otherwise schema name of the module to which the routine belongs.
ROUTINEMODULENAME	VARCHAR (128)			Unqualified name of the module to which the routine belongs. The null value if not a module routine.
ROUTINENAME	VARCHAR (128)			Unqualified name of the routine.
ROUTINETYPE	CHAR (1)			Type of routine. • F = Function • M = Method • P = Procedure
SPECIFICNAME	VARCHAR (128)			Name of the routine instance (might be system-generated).
IOS_PER_INVOC	DOUBLE		Y	Estimated number of inputs/outputs (I/Os) per invocation; 0 is the default; -1 if not known.
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 250. 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.
CARD	BIGINT		Y	Total number of rows in the table; -1 if statistics are not collected.
NPAGES	BIGINT		Y	Total number of pages on which the rows of the table exist; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
MPAGES	BIGINT		Y	Total number of pages for table metadata. Non-zero only for a table that is organized by column; -1 for a view, an alias, or if statistics are not collected; -2 for subtables or hierarchy tables.
FPAGES	BIGINT		Y	Total number of pages; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
OVERFLOW	BIGINT		Y	Total number of overflow records in the table; -1 for a view or alias, or if statistics are not collected; -2 for a subtable or hierarchy table.
CLUSTERED	CHAR (1)	Y		• T = Table is clustered by insert time • Y = Table is clustered by dimensions (even if only by one dimension) • Null value = Table is not clustered by dimensions or insert time
ACTIVE_BLOCKS	BIGINT		Y	Total number of active blocks in the table, or -1. Applies to multidimensional clustering (MDC) tables or insert time clustering (ITC) tables only.
AVGCOMPRESSEDROWSIZE	SMALLINT		Y	Average length (in bytes) of compressed rows in this table; -1 if statistics are not collected.
AVGROWCOMPRESSIONRATIO	REAL		Y	For compressed rows in the table, this is the average compression ratio by row; that is, the average uncompressed row length divided by the average compressed row length; -1 if statistics are not collected.

SYSSTAT.TABLES

Table 250. SYSSTAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Updat-able	Description
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 a row-organized table as a result of row compression. For a column-organized table, the estimate is based on the number of data pages needed to store the table in uncompressed row organization.-1 if statistics are not collected.
PCTEXTENDEDROWS	REAL			Extended rows as a percentage of the total number of rows in the table; -1 if statistics are not collected.

Appendix E. The SAMPLE database

The sample database can be used for various purposes like testing your applications, trying different features of DB2 database products and so on. Most of the sample application programs under *DB2PATH*/sqllib/samples use sample database for demonstrating various features of DB2 databases 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

ADEFUSR table

Name:	WORKDEPT	NO_OF_EMPLOYEES
Type:	CHAR(3)	INTEGER

The SAMPLE database

Name:	WORKDEPT	NO_OF_EMPLOYEES
Values:	A00	5
	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

Name:	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
Type:	CHAR(3) NOT NULL	VARCHAR(29) NOT NULL	CHAR(6)	CHAR(3) NOT NULL	CHAR(16)
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	
	B01	PLANNING	000020	A00	
	C01	INFORMATION CENTER	000030	A00	
	D01	DEVELOPMENT CENTER		A00	
	D11	MANUFACTURING SYSTEMS	000060	D01	
	D21	ADMINISTRATION SYSTEMS	000070	D01	
	E01	SUPPORT SERVICES	000050	A00	
	E11	OPERATIONS	000090	E01	
	E21	SOFTWARE SUPPORT	000100	E01	
	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.

Names:	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
Type:	CHAR(6) NOT NULL	VARCHAR(12) NOT NULL	CHAR(1) NOT NULL	VARCHAR(15) NOT NULL	CHAR(3)	CHAR(4)	DATE
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

+

JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
CHAR(8)	SMALLINT NOT NULL	CHAR(1)	DATE	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	FIRSTNME	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	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FIELDREP	16	M	1932-08-11	19950	400	1596
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	1983-02-01
	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

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

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	LUCCHESSI	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	LUCCHESSI	Ontario-South	3
	03/29/2006	LUCCHESSI	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	LUCCHESSI	Ontario-South	1
	03/30/2006	LUCCHESSI	Quebec	2
	03/30/2006	LUCCHESSI	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	LUCCHESSI	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	LUCCHESSI	Ontario-South	3
	04/01/2006	LUCCHESSI	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

The SAMPLE database

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 preceding 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 preceding 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 preceding table: `customer.xsd`

CATALOG table

Name:	NAME	CATALOG
Type:	VARCHAR(128) NOT NULL	XML

Here is the XML schema definition file for the XML column in the preceding 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

PRODUCTSUPPLIER table

Name:	PID	SID
Type:	VARCHAR(10) NOT NULL	VARCHAR(10) NOT NULL
Values:	100-101-01	100
	100-201-01	101

SUPPLIERS table

Name:	SID	ADDR
Type:	VARCHAR(10) NOT NULL	XML
Values:	100	s1.xml
	101	s2.xml

Here is the XML schema definition file for the XML column in the preceding 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 17. 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
<!-- 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  =========================================== -->
<A NAME="Top_Of_Page"><H1>Resume:  Delores M. Quintana<HR>
<H2><A NAME="ToC">Table of Contents<A NAME="ToC_1" HREF="#Header_1">
Resume:  Delores M. Quintana<BR>
<A NAME="ToC_2" HREF="#Header_2">Personal Information<BR>
<A NAME="ToC_3" HREF="#Header_3">Department Information<BR>
<A NAME="ToC_4" HREF="#Header_4">Education<BR>
<A NAME="ToC_5" HREF="#Header_5">Work History<BR>
<A NAME="ToC_6" HREF="#Header_6">Interests<BR>
<HR><P>
```

The SAMPLE database

```
<P>
<H3><A NAME="Header_1">Resume:  Delores M. Quintana<P>
<H3><A NAME="Header_2">Personal Information<DL COMPACT>
<DT>Address:
<DD>1150 Eglinton Ave
<BR>
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><A NAME="Header_3">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><A NAME="Header_4">Education<DL>
<P><DT>1965
<DD>Math and English, B.A.
<BR>
Adelphi University
<P><DT>1960
<DD>Dental Technician
<BR>
Florida Institute of Technology<P>
<H3><A NAME="Header_5">Work History<DL>
<P><DT>10/91 - present
<DD>Advisory Systems Analyst
<BR>
Producing documentation tools for engineering department.
<P><DT>12/85 - 9/91
<DD>Technical Writer
<BR>
Writer, text programmer, and planner.
<P><DT>1/79 - 11/85
<DD>COBOL Payroll Programmer
<BR>
Writing payroll programs for a diesel fuel company.<P>
<H3><A NAME="Header_6">Interests<UL COMPACT>
<LI>Cooking
<LI>Reading
<LI>Sewing
<LI>Remodeling<A NAME="Bot_Of_Page">
```

Nicholls photo

Figure 18. 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

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  ========================================= -->
<A NAME="Top_Of_Page"><H1>Resume:  Heather A. Nicholls<HR>
<H2><A NAME="ToC">Table of Contents<A NAME="ToC_1" HREF="#Header_1">
Resume:  Heather A. Nicholls<BR>
<A NAME="ToC_2" HREF="#Header_2">Personal Information<BR>
<A NAME="ToC_3" HREF="#Header_3">Department Information<BR>
<A NAME="ToC_4" HREF="#Header_4">Education<BR>
<A NAME="ToC_5" HREF="#Header_5">Work History<BR>
<A NAME="ToC_6" HREF="#Header_6">Interests<BR>
<HR><P>
<P>
<H3><A NAME="Header_1">Resume:  Heather A. Nicholls<P>
<H3><A NAME="Header_2">Personal Information<DL COMPACT>
<DT>Address:
<DD>844 Don Mills Ave
<BR>
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
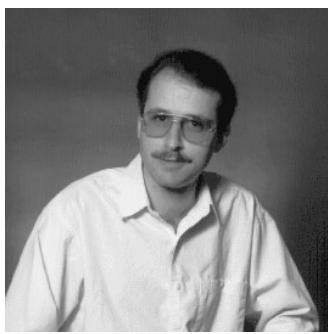
Figure 19. Bruce Adamson

Adamson resume

The following text is found in the file db200150.asc.

Resume: Bruce Adamson

The SAMPLE database

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  ========================================== -->
<A NAME="Top_Of_Page"><H1>Resume:  Bruce Adamson<HR>
<H2><A NAME="ToC">Table of Contents<A NAME="ToC_1" HREF="#Header_1">
Resume:  Bruce Adamson<BR>
<A NAME="ToC_2" HREF="#Header_2">Personal Information<BR>
<A NAME="ToC_3" HREF="#Header_3">Department Information<BR>
<A NAME="ToC_4" HREF="#Header_4">Education<BR>
<A NAME="ToC_5" HREF="#Header_5">Work History<BR>
<A NAME="ToC_6" HREF="#Header_6">Interests<BR>
<HR><P>
<P>
<H3><A NAME="Header_1">Resume:  Bruce Adamson<P>
<H3><A NAME="Header_2">Personal Information<DL COMPACT>
<DT>Address:
<DD>3600 Steeles Ave
<BR>
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"
<DT>Weight:
<DD>175 lbs.<P>
<H3><A NAME="Header_3">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><A NAME="Header_4">Education<DL>
<P><DT>1971
<DD>Environmental Engineering, M.Sc.
<BR>
Johns Hopkins University
<P><DT>1968
<DD>American History, B.A.
<BR>
Northwestern University<P>
```

```
<H3><A NAME="Header_5">Work History<DL>
<P><DT>8/79 - present
<DD>Neural Network Design
<BR>
Developing neural networks for machine intelligence products.
<P><DT>2/72 - 7/79
<DD>Robot Vision Development
<BR>
Developing rule-based systems to emulate sight.
<P><DT>9/71 - 1/72
<DD>Numerical Integration Specialist
<BR>
Helping bank systems communicate with each other.<P>
<H3><A NAME="Header_6">Interests<UL COMPACT>
<LI>Racing motorcycles
<LI>Building loudspeakers
<LI>Assembling personal computers
<LI>Sketching<A NAME="Bot_Of_Page">
```

Walker photo
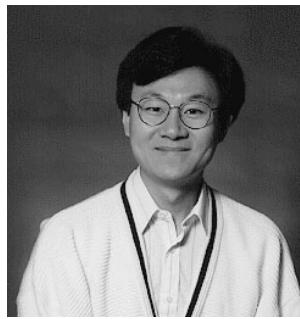
Figure 20. James H. Walker

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
- 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  ========================================= -->
<A NAME="Top_Of_Page"><H1>Resume:  James H. Walker<HR>
<H2><A NAME="ToC">Table of Contents<A NAME="ToC_1" HREF="#Header_1">
Resume:  James H. Walker<BR>
<A NAME="ToC_2" HREF="#Header_2">Personal Information<BR>
<A NAME="ToC_3" HREF="#Header_3">Department Information<BR>
<A NAME="ToC_4" HREF="#Header_4">Education<BR>
<A NAME="ToC_5" HREF="#Header_5">Work History<BR>
<A NAME="ToC_6" HREF="#Header_6">Interests<BR>
```

The SAMPLE database

```
<HR><P>
<P>
<H3><A NAME="Header_1">Resume:  James H. Walker<P>
<H3><A NAME="Header_2">Personal Information<DL COMPACT>
<DT>Address:
<DD>3500 Steeles Ave
<BR>
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><A NAME="Header_3">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><A NAME="Header_4">Education<DL>
<P><DT>1974
<DD>Computer Studies, B.Sc.
<BR>
University of Massachusetts
<P><DT>1972
<DD>Linguistic Anthropology, B.A.
<BR>
University of Toronto<P>
<H3><A NAME="Header_5">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">
```

Appendix F. Reserved schema names and reserved words

There are restrictions on the use of certain names that are required by the database manager.

In some cases, names are reserved, and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs, although their use is not prevented by the database manager.

The reserved schema names are:
- SYSCAT
- SYSFUN
- SYSIBM
- SYSIBMADM
- SYSPROC
- SYSPUBLIC
- SYSSTAT

It is strongly recommended that schema names never begin with the 'SYS' prefix, because 'SYS', by convention, is used to indicate an area that is reserved by the system. No aliases, global variables, triggers, user-defined functions, or user-defined types can be placed into a schema whose name starts with 'SYS' (SQLSTATE 42939).

The DB2QP schema and the SYSTOOLS schema are set aside for utilities used by the database. It is recommended that users not explicitly define objects in these schemas, although their use is not prevented by the database manager.

It is recommended that schema names never begin with the 'Q' prefix, because on other DB2 database managers 'Q', by convention, is used to indicate an area reserved by the system.

It is also recommended that SESSION not be used as a schema name. Because declared temporary tables must be qualified by SESSION, it is possible to have an application declare a temporary table with a name that is identical to that of a persistent table, complicating the application logic. To avoid this possibility, do not use the schema SESSION except when dealing with declared temporary tables.

Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement, unless it is delimited.

ISO/ANSI SQL2003 and other IBM database products include reserved words that are not enforced by DB2 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 IBM database products, the following words should be considered reserved words:

Reserved schema names and reserved words

ACTIVATE	DOUBLE	LOCATORS	ROLLBACK
ADD	DROP	LOCK	ROUND_CEILING
AFTER	DSSIZE	LOCKMAX	ROUND_DOWN
ALIAS	DYNAMIC	LOCKSIZE	ROUND_FLOOR
ALL	EACH	LONG	ROUND_HALF_DOWN
ALLOCATE	EDITPROC	LOOP	ROUND_HALF_EVEN
ALLOW	ELSE	MAINTAINED	ROUND_HALF_UP
ALTER	ELSEIF	MATERIALIZED	ROUND_UP
AND	ENABLE	MAXVALUE	ROUTINE
ANY	ENCODING	MICROSECOND	ROW
AS	ENCRYPTION	MICROSECONDS	ROWNUMBER
ASENSITIVE	END	MINUTE	ROWS
ASSOCIATE	END-EXEC	MINUTES	ROWSET
ASUTIME	ENDING	MINVALUE	ROW_NUMBER
AT	ERASE	MODE	RRN
ATTRIBUTES	ESCAPE	MODIFIES	RUN
AUDIT	EVERY	MONTH	SAVEPOINT
AUTHORIZATION	EXCEPT	MONTHS	SCHEMA
AUX	EXCEPTION	NAN	SCRATCHPAD
AUXILIARY	EXCLUDING	NEW	SCROLL
BEFORE	EXCLUSIVE	NEW_TABLE	SEARCH
BEGIN	EXECUTE	NEXTVAL	SECOND
BETWEEN	EXISTS	NO	SECONDS
BINARY	EXIT	NOCACHE	SECQTY
BUFFERPOOL	EXPLAIN	NOCYCLE	SECURITY
BY	EXTENDED	NODENAME	SELECT
CACHE	EXTERNAL	NODENUMBER	SENSITIVE
CALL	EXTRACT	NOMAXVALUE	SEQUENCE
CALLED	FENCED	NOMINVALUE	SESSION
CAPTURE	FETCH	NONE	SESSION_USER
CARDINALITY	FIELDPROC	NOORDER	SET
CASCADED	FILE	NORMALIZED	SIGNAL
CASE	FINAL	NOT	SIMPLE
CAST	FOR	NULL	SNAN
CCSID	FOREIGN	NULLS	SOME
CHAR	FREE	NUMPARTS	SOURCE
CHARACTER	FROM	OBID	SPECIFIC
CHECK	FULL	OF	SQL
CLONE	FUNCTION	OFFSET	SQLID
CLOSE	GENERAL	OLD	STACKED
CLUSTER	GENERATED	OLD_TABLE	STANDARD
COLLECTION	GET	ON	START
COLLID	GLOBAL	OPEN	STARTING
COLUMN	GO	OPTIMIZATION	STATEMENT
COMMENT	GOTO	OPTIMIZE	STATIC
COMMIT	GRANT	OPTION	STATMENT
CONCAT	GRAPHIC	OR	STAY
CONDITION	GROUP	ORDER	STOGROUP
CONNECT	HANDLER	OUT	STORES
CONNECTION	HASH	OUTER	STYLE
CONSTRAINT	HASHED_VALUE	OVER	SUBSTRING
CONTAINS	HAVING	OVERRIDING	SUMMARY
CONTINUE	HINT	PACKAGE	SYNONYM
COUNT	HOLD	PADDED	SYSFUN
COUNT_BIG	HOUR	PAGESIZE	SYSIBM
CREATE	HOURS	PARAMETER	SYSPROC
CROSS	IDENTITY	PART	SYSTEM
CURRENT	IF	PARTITION	SYSTEM_USER
CURRENT_DATE	IMMEDIATE	PARTITIONED	TABLE
CURRENT_LC_CTYPE	IN	PARTITIONING	TABLESPACE
CURRENT_PATH	INCLUDING	PARTITIONS	THEN
CURRENT_SCHEMA	INCLUSIVE	PASSWORD	TIME
CURRENT_SERVER	INCREMENT	PATH	TIMESTAMP
CURRENT_TIME	INDEX	PIECESIZE	TO
CURRENT_TIMESTAMP	INDICATOR	PLAN	TRANSACTION
CURRENT_TIMEZONE	INDICATORS	POSITION	TRIGGER
CURRENT_USER	INF	PRECISION	TRIM

CURSOR	INFINITY	PREPARE	TRUNCATE
CYCLE	INHERIT	PREVVAL	TYPE
DATA	INNER	PRIMARY	UNDO
DATABASE	INOUT	PRIQTY	UNION
DATAPARTITIONNAME	INSENSITIVE	PRIVILEGES	UNIQUE
DATAPARTITIONNUM	INSERT	PROCEDURE	UNTIL
DATE	INTEGRITY	PROGRAM	UPDATE
DAY	INTERSECT	PSID	USAGE
DAYS	INTO	PUBLIC	USER
DB2GENERAL	IS	QUERY	USING
DB2GENRL	ISOBID	QUERYNO	VALIDPROC
DB2SQL	ISOLATION	RANGE	VALUE
DBINFO	ITERATE	RANK	VALUES
DBPARTITIONNAME	JAR	READ	VARIABLE
DBPARTITIONNUM	JAVA	READS	VARIANT
DEALLOCATE	JOIN	RECOVERY	VCAT
DECLARE	KEEP	REFERENCES	VERSION
DEFAULT	KEY	REFERENCING	VIEW
DEFAULTS	LABEL	REFRESH	VOLATILE
DEFINITION	LANGUAGE	RELEASE	VOLUMES
DELETE	LATERAL	RENAME	WHEN
DENSERANK	LC_CTYPE	REPEAT	WHENEVER
DENSE_RANK	LEAVE	RESET	WHERE
DESCRIBE	LEFT	RESIGNAL	WHILE
DESCRIPTOR	LIKE	RESTART	WITH
DETERMINISTIC	LIMIT	RESTRICT	WITHOUT
DIAGNOSTICS	LINKTYPE	RESULT	WLM
DISABLE	LOCAL	RESULT_SET_LOCATOR	WRITE
DISALLOW	LOCALDATE	RETURN	XMLELEMENT
DISCONNECT	LOCALE	RETURNS	XMLEXISTS
DISTINCT	LOCALTIME	REVOKE	XMLNAMESPACES
DO	LOCALTIMESTAMP	RIGHT	YEAR
DOCUMENT	LOCATOR	ROLE	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	SQLEXCEPTION
CHAR_LENGTH	MERGE	SQLSTATE
CHARACTER_LENGTH	METHOD	SQLWARNING
CLOB	MIN	SQRT
COALESCE	MOD	STDDEV_POP
COLLATE	MODULE	STDDEV_SAMP
COLLECT	MULTISET	SUBMULTISET
CONVERT	NATIONAL	SUM
CORR	NATURAL	SYMMETRIC
CORRESPONDING	NCHAR	TABLESAMPLE
COVAR_POP	NCLOB	TIMEZONE_HOUR
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

Reserved schema names and reserved words
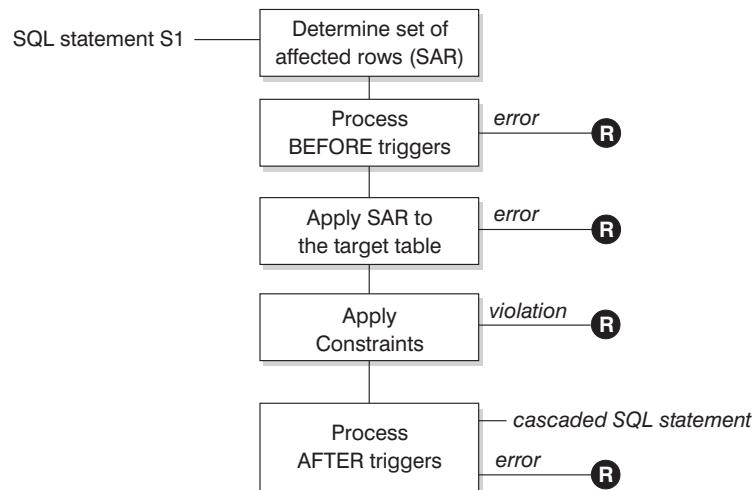
DEREF	PERCENTILE_CONT	UNNEST
ELEMENT	PERCENTILE_DISC	UPPER
EXEC	POWER	VAR_POP
EXP	REAL	VAR_SAMP
FALSE	RECURSIVE	VARCHAR
FILTER	REF	VARYING
FLOAT	REGR_AVGX	WIDTH_BUCKET
FLOOR	REGR_AVGY	WINDOW
FUSION	REGR_COUNT	WITHIN

Appendix G. Examples of interaction between triggers and referential constraints

Update operations can cause the interaction of triggers with referential constraints and check constraints.

Figure 21 and the associated description are representative of the processing that is performed for an statement that updates data in the database.

= rollback changes to before S1

Figure 21. Processing an statement with associated triggers and constraints

Figure 21 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 21.

- 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 can 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 can 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 might 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 can 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 can 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 before the execution of the original statement S_1

Appendix H. Explain tables

The Explain tables capture access plans when the Explain facility is activated.

The Explain tables must be created before Explain can be invoked. You can create them using one of the following methods:

- Call the SYSPROC.SYSINSTALLOBJECTS procedure:

```
CONNECT TO database-name
CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN', 'C',
        CAST (NULL AS VARCHAR(128)), CAST (NULL AS VARCHAR(128)))
```

  This call creates the explain tables under the SYSTOOLS schema. To create them under a different schema, specify a schema name as the last parameter in the call.

- Run the EXPLAIN.DDL command file:

```
CONNECT TO database-name
db2 -tf EXPLAIN.DDL
```

  This command file creates explain tables under the current schema. The location of this command file depends on the operating system, as illustrated in the following table:

Operating system	Location of the EXPLAIN command file
Linux AIX Solaris HP-UX	Located in the INSTHOME/sqllib/misc directory. INSTHOME is the instance home directory.
Windows	Located at the DB2PATH\misc directory on Windows operating systems. DB2PATH is the location where you install your DB2 copy

  Calling the SYSPROC.SYSINSTALLOBJECTS procedure is preferred over using the EXPLAIN.DDL file since it can automatically adapt to different database configurations. For example, if **BLOCKNONLOGGED** parameter is set to yes, then some statements in EXPLAIN.DDL fail because **NOT LOGGED** clause is used for LOB columns. However, if **BLOCKNONLOGGED** parameter is set to yes then the SYSPROC.SYSINSTALLOBJECTS procedure automatically avoids the use of **NOT LOGGED** clause.

The Explain facility uses the following IDs as the schema when qualifying Explain tables that it is populating:

- The session authorization ID for dynamic SQL
- The statement authorization ID for static SQL
- The SYSTOOLS schema if explain tables do not exist with the authorization ID schema

The schema can be associated with a set of Explain tables, or aliases that point to a set of Explain tables under a different schema. If no Explain tables are found under the schema, the Explain facility checks for Explain tables under the SYSTOOLS schema and attempts to use those tables.

Explain 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 251. 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.
SECTNO	INTEGER	No	No	Section number within package to which this explain information is related.
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
NAME	VARCHAR(128)	No	No	Name of the index.
CREATOR	VARCHAR(128)	No	No	Qualifier of the index name.
TBNAME	VARCHAR(128)	No	No	Name of the table or nickname on which the index is defined.
TBCREATOR	VARCHAR(128)	No	No	Qualifier of the table name.
COLNAMES	CLOB(2M)	No	No	List of column names.
UNIQUERULE	CHAR(1)	No	No	Unique rule: • D = Duplicates allowed • P = Primary index • U = Unique entries only allowed
COLCOUNT	SMALLINT	No	No	Number of columns in the key plus the number of include columns if any.
IID	SMALLINT	No	No	Internal index ID.
NLEAF	BIGINT	No	No	Number of leaf pages; -1 if statistics are not gathered.

ADVISE_INDEX table

Table 251. 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
NLEVELS	SMALLINT	No	No	Number of index levels; -1 if statistics are not gathered.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values; -1 if statistics are not gathered.
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values; -1 if statistics are not gathered.
CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if detailed statistics are gathered (in which case AVGPARTITION_CLUSTERFACTOR will be used instead).
AVGPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of the degree of clustering within a single data partition. -1 if the table is not table partitioned, if statistics are not gathered, or if the index is defined on a nickname.
AVGPARTITION_PAGE_ FETCH_PAIRS	VARCHAR(520)	No	No	A list of paired integers in character form. Each pair represents a potential buffer pool size and the corresponding page fetches required to access a single data partition from the table. Zero-length string if no data is available, or if the table is not table partitioned.
DATAPARTITION_ CLUSTERFACTOR	DOUBLE	No	No	A statistic measuring the "clustering" of the index keys with regard to data partitions. This field holds a number between zero and one, with one representing perfect clustering and zero representing no clustering.
CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of degree of clustering, or -1 if detailed index statistics have not been gathered or if the index is defined on a nickname.
USERDEFINED	SMALLINT	No	No	Defined by the user.

Table 251. 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

SYSTEM_REQUIRED	SMALLINT	No	No	• 1 if one or the other of the following conditions is met:
				  – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for a multi-dimensional clustering (MDC) table.
				  – This is an index on the (OID) column of a typed table.
				• 2 if both of the following conditions are met:
				  – This index is required for a primary or unique key constraint, or this index is a dimension block index or composite block index for an MDC table.
				  – This is an index on the (OID) column of a typed table.
				• 0 otherwise.

CREATE_TIME	TIMESTAMP	No	No	Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	No	Last time when any change was made to recorded statistics for this index. Null if no statistics available.
PAGE_FETCH_PAIRS	VARCHAR(520)	No	No	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
REMARKS	VARCHAR(254)	Yes	No	User-supplied comment, or null.
DEFINER	VARCHAR(128)	No	No	User who created the index.
CONVERTED	CHAR(1)	No	No	Reserved for future use.
SEQUENTIAL_PAGES	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. (-1 if no statistics are available.)
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, -1 if no statistics are available.)
FIRST2KEYCARD	BIGINT	No	No	Number of distinct keys using the first two columns of the index (-1 if no statistics or inapplicable)
FIRST3KEYCARD	BIGINT	No	No	Number of distinct keys using the first three columns of the index (-1 if no statistics or inapplicable)
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.

ADVISE_INDEX table

Table 251. 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
UNIQUE_COLCOUNT	SMALLINT	No	No	The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there a include columns. -1 if index has no unique key (permits duplicates)
MINPCTUSED	SMALLINT	No	No	If not zero, then online index defragmentation is enabled, and the value is the threshold of minimum used space before merging pages.
REVERSE_SCANS	CHAR(1)	No	No	• Y = Index supports reverse scans • N = Index does not support reverse scans
USE_INDEX	CHAR(1)	Yes	No	• Y = index recommended or evaluated • N = index not to be recommended • R = an existing clustering RID index was recommended (by the Design Advisor) to be unclustered; this is the case when a new clustering RID index is recommended for the table • I = Ignore an existing non-unique index. The EXISTS column should be 'Y' in this case or the index will not be ignored.
CREATION_TEXT	CLOB(2M)	No	No	The SQL statement used to create the index.
PACKED_DESC	BLOB(1M)	Yes	No	Internal description of the table.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.
INDEXTYPE	VARCHAR(4)	No	No	Type of index. • CLUS = Clustering • REG = Regular • DIM = Dimension block index • BLOK = Block index
EXISTS	CHAR(1)	No	No	Set to 'Y' if the index exists in the database catalog or 'N' if the index does not currently exist in the catalog.
RIDTOBLOCK	CHAR(1)	No	No	Set to 'Y' if the RID index was used to make a block index in the Design Advisor.
NULLKEYS	CHAR(1)	No	No	Specifies whether null keys are indexed. • N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause)

ADVISE_INSTANCE table

The ADVISE_INSTANCE table contains information about db2advis execution, including start time.

Contains one row for each execution of db2advis. Other ADVISE tables have a foreign key (RUN_ID) that links to the START_TIME column of the ADVISE_INSTANCE table for rows created during the same Design Advisor run.

Table 252. 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 253. 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.
STMTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	No	Statement number within package to which this Explain information is related.
NAME	VARCHAR(128)	No	No	MQT name.
CREATOR	VARCHAR(128)	No	No	MQT creator name.
IID	SMALLINT	No	No	Internal identifier.
CREATE_TIME	TIMESTAMP	No	No	Time at which the MQT was created.
STATS_TIME	TIMESTAMP	Yes	No	Time at which statistics were taken.
NUMROWS	DOUBLE	No	No	The number of estimated rows in the MQT.
NUMCOLS	SMALLINT	No	No	Number of columns defined in the MQT.
ROWSIZE	DOUBLE	No	No	Average length (in bytes) of a row in the MQT.
BENEFIT	FLOAT	No	No	Reserved for future use.
USE_MQT	CHAR(1)	Yes	No	Set to 'Y' when the MQT is recommended.
MQT_SOURCE	CHAR(1)	Yes	No	Indicates where the MQT candidate was generated. Set to 'I' if the MQT candidate is a refresh-immediate MQT, or 'D' if it can only be created as a full refresh-deferred MQT.
QUERY_TEXT	CLOB(2M)	No	No	Contains the query that defines the MQT.
CREATION_TEXT	CLOB(2M)	No	No	Contains the CREATE TABLE DDL for the MQT.
SAMPLE_TEXT	CLOB(2M)	No	No	Contains the sampling query that is used to get detailed statistics for the MQT. Only used when detailed statistics are required for the Design Advisor. The resulting sampled statistics will be shown in this table. If null, then no sampling query was created for this MQT.
COLSTATS	CLOB(2M)	No	No	Contains the column statistics for the MQT (if not null). These statistics are in XML format and include the column name, column cardinality and, optionally, the HIGH2KEY and LOW2KEY values.
EXTRA_INFO	BLOB(2M)	No	No	Reserved for miscellaneous output.

Table 253. 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
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 254. 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.

Table 254. ADVISE_PARTITION 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
USEIT	CHAR(1)	Yes	No	Specifies whether or not the database partition is used in EVALUATE PARTITION mode. A database partition is used if USEIT is set to 'Y' or 'y'.
RUN_ID	TIMESTAMP	Yes	FK	A value corresponding to the START_TIME of a row in the ADVISE_INSTANCE table, linking it to the same Design Advisor run.

ADVISE_TABLE table

The ADVISE_TABLE table stores the data definition language (DDL) for table creation, using the final Design Advisor recommendations for materialized query tables (MQTs), multidimensional clustered tables (MDCs), and database partitioning.

Table 255. 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 256. ADVISE_WORKLOAD Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
WORKLOAD_NAME	CHAR(128)	No	No	Name of the collection of SQL statements (workload) to which this statement belongs.
STATEMENT_NO	INTEGER	No	No	Statement number within the workload to which this explain information is related.
STATEMENT_TEXT	CLOB(1M)	No	No	Content of the SQL statement.
STATEMENT_TAG	VARCHAR(256)	No	No	Identifier tag for each explained SQL statement.
FREQUENCY	INTEGER	No	No	The number of times this statement appears within the workload.
IMPORTANCE	DOUBLE	No	No	Importance of the statement.
WEIGHT	DOUBLE	No	No	Priority of the statement.
COST_BEFORE	DOUBLE	Yes	No	The cost of the query (in timerons) if the recommendations are not created.
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 occured while trying to prepare the statement. If this column is NULL or does not start with SQLCA, the SQL query could be compiled by db2advis. If a compile error is found by db2advis or the Design Advisor, the COMPILABLE column value consists of an 8 byte long SQLCA.sqlcaid field, followed by a colon (:) and an 8 byte long SQLCA.sqlstate field, which is the return code for the SQL statement.

EXPLAIN_ACTUALS table

The EXPLAIN_ACTUALS table contains Explain section actuals information.

Table 257. EXPLAIN_ACTUALS Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this Explain information is related.
OPERATOR_ID	INTEGER	No	FK	Unique ID for this operator within this query.
DBPARTITIONNUM	INTEGER	No	No	The partition number of the database partition where the operator has run.
PREDICATE_ID	INTEGER	Yes	No	ID of the predicate applied on this operator. NULL if the actuals are operator actuals.
HOW_APPLIED	CHAR(10)	Yes	No	How predicate is used by this operator. NULL if PREDICATE_ID is NULL.
ACTUAL_TYPE	VARCHAR(12)	No	No	The type of the actual.
ACTUAL_VALUE	DOUBLE	Yes	No	The value of the actual. NULL if actual is not available for this operator.

EXPLAIN_ARGUMENT table

The EXPLAIN_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

Table 258. EXPLAIN_ARGUMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this Explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
ARGUMENT_TYPE	CHAR(8)	No	No	The type of argument for this operator.
ARGUMENT_VALUE	VARCHAR(1024)	Yes	No	The value of the argument for this operator. NULL if the value is in LONG_ARGUMENT_VALUE.
LONG_ARGUMENT_ VALUE	CLOB(2M)	Yes	No	The value of the argument for this operator, when the text will not fit in ARGUMENT_VALUE. NULL if the value is in ARGUMENT_VALUE.

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
AGGMODE	COMPLETE PARTIAL HASHED PARTIAL HASHED COMPLETE INTERMEDIATE FINAL COMPLETE UNIQUE PARTIAL UNIQUE INTERMEDIATE UNIQUE FINAL UNIQUE	Indicates how the operator aggregates values; for example, whether the aggregation is complete or partial aggregation. HASHED COMPLETE identifies a column-organized data grouping.
APREUSE	TRUE	Indicates if access plan reuse bind option is in effect for this statement.
BACKJOIN	TRUE FALSE	Indicates whether the ZZJOIN operator is used as a backjoin in the all-probe list-prefetch plan.
BITFLTR	INTEGER FALSE	Size of a hash join bit filter. A hash join bit filter can sometimes also be used by a table queue.
BLDLEVEL	Database build identifier	Internal identification string for source code version.

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
BLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT SHARE NONE SHARE UPDATE	Block level lock intent.
BUFFSORT	TRUE	Indicates whether SORT is used as a buffering operation.
BUSTSENS	YES NO	Indicates whether the BUSTIMESENSITIVE bind option is in effect for this statement.
BY DPART	TRUE FALSE	Indicates whether ZZJN is performed across the dimensions of a data partitioned table.

CONCACCR	Each row of this type will contain:
	
	• Level of the setting for this statement:
	
	  BIND
	
	    Application BIND with CONCURRENT ACCESS RESOLUTION option
	
	  PREP
	
	    Statement prepared with CONCURRENT ACCESS RESOLUTION attributes
	
	• The concurrent access resolution in effect:
	
	  USE CURRENTLY COMMITTED
	
	    Concurrent access resolution of application bind or statement prepare is USE CURRENTLY COMMITTED
	
	  WAIT FOR OUTCOME
	
	    Concurrent access resolution of application bind or statement prepare is WAIT FOR OUTCOME
		Indicates the concurrent access resolution used to generate the access plan for this statement.

CSERQY	TRUE FALSE	Remote query is a common subexpression.
CSETEMP	TRUE FALSE	Temporary Table over Common Subexpression Flag.
CUR_COMM	TRUE	Access currently committed rows when the value for the database configuration parameter **cur_commit** is not DISABLE. This access plan is enabled for applicable statements by using either: • CONCURRENT ACCESS RESOLUTION with the USE CURRENTLY COMMITTED option on bind or prepare • The database configuration parameter **cur_commit** with a value of ON

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
DEGREE	INTEGER	If the RETURN operator represents the return from column-organized data processing of the insertion, update, or deletion of rows, the DEGREE argument indicates the number of column-organized processing subagents that are used to process the insert, update, or delete operations in parallel.
DIRECT	TRUE	Direct fetch indicator.
DPESTFLG	TRUE FALSE	Indicates whether or not the DPNUMPRT value is based on an estimate. Possible values are 'TRUE' (DPNUMPRT represents the estimated number of accessed data partitions) or 'FALSE' (DPNUMPRT represents the actual number of accessed data partitions).
DPFXMLMV	REFERENCE COMBINATION	Indicates whether XML column data is moved between DPF partitions.
DPLSTPRT	NONE CHARACTER	Represents accessed data partitions. It is a comma-delimited list (for example: 1,3,5) or a hyphenated list (for example: 1-5) of accessed data partitions. A value of 'NONE' means that no data partition remains after specified predicates have been applied.
DPNUMPRT	INTEGER	Represents the actual or estimated number of data partitions accessed.
DSTSEVER	Server name	Destination (ship from) server.
DUPLWARN	TRUE FALSE	Duplicates Warning flag.
EARLYOUT	LEFT LEFT (REMOVE INNER DUPLICATES) RIGHT GROUPBY NONE	Early out indicator. LEFT indicates that each row from the outer table only needs to be joined with at most one row from the inner table. LEFT (REMOVE INNER DUPLICATES) indicates that an attempt to remove some duplicate rows from the inner table has taken place. RIGHT indicates that each row from the inner table only needs to be joined with at most one row from the outer table. NONE indicates no early out processing. GROUPBY indicates that early out processing is allowed because of a group by operation.
ENVVAR	Each row of this type will contain: • Environment variable name • Environment variable value	Environment variable affecting the optimizer
ERRTOL	Each row of this type will contain an SQLSTATE and SQLCODE pair.	A list of errors to be tolerated.
EXTROWS	Y N The argument does not show in the explain tables and formatted output when the value is 'N'.	Indicates that the maximum row size of the system temporary table might be too large to fit in a 32K page. Some rows might need to be represented as a large object (LOB).

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
EVALUNCO	TRUE	Evaluate uncommitted data using lock deferral. This is enabled with the **DB2_EVALUNCOMMITTED** registry variable.
EXECUTID	An opaque binary token formatted as an hexadecimal string representing the executable ID.	Indicates the executable ID of the section being explained.
FETCHMAX	IGNORE INTEGER	Override value for MAXPAGES argument on FETCH operator.
FLTRAPPL	TQ PUSHDOWN	Indicates the bit filter application method used by the optimizer. A value of 'TQ PUSHDOWN' indicates that the bit filter operation has been pushed down. This argument will not be included at all when the optimizer does not use a pushdown with the hash join.
GREEDY	TRUE	Indicates whether the optimizer used a greedy algorithm to plan access.
GLOBLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE NO LOCK OBTAINED SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Represents global lock intent information for a partitioned table object.
GROUPBYC	TRUE FALSE	Whether Group By columns were provided. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.
GROUPBYN	Integer	Number of comparison columns. This number may be less than the number of columns present in the GROUP BY clause of the SQL statement if predicates eliminated the need to compare some columns. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.
GROUPBYR	Each row of this type will contain: • Ordinal value of column in group by clause (followed by a colon and a space) • Name of column	Group By requirement. This argument can be associated with a GRPBY operator or with a TEMP operator when it is part of a query with multiple distinct aggregations.
GROUPS	Integer	Number of times the operator will repeat.
HASHCODE	24 32	Size (in bits) of hash join hash code used for hash joins. A hash join hash code can sometimes also be used by a table queue.

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
HASHTBSZ	INTEGER	The number of expected entries in the hash table of a hash join.
IDXMSTLY	TRUE	Indicates whether the FETCH is performed over block identifiers returned from a multi dimensional clustered index.
IDXOVTMP	TRUE FALSE	Indicates whether the scan builds an index or a fast integer sort structure for random access of the temporary tables. If value is 'TRUE', the scan builds an index over the temporary tables for random access of the temporary tables. If value is 'FALSE', the scan builds a fast integer sort structure for random access of the temporary tables.

INNERCOL	Each row of this type will contain:
	• Ordinal value of column in order (followed by a colon and a space)
	• Name of column
	• Order value
	  (A) Ascending
	  (D) Descending
		Inner order columns.

INPUTXID	A context node identifier	INPUTXID identifies the input context node used by the XSCAN operator.
ISCANMAX	IGNORE INTEGER	Override value for MAXPAGES argument on ISCAN operator.
JN INPUT	INNER OUTER	Indicates if operator is the operator feeding the inner or outer of a join.
JUMPSCAN	TRUE FALSE	Indicates that the index scan is a jump scan.
LCKAVOID	TRUE	Lock avoidance: row access will avoid locking committed data.
LISTENER	TRUE FALSE	Listener Table Queue indicator.
MAXPAGES	ALL NONE INTEGER	Maximum pages expected for Prefetch.
MAXRIDS	NONE INTEGER	Maximum Row Identifiers to be included in each list prefetch request.

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
MXPPSCAN	TRUE FALSE	Provides additional information about how MAXPAGES is calculated in the case of a jump scan. A jump scan can be conceptualized as multiple contiguous scans separated by jumps. If the value is 'TRUE', then MAXPAGES is the number of pages that are expected to be accessed by each contiguous scan individually. If the value is 'FALSE', then MAXPAGES is the number of pages that are expected to be accessed by all the contiguous scans in total.
NUMROWS	INTEGER	Number of rows expected to be sorted.
ONEFETCH	TRUE FALSE	Indicates the one fetch GROUP BY clause. The GROUP BY conditions are satisfied for the first row given the implied order of the input stream.
OPROFERR	TRUE FALSE	Indicates that one or more errors occurred while parsing or applying the optimization profile. For details, see explain diagnostic messages.

OUTERCOL	Each row of this type will contain:
	• Ordinal value of column in order (followed by a colon and a space)
	• Name of column
	• Order value
	  (A) Ascending
	  (D) Descending
		Outer order columns.

OUTERJN	LEFT RIGHT FULL LEFT (ANTI) RIGHT (ANTI)	Outer join indicator.
OVERHEAD	DOUBLE	Optimizer used OVERHEAD value.
PARTCOLS	Name of Column	Partitioning columns for operator.
PBLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan table lock intent.

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
PGLOLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan global table lock intent.
PLANID	Hexadecimal string representing a BIGINT value	Identifier uniquely representing a query plan configuration for a given statement. The layout of the operators, accessed objects and relevant operator arguments and other plan properties affecting performance are represented by this value.
PREFETCH	DYNAMIC LIST LIST NONE READAHEAD SEQUENTIAL SEQUENTIAL, READAHEAD	Type of prefetch eligible.
PFTCHSZ	INTEGER	Optimizer used PREFETCHSIZE value.
PROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Positioning scan row lock intent.
PTABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Positioning scan table lock intent.
RAND ACC	TRUE	Indicates that the regular TEMP table allows random access. Random access is required for the ZZJN operator.
REOPT	ALWAYS ONCE	The statement is optimized using bind-in values for parameter markers, host variables, and special registers.
RMTQTXT	Query text	Remote Query Text
RNG_PROD	Function name	Range producing function for extended index access.
ROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Row Lock Intent.
ROWWIDTH	INTEGER	Width of row to be sorted.

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
RSUFFIX	Query text	Remote SQL suffix.
SCANDIR	FORWARD REVERSE	Scan Direction.
SCANGRAN	INTEGER	Intrapartition parallelism, granularity of the intrapartition parallel scan, expressed in SCANUNITs.
SCANTYPE	LOCAL PARALLEL	Intrapartition parallelism, index scan, table scan, or column-organized data scan.
SCANUNIT	ROW PAGE	Intrapartition parallelism, scan granularity unit.
SEMEVID	Hexadecimal string representing a BIGINT value	Identifier for semantic environment at the time the statement was compiled. SEMEVID corresponds to monitoring element SEMANTIC_ENV_ID.
SEMIJOIN	TRUE FALSE	This argument on the HSJOIN operator indicates whether column-organized processing should deploy the semi-join optimization technique to reduce the size of the hash table that is created for large hash join inner tables.
SHARED	TRUE	Intrapartition parallelism, shared TEMP indicator.
SHRCSE	TRUE	Temporary table over common subexpression shared between subsections.
SKIP_INS	TRUE	Skip inserted. Row access will skip uncommitted inserted rows. This behavior is enabled with the **DB2_SKIPINSERTED** registry variable or when currently committed semantics are in effect.
SKIPDKEY	TRUE	Skip deleted keys. Row access will skip uncommitted deleted keys. This behavior is enabled with the **DB2_SKIPDELETED** registry variable.
SKIPDROW	TRUE	Skip deleted rows. Row access will skip uncommitted deleted rows. This behavior is enabled with the **DB2_SKIPDELETED** registry variable.
SKIPLOCK	TRUE	The concurrent access resolution "skip locked data" is in effect.
SLOWMAT	TRUE FALSE	Slow Materialization flag.
SNGLPROD	TRUE FALSE	Intrapartition parallelism sort or temp produced by a single agent.

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description

SORTKEY	Each row of this type will contain:
	
	• Ordinal value of column in key (followed by a colon and a space)
	
	• Name of column
	
	• Order value
	
	  (A) Ascending
	
	  (D) Descending
		Sort key columns.

SORTTYPE	PARTITIONED SHARED ROUND ROBIN REPLICATED	Intrapartition parallelism, sort type.
SRCSEVER	Server name	Source (ship to) server.
SPEED	SLOW FAST	'SLOW' indicates that the scan is expected to progress slowly over the table. For example, if the scan is the outer of a nested loop join). 'FAST' indicates that the scan is expected to progress with higher speed. This information is used to group scans together for efficient sharing of bufferpool records.
SPILLED	INTEGER	Estimated number of pages in SORT spill.
SQLCA	Warning information	Warnings and reason codes issued during Explain operation.
STARJOIN	YES	The IXAND operator is part of a star join
STMTHEAP	INTEGER	Size of statement heap at start of statement compile.
STMTID	Hexadecimal string representing a BIGINT value	Identifier uniquely representing a normalized form of the SQL statement. The statement normalization follows the optimization profile's inexact matching rules.
STREAM	TRUE FALSE	Remote source is streaming.
SYSTSENS	YES NO	Indicates that the SYSTIMESENSITIVE bind option is in effect
TABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Table Lock Intent.
TBISOLVL	UNCOMMITED READ READ STABILITY CURSOR STABILITY REPEATABLE READ	Indicates the isolation level used by the operator to access the specific table
TEMPSIZE	INTEGER	Temporary table page size.

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
THROTTLE	TRUE FALSE	Throttling improves the performance of other scans that would otherwise lag behind and be forced to reread the same pages. 'TRUE' if the scan can be throttled. 'FALSE' if the scan must not be throttled.
TMPCMPRS	YES ELIGIBLE	The value YES indicates that compression is applied. The value ELIGIBLE indicates that compression may be applied if the table becomes large enough. The absence of TMPCMPRS indicates that the temporary table is not compressed.
TQDEGREE	INTEGER	Intrapartition parallelism, number of subagents accessing Table Queue. If the TQ operator represents the transition between column-organized data processing and row-organized data processing, the TQDEGREE argument indicates the number of column-organized processing subagents that are used to process the query in parallel.
TQMERGE	TRUE FALSE	Merging (sorted) Table Queue indicator.
TQREAD	READ AHEAD STEPPING SUBQUERY STEPPING	Table Queue reading property.
TQSEND	BROADCAST DIRECTED SCATTER SUBQUERY DIRECTED	Table Queue send property.
TQ TYPE	LOCAL	Intrapartition parallelism, Table Queue.
TQORIGIN	ASYNCHRONY XTQ COLUMN-ORGANIZED DATA ROW-ORGANIZED DATA	The reason that Table Queue was introduced into the access plan. `COLUMN-ORGANIZED DATA` indicates that the TQ operator is being used to transfer data from column-organized processing to row-organized processing. `ROW-ORGANIZED DATA` indicates that the TQ operator is being used to transfer data from row-organized processing to column-organized processing. `COLUMN-ORGANIZED DATA` or `ROW-ORGANIZED DATA` values result in the TQ operator being displayed as CTQ in the access plan.
TRUNCTQ	INPUT OUTPUT INPUT AND OUTPUT	Truncated Table Queue indicator. INPUT indicates that truncation occurs on input to the Table Queue. OUPUT indicates that truncation occurs on output from the Table Queue. INPUT and OUTPUT indicates that truncation occurs on both input to the Table Queue and on output from the Table Queue.

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
TRUNCSRT	TRUE	Truncated sort (limits number of rows produced).
TUPBLKSZ	INTEGER	Component of the total sort heap required to perform a hash join that determines the number of bytes that a tuple will be stored in. This can be used by service to diagnose memory, temporary table and to some degree sort heap usage.
UNIQUE	TRUE FALSE HASHED PARTIAL HASHED COMPLETE	This operator eliminates rows having duplicate values for a set of columns. HASHED PARTIAL indicates that a partial early distinct operation was performed to efficiently remove many, if not all, duplicates. This reduces the amount of data that must be processed later in the query evaluation. HASHED COMPLETE indicates that hashing is used to eliminate duplicates during column-organized data processing.
UNIQKEY	Each row of this type will contain: • Ordinal value of column in key (followed by a colon and a space) • Name of Column	Unique key columns.
UR_EXTRA	TRUE	Uncommitted read isolation, but with extra processing to ensure correct isolation. This access has extra table level locking; the same table level locking as cursor stability. Also, when the statement is executing, the isolation level might upgrade to cursor stability, for example, if an online load is running concurrently. Another part of the statement execution plan will ensure the isolation level is correct, such as a FETCH operator at a higher isolation level.
USAGE	SCALAR_SUBQUERY	Indicates how the NLJOIN operator is being used. SCALAR_SUBQUERY indicates that the NLJOIN operator is being used to apply a scalar subquery predicate. Hash join is the only join method used for column-organized tables. However, the optimizer uses the NLJOIN operator to model the application of scalar subqueries during column-organized data processing. One of the input legs to the NLJOIN operator is the single value that is computed by the scalar subquery. The other input to the operator is the data stream whose predicate references the scalar value.

EXPLAIN_ARGUMENT table

Table 259. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
VISIBLE	TRUE FALSE	Whether shared scans are visible to other shared scans. A shared scan that is visible can influence the behavior of other scans. Examples of affected behavior include start location and throttling.
VOLATILE	TRUE	Volatile table
WRAPPING	TRUE FALSE	Whether a shared scan is allowed to start at any record in the table and wrap once it reaches the last record. Wrapping allows bufferpool records to be shared with other ongoing scans.
XFERRATE	DOUBLE	Optimizer used TRANSFERRATE value.
XDFOUT	DECIMAL	XDFOUT indicates the expected number of documents to be returned by the XISCAN operator for each context node.
XLOGID	An identifier consisting of an SQL schema name and the name of an index over XML data	XLOGID identifies the index over XML data chosen by the optimizer for the XISCAN operator.
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 260. EXPLAIN_DIAGNOSTIC Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK, FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK, FK	Level of Explain information for which this row is relevant. Valid values are: **O** Original text (as entered by user) **P** PLAN SELECTION
STMTNO	INTEGER	No	PK, FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	PK, FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
CODE	INTEGER	No	No	A unique number assigned to each diagnostic message. The number can be used by a message API to retrieve the full text of the diagnostic message.

EXPLAIN_DIAGNOSTIC_DATA table

The EXPLAIN_DIAGNOSTIC_DATA table contains message tokens for specific diagnostic messages that are recorded in the EXPLAIN_DIAGNOSTIC table. The message tokens provide additional information that is specific to the execution of the SQL statement that generated the message.

The EXPLAIN_GET_MSGS table function queries the EXPLAIN_DIAGNOSTIC and EXPLAIN_DIAGNOSTIC_DATA Explain tables, and returns formatted messages.

Table 261. EXPLAIN_DIAGNOSTIC_DATA Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant. Valid values are: **O** Original text (as entered by user) **P** PLAN SELECTION
STMTNO	INTEGER	No	FK	Statement number within package to which this Explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	FK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at run time. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
DIAGNOSTIC_ID	INTEGER	No	PK	ID of the diagnostic for a particular instance of a statement in the EXPLAIN_STATEMENT table.
ORDINAL	INTEGER	No	No	Position of token in the full message text.
TOKEN	VARCHAR(1000)	Yes	No	Message token to be inserted into the full message text; might be truncated.
TOKEN_LONG	BLOB(3M)	Yes	No	More detailed information, if available.

EXPLAIN_FORMAT_OUTPUT table

When db2exfmt utility formats explain tables, its output sometimes contains symbols around operators.

The following tables list the various symbols that are used with the operators of the ASCII explain graph.

Table 262. Symbols used with JOIN operator

Character before operator	Character after operator	Equivalent Explain argument	Argument value	Description
>		OUTERJN	LEFT	Left outer join
	<	OUTERJN	RIGHT	Right outer join
>	<	OUTERJN	FULL	Full outer join
x		OUTERJN	LEFT (ANTI)	Left outer anti-join
	x	OUTERJN	RIGHT (ANTI)	Right outer anti-join
^		EARLYOUT	LEFT	Left early out
	^	EARLYOUT	RIGHT	Right early out

Table 263. Symbols used with TQ operator

Character before operator	Character after operator	Equivalent Explain argument	Argument value	Description
A		TQORIGIN	ASYNCHRONY	Table queue that is used for asynchrony
B		TQSEND	BROADCAST	All rows sent to all receiving database partitions
C		TQORIGIN	'COLUMN-ORGANIZED DATA' or 'ROW-ORGANIZED DATA'	Represents the transition between column-organized data processing and row-organized data processing
D		TQSEND	DIRECTED	Rows are hashed to each of the receiving database partitions by using the table queue 's partitioning key
L		TQTYPE	LOCAL	Intra-partition parallelism
M		TQMERGE	TRUE	Merging (sorted) table queue

Table 263. Symbols used with TQ operator (continued)

Character before operator	Character after operator	Equivalent Explain argument	Argument value	Description
X		TQORIGIN	XML AGG TQ	Table queue that is used for XML aggregation
S		TQSEND	SCATTER	Rows are scattered to receiving database partitions without using a partitioning key
	*	LISTENER	TRUE	Table queue waits to receive values from other database partitions

Table 264. Unique symbols per key

Character before operator	Operator	Equivalent Explain argument	Argument value	Description
p	UNIQUE	UNIQUE	HASHED PARTIAL	Partial early aggregation
p	GRPBY	AGGMODE	HASHED PARTIAL	Per group aggregation
g	<many>	GROUPS		Multiple distinct processing

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 265. 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_OPTION	CHAR(1)	No	No	Indicates what Explain Information was requested for this request. Possible values are: **P** PLAN SELECTION **S** Section Explain
SNAPSHOT_TAKEN	CHAR(1)	No	No	Indicates whether an Explain Snapshot was taken for this request. Possible values are: **Y** Yes, an Explain Snapshot(s) was taken and stored in the EXPLAIN_STATEMENT table. Regular Explain information was also captured. **N** No Explain Snapshot was taken. Regular Explain information was captured. **O** Only an Explain Snapshot was taken. Regular Explain information was not captured.
DB2_VERSION	CHAR(7)	No	No	Release number for the DB2 product that processed this explain request. Format is *vv.rr.m*, where: **vv** Version number **rr** Release number **m** Maintenance release number

EXPLAIN_INSTANCE table

Table 265. 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
SQL_TYPE	CHAR(1)	No	No	Indicates whether the Explain Instance was for static or dynamic SQL.
				Possible values are:
				S Static SQL
				D Dynamic SQL
QUERYOPT	INTEGER	No	No	Indicates the query optimization class used by the SQL Compiler at the time of the Explain invocation. The value indicates what level of query optimization was performed by the SQL Compiler for the SQL statements being explained.
BLOCK	CHAR(1)	No	No	Indicates what type of cursor blocking was used when compiling the SQL statements.
				Possible values are:
				N No Blocking
				U Block Unambiguous Cursors
				B Block All Cursors
ISOLATION	CHAR(2)	No	No	Indicates what type of isolation was used when compiling the SQL statements.
				Possible values are:
				RR Repeatable Read
				RS Read Stability
				CS Cursor Stability
				UR Uncommitted Read
BUFFPAGE	INTEGER	No	No	Contains the value of the **buffpage** database configuration setting at the time of the Explain invocation. **Important:** The **buffpage** database configuration is deprecated and might be remove in a future release.
AVG_APPLS	INTEGER	No	No	Contains the value of the **avg_appls** configuration parameter at the time of the Explain invocation.
SORTHEAP	INTEGER	No	No	Contains the value of the **sortheap** database configuration parameter at the time of the Explain invocation.
LOCKLIST	INTEGER	No	No	Contains the value of the **locklist** database configuration parameter at the time of the Explain invocation.
MAXLOCKS	SMALLINT	No	No	Contains the value of the **maxlocks** database configuration parameter at the time of the Explain invocation.
LOCKS_AVAIL	INTEGER	No	No	Contains the number of locks assumed to be available by the optimizer for each user. (Derived from **locklist** and **maxlocks**.)

Table 265. 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
CPU_SPEED	DOUBLE	No	No	Contains the value of the **cpuspeed** database manager configuration parameter at the time of the Explain invocation.
REMARKS	VARCHAR(254)	Yes	No	User-provided comment.
DBHEAP	INTEGER	No	No	Contains the value of the **dbheap** database configuration parameter at the time of Explain invocation.
COMM_SPEED	DOUBLE	No	No	Contains the value of the **comm_bandwidth** database configuration parameter at the time of Explain invocation.
PARALLELISM	CHAR(2)	No	No	Possible values are: • N = No parallelism • P = Intrapartition parallelism • IP = Interpartition parallelism • BP = Intrapartition parallelism and interpartition parallelism
DATAJOINER	CHAR(1)	No	No	Possible values are: • N = Non-federated systems plan • Y = Federated systems plan
EXECUTABLE_ID	VARCHAR(32) FOR BIT DATA	Yes	No	A binary token generated on the data server that uniquely identifies the SQL statement section that was executed.
EXECUTION_TIME	TIMESTAMP	Yes	No	Time the section started execution.

EXPLAIN_OBJECT table

The EXPLAIN_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

Table 266. 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 266. 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 • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
TRANSFER_RATE	DOUBLE	No	No	Estimated time to read a data page, in milliseconds, from the specified table space. Set to -1 if: • no table space is involved • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
PREFETCHSIZE	INTEGER	No	No	Number of data pages to be read when prefetch is performed. Set to -1 if: • no table space is involved • a partitioned object is involved and the respective table spaces for the partitions have different values • a partitioned object is involved and the EXPLAIN_OBJECT table is populated using section explain
EXTENTSIZE	INTEGER	No	No	Size of extent, in data pages. This many pages are written to one container in the table space before switching to the next container. Set to -1 for a table function.
CLUSTER	DOUBLE	No	No	Degree of data clustering with the index. If >= 1, this is the CLUSTERRATIO. If >= 0 and < 1, this is the CLUSTERFACTOR. Set to -1 for a table, table function, or if this statistic is not available.
NLEAF	BIGINT	No	No	Number of leaf pages this index object's values occupy. Set to -1 for a table, table function, or if this statistic is not available.
NLEVELS	INTEGER	No	No	Number of index levels in this index object's tree. Set to -1 for a table, table function, or if this statistic is not available.
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.

EXPLAIN_OBJECT table

Table 266. 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
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values. Set to -1 for a table, table function, or if this statistic is not available.
FIRST2KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 2 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
FIRST3KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 3 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
FIRST4KEYCARD	BIGINT	No	No	Number of distinct first key values using the first 4 columns of the index. Set to -1 for a table, table function, or if this statistic is not available.
SEQUENTIAL_PAGES	BIGINT	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. Set to -1 for a table, table function, or if this statistic is not available.
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percentage (integer between 0 and 100). Set to -1 for a table, table function, or if this statistic is not available.
STATS_SRC	CHAR(1)	No	No	Indicates the source for the statistics. Set to 1 if from single node.
AVERAGE_SEQUENCE_ GAP	DOUBLE	No	No	Gap between sequences.
AVERAGE_SEQUENCE_ FETCH_GAP	DOUBLE	No	No	Gap between sequences when fetching using the index.
AVERAGE_SEQUENCE_ PAGES	DOUBLE	No	No	Average number of index pages accessible in sequence.
AVERAGE_SEQUENCE_ FETCH_PAGES	DOUBLE	No	No	Average number of table pages accessible in sequence when fetching using the index.
AVERAGE_RANDOM_ PAGES	DOUBLE	No	No	Average number of random index pages between sequential page accesses.
AVERAGE_RANDOM_ FETCH_PAGES	DOUBLE	No	No	Average number of random table pages between sequential page accesses when fetching using the index.
NUMRIDS	BIGINT	No	No	Total number of row identifiers in the index.
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 266. 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
NULLKEYS	CHAR(1)	Yes	No	Specifies whether null keys are indexed. • N = Keys that contain all null values are not indexed (not considering columns or expressions from the INCLUDE clause) • Y = Keys that contain all null values are indexed (not considering columns or expressions from the INCLUDE clause) A null value indicates that this object is not an index.

Table 267. Possible OBJECT_TYPE Values

Value	Description
IX	Index
NK	Nickname
RX	RCT Index
DP	Data partitioned table
TA	Table
TF	Table Function
+A	Compiler-referenced Alias
+C	Compiler-referenced Constraint
+F	Compiler-referenced Function
+G	Compiler-referenced Trigger
+N	Compiler-referenced Nickname
+T	Compiler-referenced Table
+V	Compiler-referenced View
XI	Logical XML index
PI	Physical XML index
LI	Partitioned index
LX	Partitioned logical XML index
LP	Partitioned physical XML index
CO	Column-organized table

EXPLAIN_OPERATOR table

The EXPLAIN_OPERATOR table contains all the operators needed to satisfy the query statement by the query compiler.

Table 268. EXPLAIN_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	PK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	PK	Section number within package to which this explain information is related.
OPERATOR_ID	INTEGER	No	PK	Unique ID for this operator within this query.
OPERATOR_TYPE	CHAR(6)	No	No	Descriptive label for the type of operator.
TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of executing the chosen access plan up to and including this operator.
IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of executing the chosen access plan up to and including this operator.
CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of executing the chosen access plan up to and including this operator.
FIRST_ROW_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the first row for the access plan up to and including this operator. This value includes any initial overhead required.
RE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.
RE_IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of fetching the next row for the chosen access plan up to and including this operator.
RE_CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of fetching the next row for the chosen access plan up to and including this operator.
COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost (in TCP/IP frames) of network traffic flowing across a single network adapter when executing the chosen access plan up to and including this operator. (See notes 1 and 2.)

Table 268. 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
FIRST_COMM_COST	DOUBLE	No	No	Estimated cumulative communications cost (in TCP/IP frames) of network traffic flowing across a single network adapter when fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required. (See notes 1 and 2.)
BUFFERS	DOUBLE	No	No	Estimated buffer requirements for this operator and its inputs.
REMOTE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of performing operation(s) on remote database(s).
REMOTE_COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost of executing the chosen remote access plan up to and including this operator.

Note:

1. If there is more than one network adapter involved, the cumulative communication cost for the adapter with the highest value is returned.
2. This value only includes the costs of network traffic between physical machines. It does not include the virtual communication costs between node partitions on the same physical machine in a partitioned database environment.

Table 269. 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
REBAL	Rebalance rows between SMP subagents
RETURN	Result
RIDSCN	Row Identifier (RID) Scan
RPD	Remote PushDown
SHIP	Ship query to remote system
SORT	Sort
TBFUNC	In-stream table function operator
TBSCAN	Table Scan

EXPLAIN_OPERATOR table

Table 269. OPERATOR_TYPE values (continued)

Value	Description
TEMP	Temporary Table Construction
TQ	Table Queue
UNION	Union
UNIQUE	Duplicate Elimination
UPDATE	Update
XISCAN	Index scan over XML data
XSCAN	XML document navigation scan
XANDOR	Index ANDing and ORing over XML data
ZZJOIN	Zigzag join

EXPLAIN_PREDICATE table

The EXPLAIN_PREDICATE table identifies which predicates are applied by a specific operator.

Table 270. 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: **blank** This predicate does not contain a subquery. **EAA** The subquery used in this predicate is evaluated at application (EAA). That is, it is re-evaluated for every row processed by the specified operator, as the predicate is being applied. **EAO** The subquery used in this predicate is evaluated at open (EAO). That is, it is re-evaluated only once for the specified operator, and its results are re-used in the application of the predicate for each row. **MUL** There is more than one type of subquery in this predicate.
RELOP_TYPE	CHAR(2)	No	No	The type of relational operator used in this predicate.

EXPLAIN_PREDICATE table

Table 270. 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	Whether or not a data stream from a subquery is required for this predicate. There may be multiple subquery streams required. Possible values are: **N** No subquery stream is required **Y** One or more subquery streams is required
FILTER_FACTOR	DOUBLE	No	No	The estimated fraction of rows that will be qualified by this predicate. A value of "-1" is shown when FILTER_FACTOR is not applicable. FILTER_FACTOR is not applicable for operator predicates constructed by the Explain tool which are not optimizer objects and do not exist in the optimizer plan.
PREDICATE_TEXT	CLOB(2M)	Yes	No	The text of the predicate as recreated from the internal representation of the SQL or XQuery statement. If the value of a host variable, special register, or parameter marker is used during compilation of the statement, this value will appear at the end of the predicate text enclosed in a comment. The value will be stored in the EXPLAIN_PREDICATE table only if the statement is executed by a user who has DBADM authority, or if the DB2_VIEW_REOPT_VALUES registry variable is set to YES; otherwise, an empty comment will appear at the end of the predicate text. Null if not available.
RANGE_NUM	INTEGER	Yes	No	Range of data partition elimination predicates, which enables the grouping of predicates that are used for data partition elimination by range. Null value for all other predicate types.
INDEX_CLOSEQ	INTEGER	No	No	Indicates the index column that the predicate belongs to if it is part of a key predicate. A key predicate always belongs to one index key part. A value of "-1" is shown for predicates that are not part of a key predicate.

Table 271. 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

Table 271. Possible HOW_APPLIED Values (continued)

Value	Description
SARG	Evaluated as a sargable predicate for index or data page
GAP_START	Used as a start condition on an index gap
GAP_STOP	Used as a stop condition on an index gap
START	Used as a start condition
STOP	Used as a stop condition
FEEDBACK	Zigzag join feedback predicate

Table 272. Possible RELOP_TYPE Values

Value	Description
blanks	Not Applicable
EQ	Equals
GE	Greater Than or Equal
GT	Greater Than
IN	In list
IC	In list, sorted during query optimization
IR	In list, sorted at runtime
LE	Less Than or Equal
LK	Like
LT	Less Than
NE	Not Equal
NL	Is Null
NN	Is Not Null

EXPLAIN_STATEMENT table

The EXPLAIN_STATEMENT table contains the text of the SQL statement as it exists for the different levels of Explain information.

The original SQL statement as entered by the user is stored in this table along with the version used (by the optimizer) to choose an access plan to satisfy the SQL statement. The latter version may bear little resemblance to the original as it may have been rewritten or enhanced with additional predicates as determined by the SQL Compiler. In addition, if statement concentrator is enabled and the statement was changed as a result of statement concentrator, the effective SQL statement will also be stored in this table. This statement will resemble the original statement except that the literal values will be replaced with system generated named parameter markers. The plan information will be based on the effective statement in this case.

Table 273. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	PK, FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant. Valid values are: **E** Effective SQL text **F** Statement with row and column access control applied, before optimization **O** Original Text (as entered by user) **P** PLAN SELECTION **S** Section Explain
STMTNO	INTEGER	No	PK	Statement number within package to which this explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	PK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at runtime. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.

Table 273. 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
QUERYNO	INTEGER	No	No	Numeric identifier for explained SQL statement. For dynamic SQL statements (excluding the EXPLAIN SQL statement) issued through CLP or CLI, the default value is a sequentially incremented value. Otherwise, the default value is the value of STMTNO for static SQL statements and 1 for dynamic SQL statements.
QUERYTAG	CHAR(20)	No	No	Identifier tag for each explained SQL statement. For dynamic SQL statements issued through CLP (excluding the EXPLAIN SQL statement), the default value is 'CLP'. For dynamic SQL statements issued through CLI (excluding the EXPLAIN SQL statement), the default value is 'CLI'. Otherwise, the default value used is blanks.
STATEMENT_TYPE	CHAR(2)	No	No	Descriptive label for type of query being explained. Possible values are: **CL** Call **CP** Compound SQL (Dynamic) **D** Delete **DC** Delete where current of cursor **I** Insert **M** Merge **S** Select **SI** Set Integrity or Refresh Table **U** Update **UC** Update where current of cursor
UPDATABLE	CHAR(1)	No	No	Indicates if this statement is considered updatable. This is particularly relevant to SELECT statements which may be determined to be potentially updatable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes
DELETABLE	CHAR(1)	No	No	Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes

EXPLAIN_STATEMENT table

Table 273. 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
TOTAL_COST	DOUBLE	No	No	Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is 0 or E (original or effective text) since no access plan has been chosen at this time.
STATEMENT_TEXT	CLOB(2M)	No	No	Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection or Section Explain levels of Explain has been reconstructed from the internal representation and is SQL-like in nature; that is, the reconstructed statement is not guaranteed to follow correct SQL syntax.
SNAPSHOT	BLOB(10M)	Yes	No	Snapshot of internal representation for this SQL statement at the Explain_Level shown. Column is set to NULL if EXPLAIN_LEVEL is not P (Plan Selection) since no access plan has been chosen at the time that this specific version of the statement is captured.
QUERY_DEGREE	INTEGER	No	No	Indicates the degree of intrapartition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intrapartition parallelism. Otherwise, this contains the degree of intrapartition parallelism generated for the plan to use.

EXPLAIN_STREAM table

The EXPLAIN_STREAM table represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are to be found in the EXPLAIN_OPERATOR table.

Table 274. EXPLAIN_STREAM Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
STREAM_ID	INTEGER	No	No	Unique ID for this data stream within the specified operator.
SOURCE_TYPE	CHAR(1)	No	No	Indicates the source of this data stream: **O** Operator **D** Data Object
SOURCE_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the source of this data stream. Set to -1 if SOURCE_TYPE is 'D'.
TARGET_TYPE	CHAR(1)	No	No	Indicates the target of this data stream: **O** Operator **D** Data Object
TARGET_ID	SMALLINT	No	No	Unique ID for the operator within this query that is the target of this data stream. Set to -1 if TARGET_TYPE is 'D'.
OBJECT_SCHEMA	VARCHAR(128)	Yes	No	Schema to which the affected data object belongs. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
OBJECT_NAME	VARCHAR(128)	Yes	No	Name of the object that is the subject of data stream. Set to null if both SOURCE_TYPE and TARGET_TYPE are 'O'.
STREAM_COUNT	DOUBLE	No	No	Estimated cardinality of data stream.
COLUMN_COUNT	SMALLINT	No	No	Number of columns in data stream.
PREDICATE_ID	INTEGER	No	No	If this stream is part of a subquery for a predicate, the predicate ID will be reflected here, otherwise the column is set to -1.

EXPLAIN_STREAM table

Table 274. 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
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.
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: **MULT** On multiple database partitions **COOR** On coordinator node **HASH** Directed using hashing **RID** Directed using the row ID **FUNC** Directed using a function (HASHEDVALUE() or DBPARTITIONNUM()) **CORR** Directed using a correlation value **Numeric** Directed to predetermined single node
PARTITION_COLUMNS	CLOB(2M)	Yes	No	List of the columns on which this data stream is distributed.
SEQUENCE_SIZES	CLOB(2M)	Yes	No	Lists the expected sequence size for XML columns, or shows "NA" (not applicable) for any non-XML columns in the data stream. Set to null if there is not at least one XML column in the data stream.

OBJECT_METRICS table

The OBJECT_METRICS table contains runtime statistics collected for each object referenced in a specific execution of a section (identified by executable ID) at a specific time (identified by execution time).

If object statistics are collected on multiple database members, there will be a row per database member for each object. If it is a partitioned object, then there will be one row per data partition.

The OBJECT_METRICS table is populated only if section actuals were captured by the activity event monitor.

Table 275. OBJECT_METRICS table

Column Name	Data Type	Nullable?	Key?	Description
EXECUTABLE_ID	VARCHAR(32) FOR BIT DATA	No	PK	A binary token generated on the data server that uniquely identifies the SQL statement section that was executed.
EXECUTION_TIME	TIMESTAMP	No	PK	Time the section started execution.
OBJECT_SCHEMA	VARCHAR(128)	No	PK	Schema to which this object belongs.
OBJECT_NAME	VARCHAR(128)	No	PK	Name of the object.
OBJECT_TYPE	CHAR(2)	No	PK	Descriptive label for the type of object. Possible values: **IX** Index **DP** Data partitioned table **TA** Table **PI** Physical XML Index **LI** Partitioned index **LP** Partitioned physical XML index
MEMBER	SMALLINT	No	PK	The database member where the object statistics is collected.
DATA_PARTITION_ID	INTEGER	No	PK	The identifier of the data partition for which information is returned . This element is applicable only to partitioned tables or indexes.
ROWS_READ	BIGINT	Yes	No	The total number of rows read.
ROWS_INSERTED	BIGINT	Yes	No	The total number of row insertions attempted.
ROWS_UPDATED	BIGINT	Yes	No	The total number of row updates attempted.
ROWS_DELETED	BIGINT	Yes	No	The total number of row deletions attempted.
OVERFLOW_CREATES	BIGINT	Yes	No	The number of overflowed rows created on this table object.
OVERFLOW_ACCESSES	BIGINT	Yes	No	The number of accesses (reads and writes) to overflowed rows of this table object.
LOCK_WAIT_TIME	BIGINT	Yes	No	The total elapsed time spent waiting for local locks. The value is given in milliseconds.
LOCK_WAIT_TIME_ GLOBAL	BIGINT	Yes	No	The total elapsed time spent waiting for global locks. The value is given in milliseconds.
LOCK_WAITS	BIGINT	Yes	No	The total number of times that the section waited on locks.

OBJECT_METRICS table

Table 275. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
LOCK_WAITS_ GLOBAL	BIGINT	Yes	No	The total number of times that the section waited on global locks.
LOCK_ESCALS	BIGINT	Yes	No	The total number of times that local locks have been escalated.
LOCK_ESCALS_ GLOBAL	BIGINT	Yes	No	The total number of times that global locks have been escalated.
DIRECT_WRITES	BIGINT	Yes	No	The total number of write operations that do not use a buffer pool.
DIRECT_WRITE_REQS	BIGINT	Yes	No	The total number of requests to perform a direct write of one or more sectors of data.
DIRECT_READS	BIGINT	Yes	No	The total number of read operations that do not use a buffer pool.
DIRECT_READ_REQS	BIGINT	Yes	No	The total number of requests to perform a direct read of one or more sectors of data.
OBJECT_DATA_ L_READS	BIGINT	Yes	No	Indicates the number of data pages which have been requested for the table (logical).
OBJECT_DATA_ P_READS	BIGINT	Yes	No	Indicates the number of data pages read in for the table (physical).
OBJECT_DATA_ GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page was attempted to be read for the table from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).
OBJECT_DATA_ GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page was read for the table into the local buffer pool from disk because it was not found in the group buffer pool.
OBJECT_DATA_ GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times a data page for XML Storage (XDAs) was attempted to be read for the table from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_DATA_ LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times a data page for the table was present in the local buffer pool.
OBJECT_DATA_ GBP_INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent data pages found in the local buffer pool (LBP) by the agent.
OBJECT_XDA_ L_READS	BIGINT	Yes	No	Indicates the number of data pages for XML Storage (XDAs) which have been requested for the table (logical).
OBJECT_XDA_ P_READS	BIGINT	Yes	No	Indicates the number of data pages for XML storage (XDAs) read in for the table (physical).
OBJECT_XDA_ GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page for XML Storage (XDAs) was attempted to be read for the table from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).

Table 275. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
OBJECT_XDA_ GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent data page for XML Storage (XDAs) was read for the table into the local buffer pool from disk because it was not found in the GBP.
OBJECT_XDA_ GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times a data page was attempted to be read for the table from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_XDA_ LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times a XML Storage (XDAs) data page for the table was present in the local buffer pool.
OBJECT_XDA_ GBP_INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent XML storage object (XDA) data pages found in the local buffer pool (LBP) by the agent.
OBJECT_INDEX_ L_READS	BIGINT	Yes	No	Indicates the number of index pages which have been requested for the index (logical).
OBJECT_INDEX_ P_READS	BIGINT	Yes	No	Indicates the number of index pages read in for the index (physical).
OBJECT_INDEX_ GBP_L_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent index page was attempted to be read for the index from the group buffer pool because the page was either invalid or not present in the Local Buffer Pool (LBP).
OBJECT_INDEX_ GBP_P_READS	BIGINT	Yes	No	The number of times a Group Buffer Pool (GBP) dependent index page was read for the index into the local buffer pool from disk because it was not found in the GBP.
OBJECT_INDEX_ GBP_INVALID_PAGES	BIGINT	Yes	No	The number of times an index page was attempted to be read for the index from the group buffer pool because the page was invalid in the local buffer pool.
OBJECT_INDEX_ LBP_PAGES_FOUND	BIGINT	Yes	No	The number of times an index page for the index was present in the local buffer pool.
OBJECT_INDEX_ GBP_INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent index pages found in the local buffer pool (LBP) by the agent.
OBJECT_COL_L_READS	BIGINT	Yes	No	The number of column-organized pages that are logically read from the buffer pool for a table.
OBJECT_COL_P_READS	BIGINT	Yes	No	The number of column-organized pages that are physically read for a table.
OBJECT_COL_GBP_ L_READS	BIGINT	Yes	No	The number of times that a group buffer pool (GBP) dependent column-organized page is requested from the GBP for a table. The page is requested because a valid version of the page does not exist in the local buffer pool (LBP).
OBJECT_COL_GBP_ P_READS	BIGINT	Yes	No	The number of times that a group buffer pool (GBP) dependent column-organized page is read into the local buffer pool (LBP) from disk for a table. The page is read from disk into the LBP because the page is not in the GBP.

OBJECT_METRICS table

Table 275. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
OBJECT_COL_GBP_ INVALID_PAGES	BIGINT	Yes	No	The number of times that a column-organized page is requested from the group buffer pool (GBP) for a table. The page is requested because the version of the page in the local buffer pool (LBP) is invalid. Outside of a DB2 pureScale environment, this value is null.
OBJECT_COL_LBP_ PAGES_FOUND	BIGINT	Yes	No	The number of times that a column-organized page for a table is present in the local buffer pool (LBP).
OBJECT_COL_GBP_ INDEP_PAGES_ FOUND_IN_LBP	BIGINT	Yes	No	The number of group buffer pool (GBP) independent column-organized pages found in a local buffer pool (LBP) by an agent.
OBJECT_DATA_ CACHING_TIER_ L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_DATA_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_ CACHING_TIER_ L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_XDA_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_INDEX_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.

Table 275. OBJECT_METRICS table (continued)

Column Name	Data Type	Nullable?	Key?	Description
OBJECT_COL_ CACHING_TIER_ L_READS	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ GBP_INVALID_PAGES	BIGINT	Yes	No	Reserved for future use.
OBJECT_COL_ CACHING_TIER_ GBP_INDEP_ PAGES_FOUND	BIGINT	Yes	No	Reserved for future use.

Appendix I. Explain register values

The tables in this topic describe the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values, both with each other and with the PREP and BIND commands.

With dynamic SQL, the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact as follows.

Table 276. Interaction of Explain Special Register Values (Dynamic SQL)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	**YES**	**EXPLAIN**	**REOPT**	**RECOMMEND INDEXES**	**EVALUATE INDEXES**
NO	• Results of query returned.	• Explain tables populated. • Results of query returned.	• Explain tables populated. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.
YES	• Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.

Explain register values

Table 276. Interaction of Explain Special Register Values (Dynamic SQL) (continued)

EXPLAIN SNAPSHOT values	EXPLAIN MODE values					
	NO	YES	EXPLAIN	REOPT	RECOMMEND INDEXES	EVALUATE INDEXES
EXPLAIN	• Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed).	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken. • Results of query not returned (dynamic statements not executed). • Indexes evaluated.
REOPT	• Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed).	• Explain tables populated when a statement qualifies for reoptimization at execution time. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed). • Indexes recommended.	• Explain tables populated. • Explain Snapshot taken when a statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic or incremental-bind statements not executed). • Indexes evaluated.

The CURRENT EXPLAIN MODE special register interacts with the EXPLAIN bind option in the following way for dynamic SQL.

Table 277. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	**YES**	**REOPT**	**ALL**
NO	• Results of query returned.	• Explain tables populated for static SQL. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.
YES	• Explain tables populated for dynamic SQL. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query returned.
EXPLAIN	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed).	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed).

Explain register values

Table 277. Interaction of EXPLAIN Bind Option and CURRENT EXPLAIN MODE (continued)

EXPLAIN MODE values	EXPLAIN Bind option values			
	NO	YES	REOPT	ALL
REOPT	• Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.
RECOMMEND INDEXES	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed). • Recommend indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Recommend indexes.
EVALUATE INDEXES	• Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL when statement qualifies for reoptimization at execution time. • Explain tables populated for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.	• Explain tables populated for static SQL. • Explain tables populated for dynamic SQL. • Results of query not returned (dynamic statements not executed). • Evaluate indexes.

The CURRENT EXPLAIN SNAPSHOT special register interacts with the EXPLSNAP bind option in the following way for dynamic SQL.

Table 278. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	YES	REOPT	ALL
NO	• Results of query returned.	• Explain Snapshot taken for static SQL. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.
YES	• Explain Snapshot taken for dynamic SQL. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query returned.
EXPLAIN	• Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query not returned (dynamic statements not executed).	• Explain Snapshot taken for static SQL. • Explain Snapshot taken for dynamic SQL. • Results of query not returned (dynamic statements not executed).

Explain register values

Table 278. Interaction of EXPLSNAP bind Option and CURRENT EXPLAIN SNAPSHOT (continued)

EXPLAIN SNAPSHOT values	EXPLSNAP Bind option values			
	NO	YES	REOPT	ALL
REOPT	• Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.	• Explain Snapshot taken for static SQL when statement qualifies for reoptimization at execution time. • Explain Snapshot taken for dynamic SQL when statement qualifies for reoptimization at execution time. • Results of query returned.

Appendix J. Exception tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using the SET INTEGRITY statement with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables that are used by the load utility are identical to the ones described here, and can therefore be reused during checking with the SET INTEGRITY statement.

Rules for creating an exception table

The rules for creating an exception table are as follows:
- If the table is protected by a security policy, the exception table must be protected by the same security policy.
- The first "n" columns of the exception table are the same as the columns of the table being checked. All column attributes, including name, data type, and length should be identical. For protected columns, the security label protecting the column must be the same in both tables.
- All of the columns of the exception table must be free of constraints and triggers. Constraints include referential integrity and check constraints, as well as unique index constraints that could cause errors on insert.
- The "(n+1)" column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET INTEGRITY statement on the same table, if the rows within the exception table have not been deleted before issuing the SET INTEGRITY statement to check the data. The timestamp precision can be any value from 0 to 12 and the value assigned will be the result of CURRENT TIMESTAMP special register
- The "(n+2)" column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
- The exception table should be created with both "(n+1)" and "(n+2)" columns.
- There is no enforcement of any particular name for the previously listed additional columns. However, the type specification must be exactly followed.
- No additional columns are allowed.
- If the original table has generated columns (including the IDENTITY property), the corresponding columns in the exception table should not specify the generated property.
- Users invoking the SET INTEGRITY statement to check data must hold the INSERT privilege on the exception tables.
- The exception table cannot be a data partitioned table, a range clustered table, or a detached table.
- The exception table cannot be a materialized query table or a staging table.
- The exception table cannot have any dependent refresh immediate materialized query tables or any dependent propagate immediate staging tables.

Exception tables

The information in the "message" column has the following structure:

Table 279. Exception Table Message Column Structure

Field number	Contents	Size	Comments
1	Number of constraint violations	5 bytes	Right justified padded with '0'
2	Type of first constraint violation	1 byte	'D' - Delete Cascade violation 'F' - Foreign Key violation 'G' - Generated Column violation 'I' - Unique Index violation[a] 'K' - Check Constraint violation 'L' - LBAC Write rules violation 'P' - Data Partitioning violation 'S' - Invalid Row Security Label 'X' - Index defined on XML column violation[d]
3	Length of constraint/column[b] /index ID[c]	5 bytes	Right justified padded with '0'
4	Constraint name/Column name[b]/index ID[c]	length from the previous field	
5	Separator	3 bytes	\<space>\<colon>\<space>
6	Type of next constraint violation	1 byte	'D' - Delete Cascade violation 'F' - Foreign Key violation 'G' - Generated Column violation 'I' - Unique Index violation 'K' - Check Constraint violation 'L' - LBAC Write rules violation 'P' - Data Partitioning violation 'S' - Invalid Row Security Label 'X' - Index defined on XML column violation[d]
7	Length of constraint/column/index ID	5 bytes	Right justified padded with '0'
8	Constraint name/Column name/Index ID	length from the previous field	
.....	.....	.....	Repeat Field 5 through 8 for each violation

- [a] Unique index violations will not occur during checking using the SET INTEGRITY statement, unless it is after an attach operation. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. However, LOAD will not report check constraint, generated column, foreign key, delete cascade, or data partitioning violations in the exception tables.

- [b] To retrieve the expression of a generated column from the catalog views, use a select statement. For example, if field 4 is MYSCHEMA.MYTABLE.GEN_1, then SELECT SUBSTR(TEXT, 1, 50) FROM SYSCAT.COLUMNS WHERE TABSCHEMA='MYSCHEMA' AND TABNAME='MYNAME' AND COLNAME='GEN_1'; will return the first fifty bytes of the expression, in the form "AS (\<expression>)"

- [c] To retrieve an index ID from the catalog views, use a select statement. For example, if field 4 is 1234, then SELECT INDSCHEMA, INDNAME FROM SYSCAT.INDEXES WHERE IID=1234.

- [d] For Index defined on XML column violations, the constraint name, column name, or index ID field identifies the XML column that had an integrity violation in one of its indexes. It does not identify the index that had the integrity violation. It identifies only the name of the XML column on which the index violation occurs. For example, the value 'X00006XTCOL2' in the message column indicates an index violation occurred in one of the indexes on the XTCOL2 column.

Handling rows in an exception table

The information in exception tables can be processed in various ways. Data can be corrected and rows re-inserted into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers, and you want to complete the load operation with the corrected rows from exception tables without firing the triggers:

- Design the INSERT triggers to be fired depending on the value in a column that has been defined explicitly for the purpose.
- Unload data from the exception tables and append it using the load utility. In this case, if you want to recheck the data, note that constraints checking is not confined to the appended rows.
- Save the trigger definition text from the relevant system catalog view. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally, re-create the trigger using the saved trigger definition.

No explicit provision is made to prevent the firing of triggers when inserting rows from exception tables.

Only one violation per row is reported for unique index violations.

If values with LONG VARCHAR, LONG VARGRAPHIC, or LOB data types are in the table, the values are not inserted into the exception table in the case of unique index violations.

Querying exception tables

The message column structure in an exception table is a concatenated list of constraint names, lengths, and delimiters, as described earlier. This information can be queried.

For example, to retrieve a list of all violations, repeating each row with only the constraint name, assume that the original table T1 had two columns, C1 and C2. Assume also, that the corresponding exception table, E1, has columns C1 and C2, corresponding to those in T1, as well as a message column, MSGCOL. The following query uses recursion to list one constraint name per row (repeating rows that have more than one violation):

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
     CHAR(SUBSTR(MSGCOL, 12,
               INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
     1,
     15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
   FROM E1
 UNION ALL
 SELECT C1, C2, MSGCOL,
     CHAR(SUBSTR(MSGCOL, J+6,
               INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
     I+1,
     J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
   FROM IV
   WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV;
```

To list all of the rows that violated a particular constraint, the previous query could be extended as follows:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
     CHAR(SUBSTR(MSGCOL, 12,
               INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
```

```
         1,
         15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
     FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, J+6,
                   INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
       I+1,
       J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
     FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';
```

The following query could be used to obtain all of the check constraint violations:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
  (SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, 12,
                   INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, 6, 1)),
       1,
       15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
     FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, J+6,
                   INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, J, 1)),
       I+1,
       J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
     FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';
```

Appendix K. SQL statements that can be executed in routines and triggers

Successful execution of SQL statements in routines is subject to restrictions and conditional on certain prerequisites being met. However, it is possible to execute many SQL statements in routines and triggers.

If a statement invokes a routine, the effective SQL data access level for the statement will be the greater of:

- The SQL data access level of the statement from the following table.
- The SQL data access level of the routine specified when the routine was created.

For example, the CALL statement has an SQL data access level of CONTAINS SQL. However, if a stored procedure defined as READS SQL DATA is called, the effective SQL data access level for the CALL statement is READS SQL DATA.

The following table lists all supported SQL statements, including SQL PL control-statements, and identifies if each SQL statement can be executed within the various types of routines. For each SQL statement listed in the first column, each of the subsequent columns shows an X to indicate if the statement is executable within the routine. The final column identifies the minimum SQL access level required to allow the statement execution to succeed. 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 procedure defined as MODIFIES SQL DATA. Unless otherwise noted in a footnote, all of the SQL statements may be executed either statically or dynamically.

Table 280. SQL statements that can be executed in routines

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
ALLOCATE CURSOR	X		X	X	MODIFIES SQL DATA
ALTER {BUFFERPOOL, DATABASE PARTITION GROUP, FUNCTION, METHOD, NICKNAME, PROCEDURE, SEQUENCE, SERVER, TABLE, TABLESPACE, TYPE, USER MAPPING, VIEW}			X	X	MODIFIES SQL DATA
ASSOCIATE LOCATORS	X				

SQL statements that can be executed in routines and triggers

Table 280. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
AUDIT			X	X	MODIFIES SQL DATA
BEGIN DECLARE SECTION			X	X	NO SQL(3)
CALL	X	X	X	X	CONTAINS SQL(12)
CASE	X	X			CONTAINS SQL
CLOSE	X		X	X	READS SQL DATA
COMMENT ON	X		X	X	MODIFIES SQL DATA
COMMIT	X(6)		X(6)		MODIFIES SQL DATA
Compound SQL	X	X	X	X	CONTAINS SQL
CONNECT(2)					
CREATE {ALIAS, BUFFERPOOL, DATABASE PARTITION GROUP, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING, GLOBAL TEMPORARY TABLE(11), INDEX(11), INDEX EXTENSION, METHOD, NICKNAME, PROCEDURE, SCHEMA, SEQUENCE, SERVER, TABLE(11), TABLESPACE, TRANSFORM, TRIGGER, TYPE, TYPE MAPPING, USER MAPPING, VIEW(11), WRAPPER }	X (8)		X		MODIFIES SQL DATA
DECLARE CURSOR	X	X	X		NO SQL(3)

Table 280. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
DECLARE GLOBAL TEMPORARY TABLE	X		X	X	MODIFIES SQL DATA
DELETE	X	X	X	X	MODIFIES SQL DATA
DESCRIBE(9)			X	X	READS SQL DATA
DISCONNECT(4)					
DROP	X(8)		X	X	MODIFIES SQL DATA
END DECLARE SECTION			X	X	NO SQL(3)
EXECUTE	X		X	X	CONTAINS SQL(5)
EXECUTE IMMEDIATE	X		x	X	CONTAINS SQL(5)
EXPLAIN	X		X	X	MODIFIES SQL DATA
FETCH	X		X	X	READS SQL DATA
FREE LOCATOR			X	X	CONTAINS SQL
FLUSH EVENT MONITOR			X	X	MODIFIES SQL DATA
FLUSH PACKAGE CACHE			X	X	MODIFIES SQL DATA
FOR	X	X			READS SQL DATA
FREE LOCATOR	X		X	X	CONTAINS SQL
GET DIAGNOSTICS	X	X			READS SQL DATA
GOTO	X	X			CONTAINS SQL
GRANT	X		X	X	MODIFIES SQL DATA
IF	X	X			CONTAINS SQL
INCLUDE			X	X	NO SQL
INSERT	X	X	X	X	MODIFIES SQL DATA
ITERATE	X	X			CONTAINS SQL
LEAVE	X	X			CONTAINS SQL
LOCK TABLE	X		X	X	CONTAINS SQL
LOOP	X	X			CONTAINS SQL

SQL statements that can be executed in routines and triggers

Table 280. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
MERGE	X	X	X	X	MODIFIES SQL DATA
OPEN	X		X	X	READS SQL DATA(7)
PREPARE	X		X	X	CONTAINS SQL
REFRESH TABLE			X	X	MODIFIES SQL DATA
RELEASE(4)					
RELEASE SAVEPOINT	X		X	X	MODIFIES SQL DATA
RENAME TABLE			X	X	MODIFIES SQL DATA
RENAME TABLESPACE			X	X	MODIFIES SQL DATA
REPEAT	X	X			CONTAINS SQL
RESIGNAL	X				MODIFIES SQL DATA
RETURN	X				CONTAINS SQL
REVOKE			X	X	MODIFIES SQL DATA
ROLLBACK(6)	X		X		
ROLLBACK TO SAVEPOINT	X		X	X	MODIFIES SQL DATA
SAVEPOINT	X				MODIFIES SQL DATA
select-statement	X		X	X	READS SQL DATA
SELECT INTO	X		X(10)	X(10)	READS SQL DATA(7)
SET CONNECTION(4)					
SET INTEGRITY			X		MODIFIES SQL DATA
SET special register	X	X	X	X	CONTAINS SQL
SET variable	X	X			CONTAINS SQL
SIGNAL	X	X			MODIFIES SQL DATA
TRANSFER OWNERSHIP			X	X	MODIFIES SQL DATA
TRUNCATE			X	X	MODIFIES SQL DATA
UPDATE	X	X	X		MODIFIES SQL DATA

Table 280. SQL statements that can be executed in routines (continued)

SQL statement	Executable in compound SQL (compiled) statements(1)	Executable in compound SQL (inlined) statements(2)	Executable in external procedures	Executable in external functions	Minimum required SQL data access level
VALUES INTO	X		X	X	READS SQL DATA
WHENEVER	X		X		NO SQL(3)
WHILE	X	X			

Note:

1. Compound SQL (compiled) statements can be used as the body of SQL procedures, SQL functions, triggers, or as stand-alone statements.

2. Compound SQL (inline) statements can be used as the body of SQL functions, SQL methods, triggers, or as stand-alone statements.

3. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.

4. Connection management statements are not allowed in any routine execution context.

5. This situation depends on the statement being executed. The statement specified for the EXECUTE statement must be 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.

6. 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, a COMMIT or a ROLLBACK of a unit of work is not allowed.

7. If the SQL access level READS SQL DATA is in effect, no SQL data change statement can be embedded in the SELECT INTO statement, or in the cursor referenced by the OPEN statement.

8. SQL procedures can only issue CREATE and DROP statements for indexes, tables, and views.

9. The DESCRIBE SQL statement has a different syntax than that of the CLP DESCRIBE command.

10. This is only supported for embedded SQL routines.

11. When referenced in an SQL procedure, the statement can only be executed statically.

12. The procedure that is called must have the same or more restrictive level of SQL data access than the current level in effect. For example, a routine defined as MODIFIES SQL DATA can call a procedure defined as MODIFIES SQL DATA, READS SQL DATA, CONTAINS SQL, or NO SQL. A routine defined as CONTAINS SQL can call a procedure defined as CONTAINS SQL or NO SQL. The arguments specified for the procedure might also require a different data access level. For example, a scalar fullselect as an argument would require the data access level for the statement to be READS SQL DATA.

Errors

Table 280 on page 1225 indicates whether the SQL statement specified by the first column is allowed to execute inside a routine that has the specified SQL data access level. If the statement exceeds the data access level, an error is returned when the routine is executed.

- If an executable SQL statement is encountered inside a routine defined with the NO SQL data access level, then SQLSTATE 38001 is returned.
- For other execution contexts, the SQL statements that are unsupported in any context return an SQLSTATE 38003 error.
- For other SQL statements that are 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 level returns an SQLSTATE 42985 error.

Appendix L. DB2 technical information

DB2 technical information is available in multiple formats that can be accessed in multiple ways.

DB2 technical information is available through the following tools and methods:
- Online DB2 documentation in IBM Knowledge Center:
  - Topics (task, concept, and reference topics)
  - Sample programs
  - Tutorials
- Locally installed DB2 Information Center:
  - Topics (task, concept, and reference topics)
  - Sample programs
  - Tutorials
- DB2 books:
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - Printed books
- Command-line help:
  - Command help
  - Message help

Important: The documentation in IBM Knowledge Center and the DB2 Information Center is updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 documentation in IBM Knowledge Center.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

The DB2 Information Development team values 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. The DB2 Information Development team reads all of your feedback but cannot respond to you directly. Provide specific examples wherever possible to 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 the db2docs@ca.ibm.com email address to contact DB2 Customer Support. If you have a DB2 technical issue that you cannot resolve by using the documentation, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

You can download the DB2 technical library in PDF format or you can order in hardcopy from the IBM Publications Center.

English and translated DB2 Version 10.5 manuals in PDF format can be downloaded from DB2 database product documentation at www.ibm.com/support/docview.wss?rs=71&uid=swg27009474.

The following tables describe the DB2 library available from the IBM Publications Center at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. Although the tables identify books that are available in print, the books might not be available in your country or region.

The form number increases each time that a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed in the following tables.

The DB2 documentation online in IBM Knowledge Center is updated more frequently than either the PDF or the hardcopy books.

Table 281. DB2 technical information

Name	Form number	Available in print	Availability date
Administrative API Reference	SC27-5506-00	Yes	28 July 2013
Administrative Routines and Views	SC27-5507-01	No	1 October 2014
Call Level Interface Guide and Reference Volume 1	SC27-5511-01	Yes	1 October 2014
Call Level Interface Guide and Reference Volume 2	SC27-5512-01	No	1 October 2014
Command Reference	SC27-5508-01	No	1 October 2014
Database Administration Concepts and Configuration Reference	SC27-4546-01	Yes	1 October 2014
Data Movement Utilities Guide and Reference	SC27-5528-01	Yes	1 October 2014
Database Monitoring Guide and Reference	SC27-4547-01	Yes	1 October 2014
Data Recovery and High Availability Guide and Reference	SC27-5529-01	No	1 October 2014
Database Security Guide	SC27-5530-01	No	1 October 2014
DB2 Workload Management Guide and Reference	SC27-5520-01	No	1 October 2014
Developing ADO.NET and OLE DB Applications	SC27-4549-01	Yes	1 October 2014
Developing Embedded SQL Applications	SC27-4550-00	Yes	28 July 2013

Table 281. DB2 technical information (continued)

Name	Form number	Available in print	Availability date
Developing Java Applications	SC27-5503-01	No	1 October 2014
Developing Perl, PHP, Python, and Ruby on Rails Applications	SC27-5504-01	No	1 October 2014
Developing RDF Applications for IBM Data Servers	SC27-5505-00	Yes	28 July 2013
Developing User-defined Routines (SQL and External)	SC27-5501-00	Yes	28 July 2013
Getting Started with Database Application Development	GI13-2084-01	Yes	1 October 2014
Getting Started with DB2 Installation and Administration on Linux and Windows	GI13-2085-01	Yes	1 October 2014
Globalization Guide	SC27-5531-00	No	28 July 2013
Installing DB2 Servers	GC27-5514-01	No	1 October 2014
Installing IBM Data Server Clients	GC27-5515-01	No	1 October 2014
Message Reference Volume 1	SC27-5523-00	No	28 July 2013
Message Reference Volume 2	SC27-5524-00	No	28 July 2013
Net Search Extender Administration and User's Guide	SC27-5526-01	No	1 October 2014
Partitioning and Clustering Guide	SC27-5532-01	No	1 October 2014
pureXML Guide	SC27-5521-00	No	28 July 2013
Spatial Extender User's Guide and Reference	SC27-5525-00	No	28 July 2013
SQL Procedural Languages: Application Enablement and Support	SC27-5502-00	No	28 July 2013
SQL Reference Volume 1	SC27-5509-01	No	1 October 2014
SQL Reference Volume 2	SC27-5510-01	No	1 October 2014
Text Search Guide	SC27-5527-01	Yes	1 October 2014
Troubleshooting and Tuning Database Performance	SC27-4548-01	Yes	1 October 2014
Upgrading to DB2 Version 10.5	SC27-5513-01	Yes	1 October 2014
What's New for DB2 Version 10.5	SC27-5519-01	Yes	1 October 2014
XQuery Reference	SC27-5522-01	No	1 October 2014

Table 282. DB2 Connect technical information

Name	Form number	Available in print	Availability date
Installing and Configuring DB2 Connect Servers	SC27-5517-00	Yes	28 July 2013
DB2 Connect User's Guide	SC27-5518-01	Yes	1 October 2014

Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure

To start 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 DB2 documentation online for different DB2 versions

You can access online the documentation for all the versions of DB2 products in IBM Knowledge Center.

About this task

All the DB2 documentation by version is available in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSEPGG/welcome. However, you can access a specific version by using the associated URL for that version.

Procedure

To access online the DB2 documentation for a specific DB2 version:

- To access the DB2 Version 10.5 documentation, follow this URL: http://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.kc.doc/welcome.html.
- To access the DB2 Version 10.1 documentation, follow this URL: http://www.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.kc.doc/welcome.html.
- To access the DB2 Version 9.8 documentation, follow this URL: http://www.ibm.com/support/knowledgecenter/SSEPGG_9.8.0/com.ibm.db2.luw.kc.doc/welcome.html.
- To access the DB2 Version 9.7 documentation, follow this URL: http://www.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.kc.doc/welcome.html.

- To access the DB2 Version 9.5 documentation, follow this URL: http://www.ibm.com/support/knowledgecenter/SSEPGG_9.5.0/ com.ibm.db2.luw.kc.doc/welcome.html.

Terms and conditions

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

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

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

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

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

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the previous 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.

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

Appendix M. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

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

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, changes, or both in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to websites not owned by IBM are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you 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
   U59/3600
   3600 Steeles Avenue East
   Markham, Ontario L3R 9Z7
   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 contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

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

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 or registered trademarks of Oracle, its affiliates, 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, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- 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 389
ABSVAL scalar function 389
access plans
    description 54
ACOS scalar function
    details 390
ADD_MONTHS scalar function 391
ADVISE_INDEX table 1163
ADVISE_INSTANCE table 1167
ADVISE_MQT table 1168
ADVISE_PARTITION table 1170
ADVISE_TABLE table 1172
ADVISE_WORKLOAD table 1173
aggregate functions
    ARRAY_AGG 359
    COUNT 366
    details 357
    MIN 376
    TRIM_ARRAY 677
    UNNEST 764
aliases
    chaining process 14
    details 14, 60
    TABLE_NAME function 644
    TABLE_SCHEMA function 645
ALL clause
    quantified predicate 305
    SELECT statement 787
ambiguous reference errors 60
analytics
    in-database 792
anchored data types
    resolving anchor object 154
AND truth table 300
ANY clause 305
application design
    character conversions
        SQL statements 31
    code points for special characters 31
    double-byte character support (DBCS) 31
application processes
    connection states 37
    details 20
application requesters
    overview 32
application-directed distributed unit of work facility 36
arithmetic
    adding values 381
    AVG function 363
    CORRELATION function 365
    COVARIANCE function 369
    decimal values from numeric expressions 447
    finding maximum value 374
    floating-point values from numeric expressions 459, 583
    floating-point values from string expressions 583
    integer values
        returning from expressions 403, 502
    operators 239
    regression functions 377

arithmetic *(continued)*
    small integer values
        returning from expressions 623
    STDDEV function 380
    VARIANCE function 382
array constructors 269
ARRAY element
    specification 268
ARRAY_AGG function 359
ARRAY_DELETE scalar function 393
ARRAY_EXISTS predicate 308
ARRAY_FIRST scalar function 394
ARRAY_LAST scalar function 395
ARRAY_NEXT scalar function 396
ARRAY_PRIOR scalar function 397
arrays
    values 109
AS clause
    ORDER BY clause 832
    SELECT clause 787
ASCII scalar function
    details 398
ASIN scalar function
    details 399
assignments
    basic SQL operations 125
ATAN scalar function
    details 400
ATAN2 scalar function
    details 401
ATANH scalar function 402
attributes
    names 60
authorities
    overview 14
authorization IDs
    details 60
    global variables 209
authorization names
    details 60
    restrictions 60
AVG aggregate function 363

B

base tables
    comparison with other table types 6
BASE_TABLE function 762
basic predicate 303
BETWEEN predicate 309
BIGINT data type
    overview 90
    precision 90
    sign 90
BIGINT function 403
binary large objects (BLOBs)
    overview 100
    scalar function 407
binary string data types 100
binding
    function semantics 236

MOD scalar function
    details   544
MON_INTERVAL_ID global variable
    details   334
monitoring
    database events
        configuring event monitors   41
MONTH scalar function
    details   547
MONTHNAME scalar function
    details   548
months
    date arithmetic   391, 550
MONTHS_BETWEEN scalar function   550
MQTs
    overview   6
multibyte character support
    code points for special characters   31
multiple row VALUES clause
    result data type   144
MULTIPLY_ALT function   552

N

naming conventions
    identifiers   60
    qualified column rules   60
national character strings   99
NCHAR national character string   99
NCHAR scalar function   554
NCLOB national character string   99
NCLOB scalar function   556
nested table expressions
    subselect   787, 792, 817, 832
NEXT_DAY scalar function   559
nickname column options
    description   49
nicknames
    description
        data source objects   48
    FROM clause
        exposed names   60
        nonexposed names   60
        subselect   787
    qualifying column names   60
    SELECT clause   787
NLS_STRING_UNITS global variable   335
NODENUMBER function   441
non-exposed correlation name in FROM clause   60
non-repeatable reads
    isolation levels   22
nonrelational data sources
    specifying data type mappings   50
NORMALIZE_DECFLOAT scalar function   561
notices   1237
NULL
    SQL value
        assigning   125
        grouping-expressions   817
        indicator variables   60
        occurrences in duplicate rows   787
        overview   88
        result columns   787
NULL predicate   320
NULL-terminated character strings   93
NULLIF function   562

numbers
    precision   881
    scale   881
numeric assignments in SQL operations   125
numeric comparisons in SQL operations   125
NUMERIC data type
    precision   90
    sign   90
numeric data types
    summary   90
NVARCHAR national character string   99
NVARCHAR scalar function   557
NVL scalar function   563
NVL2 scalar function   564

O

OBJECT_METRICS table   1209
objects
    ownership   14
    tables   60
OCTET_LENGTH scalar function   565
ODBC
    overview   2
    supported versions   51
OLAP
    functions   275
    specification   275
OLE DB
    supported versions   51
online DB2 documentation
    IBM Knowledge Center   1234
operands
    decimal   239
    floating-point   239
    integer   239
    result data type   144
    strings   239
operations
    assignments   125
    comparisons   125
    datetime   252
    dereference   271
operators
    arithmetic   239
optimization
    description   54
optimize-for-clause   855
OR truth table   300
ORDER BY clause
    culturally correct collation   422
    SELECT statement   832
order of evaluation   239
ordinary tokens   59
ORG sample table   1129
outer joins
    joined tables   811
OVERLAY scalar function   566
overloaded functions
    multiple function instances   213
overloaded methods   228
ownership
    database objects   14

T

U

UCASE (locale sensitive) scalar function 687
UCASE scalar function
    details 686
UDFs
    details 213, 770
UDTs
    casting 117
    details 111
    distinct types
        details 111
    reference types 111
    structured types 111
unary operators
    minus sign 239
    plus sign 239
uncommitted read (UR) isolation level
    details 22
undefined reference errors 60
Unicode
    conventions xix
    conversion to uppercase 59
Unicode UCS-2 encoding
    functions 387
    pattern matching 152
    string comparisons 152
UNION operator
    role in comparison of fullselect 840
unique correlation names 60
units of work
    application-directed distributed 36
    overview 20
    semantics 40
UNNEST function 764
unqualified names 60
untyped expressions
    determining data types 291
update-clause 853
updates
    updatable special registers 165
UPPER (locale sensitive) scalar function 689
UPPER scalar function 688
user mappings
    description 47
    storing 47
USER special register 207
user-defined array types 109
user-defined functions
    See UDFs 770
user-defined global variables 208
user-defined methods
    details 228
user-defined types (UDTs)
    unsupported data types 50

V

VALIDATED predicate 323
VALUE function 691
values
    null 88
    overview 88
    sequence 286
VALUES clause
    fullselect 840

VARCHAR data type
    details 93
    DOUBLE_PRECISION or DOUBLE scalar function 459
    WEEK scalar function 718
    WEEK_ISO scalar function 719
VARCHAR function 692
VARCHAR_BIT_FORMAT function 698
VARCHAR_FORMAT function 700
VARCHAR_FORMAT_BIT function 708
VARGRAPHIC data type
    details 97
VARGRAPHIC function 709
variables
    global
        assigning values 211
        authorizations 209
        built-in 329
        overview 208
        resolving references to 209
        restrictions 211
        retrieving values 211
        types 208
    resolving global variable references 209
VARIANCE aggregate function 382
varying-length character string 93
varying-length graphic string 97
VERIFY_GROUP_FOR_USER scalar function 715
VERIFY_ROLE_FOR_USER scalar function 716
VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER scalar
  function 717
views
    exposed names in FROM clause 60
    FROM clause 60, 787
    names
        FROM clauses 791
        identifiers 60
        SELECT clauses 787
    non-exposed names in FROM clause 60
    overview 12
    qualifying column names 60

W

WEEK scalar function
    details 718
WEEK_ISO scalar function
    details 719
WHERE clause
    subselect component of fullselect 816
wild cards
    LIKE predicate 315
WITH common table expression
    select-statement 846
words
    SQL reserved 1155
wrappers
    description 46
    names 60

X

X/Open Company 2
X/Open SQL CLI 2
XML
    size limits 861
    supported versions 51

Y

IBM ®

Printed in USA

Spine information:

IBM DB2 10.5 for Linux, UNIX, and Windows

SQL Reference Volume 1

IBM