IBM® DB2 Universal Database™

SQL Reference Volume 1

Version 8.2

SC09-4844-01

IBM® DB2 Universal Database™

SQL Reference Volume 1

Version 8.2

SC09-4844-01

Before using this information and the product it supports, be sure to read the general information under *Notices*.

Contents

About this book

The SQL Reference in its two volumes defines the SQL language used by DB2 Universal Database Version 8, and 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

This book contains information about the following major topics:

- Chapter 1, “Concepts,” on page 1 discusses the basic concepts of relational databases and SQL.
- Chapter 2, “Language elements,” on page 55 describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- Chapter 3, “Functions,” on page 225 contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.
- Chapter 4, “Queries,” on page 449 describes the various forms of a query.
- Appendix A, “SQL limits,” on page 501 lists SQL limitations.
- Appendix B, “SQLCA (SQL communications area),” on page 507 describes the SQLCA structure.
- Appendix C, “SQLDA (SQL descriptor area),” on page 513 describes the SQLDA structure.
- Appendix D, “Catalog views,” on page 523 describes the database catalog views.
- Appendix E, “Federated systems,” on page 637 describes options and type mappings for Federated Systems.
- Appendix F, “The SAMPLE database,” on page 687 describes the sample tables used in examples.
- Appendix G, “Reserved schema names and reserved words,” on page 703 contains the reserved schema names and the reserved words for the IBM SQL and ISO/ANSI SQL99 standards.
- Appendix H, “Interaction of triggers and constraints,” on page 707 discusses the interaction of triggers and referential constraints.
- Appendix I, “Explain tables,” on page 709 describes the Explain tables.
- Appendix J, “Explain register values,” on page 735 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.

- Appendix K, “Exception tables,” on page 741 contains information about user-created tables that are used with the SET INTEGRITY statement.
- Appendix L, “SQL statements allowed in routines,” on page 745 lists the SQL statements that are allowed to execute in routines with different SQL data access contexts.
- Appendix M, “CALL invoked from a compiled statement,” on page 749 describes the CALL statement that can be invoked from a compiled statement.
- Appendix N, “Japanese and traditional-Chinese extended UNIX code (EUC) considerations,” on page 755 lists considerations when using extended UNIX code (EUC) character sets.
- Appendix O, “Backus-Naur form (BNF) specifications for DATALINKs,” on page 761 contains the Backus-Naur form (BNF) specifications for DATALINKs.

A brief overview of Volume 2

The second volume of the SQL Reference contains information about the syntax and semantics of SQL statements. The specific chapters in that volume are briefly described here:

- “SQL statements” contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- “SQL control statements” contains syntax diagrams, semantic descriptions, rules, and examples of SQL procedure statements.

How to read the syntax diagrams

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

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

The ►►─── symbol indicates the beginning of a syntax diagram.

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

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

The ──►◄ symbol indicates the end of a syntax diagram.

Syntax fragments start with the ├─── symbol and end with the ───┤ symbol.

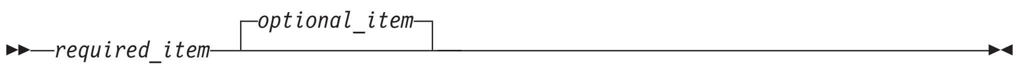
Required items appear on the horizontal line (the main path).

```
►►──required_item──────────────────────────────────────────────────────►◄
```

Optional items appear below the main path.

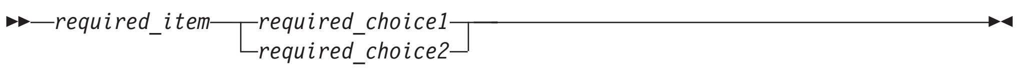
```
►►──required_item──┬───────────────┬───────────────────────────────────►◄
                   └─optional_item─┘
```

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

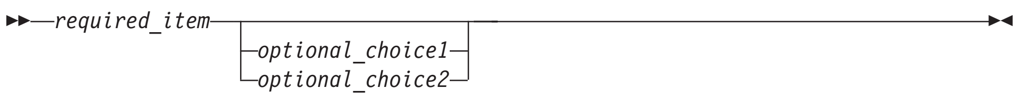


If you can choose from two or more items, they appear in a stack.

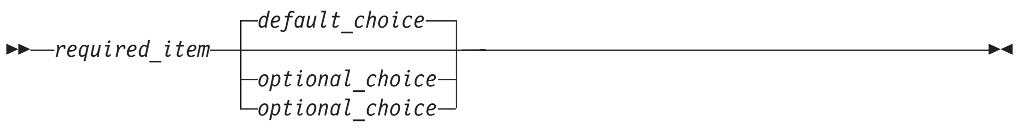
If you *must* choose one of the items, one item of the stack appears on the main path.



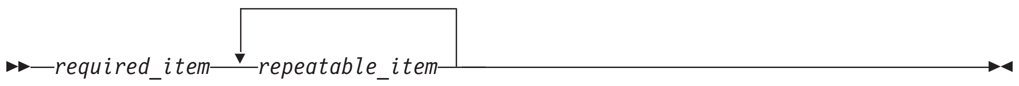
If choosing one of the items is optional, the entire stack appears below the main path.



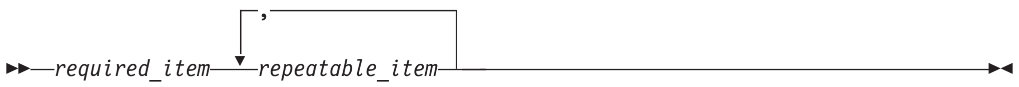
If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



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



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

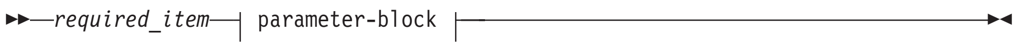


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items or repeat a single choice.

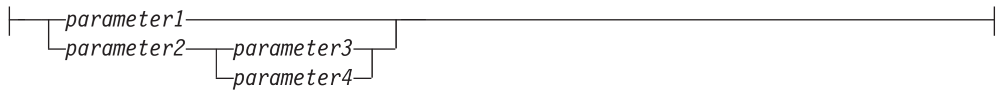
Keywords appear in uppercase (for example, `FROM`). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

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

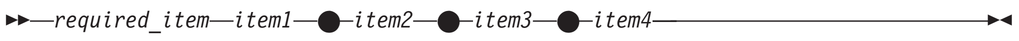
Sometimes a single variable represents a larger fragment of the syntax. For example, in the following diagram, the variable `parameter-block` represents the whole syntax fragment that is labeled **parameter-block**:



parameter-block:



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

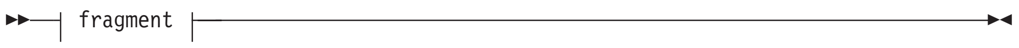


The above diagram shows that item2 and item3 may be specified in either order. Both of the following are valid:

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

Common syntax elements

The following sections describe a number of syntax fragments that are used in syntax diagrams. The fragments are referenced as follows:

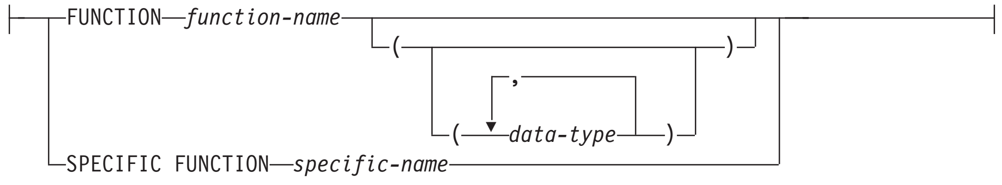


Function designator

A function designator uniquely identifies a single function. Function designators typically appear in DDL statements for functions (such as DROP or ALTER).

Syntax:

function-designator:



Description:

FUNCTION *function-name*
: Identifies a particular function, and is valid only if there is exactly one function instance with the name *function-name* in the schema. The identified function can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER

precompile/bind option implicitly specifies the qualifier for unqualified object names. If no function by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the function in the named or implied schema, an error (SQLSTATE 42725) is raised.

FUNCTION *function-name (data-type,...)*
Provides the function signature, which uniquely identifies the function. The function resolution algorithm is not used.

function-name
Specifies the name of the function. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE FUNCTION statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific function instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE FUNCTION statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no function with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC FUNCTION *specific-name*
Identifies a particular user-defined function, using the name that is specified or defaulted to at function creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific function instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Method designator

A method designator uniquely identifies a single method. Method designators typically appear in DDL statements for methods (such as DROP or ALTER).

Syntax:

Method designator

method-designator:

```
├──┬─METHOD──method-name──┬────────────────────────────┬──FOR──type-name─┬───────┤
   │                      └─(──┬──────────────────┬──)─┘                 │
   │                           │     ┌─,──────┐   │                      │
   │                           │     ▼        │   │                      │
   │                           └─(───data-type──)─┘                      │
   └─SPECIFIC METHOD──specific-name──────────────────────────────────────┘
```

Description:

METHOD *method-name*
: Identifies a particular method, and is valid only if there is exactly one method instance with the name *method-name* for the type *type-name*. The identified method can have any number of parameters defined for it. If no method by this name exists for the type, an error (SQLSTATE 42704) is raised. If there is more than one instance of the method for the type, an error (SQLSTATE 42725) is raised.

METHOD *method-name (data-type,...)*
: Provides the method signature, which uniquely identifies the method. The method resolution algorithm is not used.

  method-name
  : Specifies the name of the method for the type *type-name*.

  (data-type,...)
  : Values must match the data types that were specified (in the corresponding position) on the CREATE TYPE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific method instance.

    If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

    It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

    FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

    If length, precision, or scale is coded, the value must exactly match that specified in the CREATE TYPE statement.

    A type of FLOAT(*n*) does not need to match the defined value for *n*, because $0 < n < 25$ means REAL, and $24 < n < 54$ means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

    If no method with the specified signature exists for the type in the named or implied schema, an error (SQLSTATE 42883) is raised.

  FOR *type-name*
  : Names the type with which the specified method is to be associated. The name must identify a type already described in the catalog (SQLSTATE 42704). In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

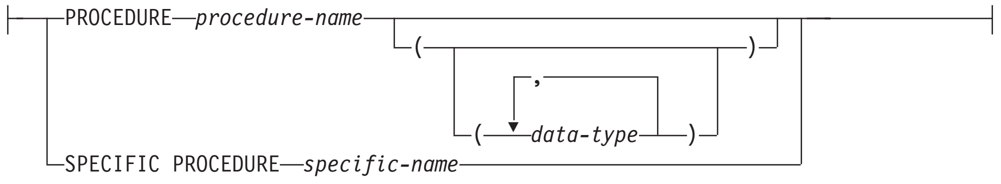
SPECIFIC METHOD *specific-name*
Identifies a particular method, using the name that is specified or defaulted to at method creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific method instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Procedure designator

A procedure designator uniquely identifies a single stored procedure. Procedure designators typically appear in DDL statements for procedures (such as DROP or ALTER).

Syntax:

procedure-designator:



Description:

PROCEDURE *procedure-name*
Identifies a particular procedure, and is valid only if there is exactly one procedure instance with the name *procedure-name* in the schema. The identified procedure can have any number of parameters defined for it. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. If no procedure by this name exists in the named or implied schema, an error (SQLSTATE 42704) is raised. If there is more than one instance of the procedure in the named or implied schema, an error (SQLSTATE 42725) is raised.

PROCEDURE *procedure-name (data-type,...)*
Provides the procedure signature, which uniquely identifies the procedure. The procedure resolution algorithm is not used.

procedure-name
Specifies the name of the procedure. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names.

(data-type,...)
Values must match the data types that were specified (in the corresponding position) on the CREATE PROCEDURE statement. The number of data types, and the logical concatenation of the data types, is used to identify the specific procedure instance.

If a data type is unqualified, the type name is resolved by searching the schemas on the SQL path. This also applies to data type names specified for a REFERENCE type.

It is not necessary to specify the length, precision, or scale for the parameterized data types. Instead, an empty set of parentheses can be coded to indicate that these attributes are to be ignored when looking for a data type match.

FLOAT() cannot be used (SQLSTATE 42601), because the parameter value indicates different data types (REAL or DOUBLE).

If length, precision, or scale is coded, the value must exactly match that specified in the CREATE PROCEDURE statement.

A type of FLOAT(*n*) does not need to match the defined value for *n*, because 0 < *n* < 25 means REAL, and 24 < *n* < 54 means DOUBLE. Matching occurs on the basis of whether the type is REAL or DOUBLE.

If no procedure with the specified signature exists in the named or implied schema, an error (SQLSTATE 42883) is raised.

SPECIFIC PROCEDURE *specific-name*
Identifies a particular procedure, using the name that is specified or defaulted to at procedure creation time. In dynamic SQL statements, the CURRENT SCHEMA special register is used as a qualifier for an unqualified object name. In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified object names. The *specific-name* must identify a specific procedure instance in the named or implied schema; otherwise, an error (SQLSTATE 42704) is raised.

Conventions used in this manual

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

Error conditions

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

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

Highlighting conventions

The following conventions are used in this book.

Bold	Indicates commands, keywords, and other items whose names are predefined by the system.
Italics	Indicates one of the following: • 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.
`Monospace`	Indicates one of the following: • Files and directories. • Information that you are instructed to type at a command prompt or in a window. • Examples of specific data values. • Examples of text similar to what may be displayed by the system. • Examples of system messages.

Related documentation

The following publications may prove useful in preparing applications:

- *Administration Guide*
  - Contains information required to design, implement, and maintain a database to be accessed either locally or in a client/server environment.
- *Application Development Guide*
  - Discusses the application development process and how to code, compile, and execute application programs that use embedded SQL and APIs to access the database.
- *DB2 Universal Database for iSeries SQL Reference*
  - This book defines Structured Query Language (SQL) as supported by DB2 Query Manager and SQL Development Kit on iSeries (AS/400). 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 iSeries (AS/400) systems running DB2.
- *DB2 Universal Database for z/OS and OS/390 SQL Reference*
  - This book defines Structured Query Language (SQL) used in DB2 for z/OS (OS/390). It provides query forms, SQL statements, SQL procedure statements, DB2 limits, SQLCA, SQLDA, catalog tables, and SQL reserved words for z/OS (OS/390) systems running DB2.
- *DB2 Spatial Extender User's Guide and Reference*
  - This book discusses how to write applications to create and use a geographic information system (GIS). Creating and using a GIS involves supplying a database with resources and then querying the data to obtain information such as locations, distances, and distributions within areas.
- *IBM SQL Reference*
  - This book contains all the common elements of SQL that span IBM's database products. It provides limits and rules that assist in preparing portable programs using IBM databases. This manual provides a list of SQL extensions and incompatibilities among the following standards and products: SQL92E, XPG4-SQL, IBM-SQL and the IBM relational database products.
- *American National Standard X3.135-1992, Database Language SQL*
  - Contains the ANSI standard definition of SQL.
- *ISO/IEC 9075:1992, Database Language SQL*
  - Contains the 1992 ISO standard definition of SQL.
- *ISO/IEC 9075-2:1999, Database Language SQL -- Part 2: Foundation (SQL/Foundation)*
  - Contains a large portion of the 1999 ISO standard definition of SQL.
- *ISO/IEC 9075-4:1999, Database Language SQL -- Part 4: Persistent Stored Modules (SQL/PSM)*
  - Contains the 1999 ISO standard definition for SQL procedure control statements.
- *ISO/IEC 9075-5:1999, Database Language SQL -- Part 4: Host Language Bindings (SQL/Bindings)*
  - Contains the 1999 ISO standard definition for host language bindings and dynamic SQL.

Related documentation

Chapter 1. Concepts

This chapter provides a high-level view of concepts that are important to understand when using Structured Query Language (SQL). The reference material contained in the rest of this manual provides a more detailed view.

Relational databases

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.

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

Privileges, authority levels, and database authorities

Privileges enable users to create or access database resources. *Authority levels* provide a method of grouping privileges and higher-level database manager maintenance and utility operations. *Database authorities* enable users to perform activities at the database level. Privileges, authority levels, and database authorities can be used together to control access to the database manager and its database objects. Users can access only those objects for which they have the required privilege, authority level, or database authority, which DB2® Universal Database (DB2 UDB) determines when it performs an authorization check for an authenticated user.

The database manager requires that each user be specifically authorized, either implicitly or explicitly, to use each database function needed to perform a specific task. *Explicit* authorities or privileges are granted to the user (GRANTEETYPE of U in the database catalogs). *Implicit* authorities or privileges are granted to a group to which the user belongs (GRANTEETYPE of G in the database catalogs). Thus, to create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table, and so on.

Figure 1 illustrates the relationship between authorities and their span of control (database, database manager).

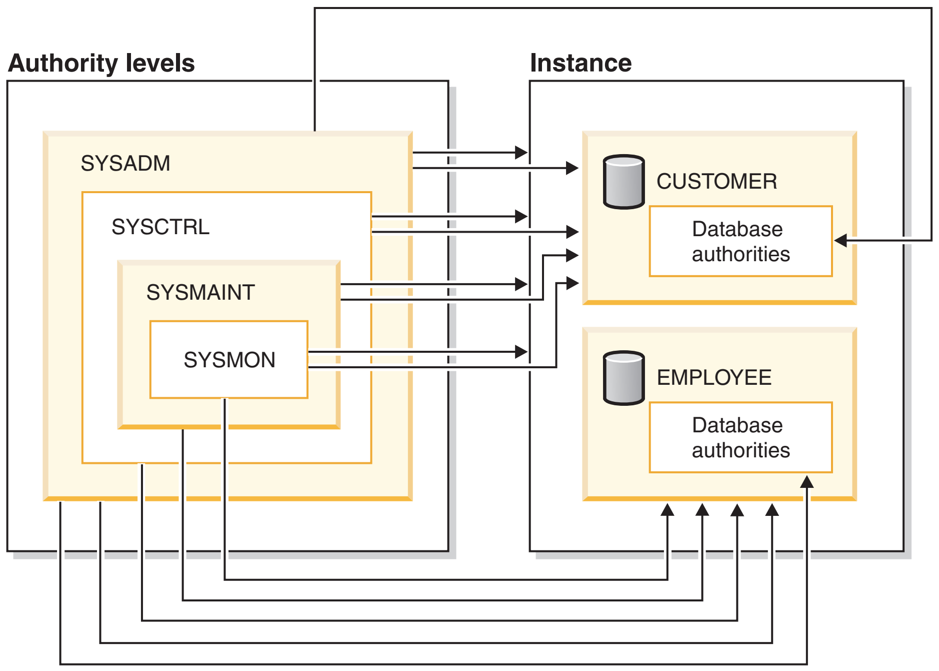


Figure 1. Hierarchy of Authorities

A user or group can have one or more of the following authorities or privileges:

- Administrative authority:
  - SYSADM (system administrator)

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

    The user who possesses SYSADM authority is responsible both for controlling the database manager, and for ensuring the safety and integrity of the data. SYSADM authority provides implicit privileges on all objects in the database, control over which users can access the database manager, and the extent of this access. For more information about SYSADM authority, see ″System administration authority (SYSADM)″.
  - DBADM (database administrator)

The DBADM database authority provides administrative authority over a single database. This database administrator possesses the privileges required to create objects, issue database commands, and access table data. The database administrator can also grant and revoke CONTROL and individual privileges. For more information about DBADM authority, see ″Database administration authority (DBADM)″.

- System control authority:
  - SYSCTRL (system control)

    The SYSCTRL authority level provides control over operations that affect system resources. For example, a user with SYSCTRL authority can create, update, stop, or drop a database. This user can also stop an instance, but cannot access table data. Users with SYSCTRL authority also have SYSMON authority. For more information about SYSCTRL authority, see ″System control authority (SYSCTRL)″.
  - SYSMAINT (system maintenance)

    The SYSMAINT authority level 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. For more information about SYSMAINT authority, see ″System maintenance authority (SYSMAINT)″.
- SYSMON (system monitor authority)

  The SYSMON authority level provides the authority required to use the database system monitor. For more information about SYSMON authority, see ″System monitor authority (SYSMON)″
- Database authorities

  To perform activities such as creating a table or a routine, or for loading data into a table, specific database authorities are required. For more information, see ″Database authorities″.
- Privileges:

  Privileges are required to perform activities on database objects (for example, to create and drop an index). Privileges strictly define the tasks that a user can perform. For example, a user may have the privilege to create an index on a table, but not a trigger on the same table.
  - 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 SYSADM or DBADM authority must grant the CONTROL privilege to that object. The CONTROL privilege cannot be revoked from the object owner.

    In some situations, the creator of an object automatically obtains the CONTROL privilege on that object. For more information, see ″Object creation, ownership, and privileges″.
  - Individual privileges can be granted to allow a user to carry out specific tasks on specific objects.

Users with administrative authority (SYSADM or DBADM) or the CONTROL privilege can grant and revoke privileges to and from users.

Individual privileges and database authorities allow a specific function, but do not include the right to grant the same privileges or authorities to other users. The right to grant table, view, schema, package, routine, and sequence privileges to others can be extended to other users through the WITH GRANT OPTION on the GRANT statement. However, the WITH GRANT OPTION does not allow the person granting the privilege to revoke the privilege once granted. You must have SYSADM authority, DBADM authority, or the CONTROL privilege to revoke the privilege.

Privileges can also be granted to PUBLIC. PUBLIC privileges apply to all users (authorization names), including any future users, regardless of whether any individual users have previously been granted the privilege.

- Implicit privileges may be granted to a user who has the privilege to execute a package. While users can run the application, they do not necessarily require explicit privileges on the data objects used within the package.

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 is given authorities and privileges and there is no user created with that authorization name. At some later time, a user can be created with that authorization 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. In DB2 UDB, 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.

Related concepts:

- “System administration authority (SYSADM)” in the *Administration Guide: Implementation*
- “System control authority (SYSCTRL)” in the *Administration Guide: Implementation*
- “System maintenance authority (SYSMAINT)” in the *Administration Guide: Implementation*
- “Database administration authority (DBADM)” in the *Administration Guide: Implementation*
- “LOAD authority” in the *Administration Guide: Implementation*
- “Database authorities” in the *Administration Guide: Implementation*
- “Schema privileges” in the *Administration Guide: Implementation*
- “Table space privileges” in the *Administration Guide: Implementation*
- “Table and view privileges” in the *Administration Guide: Implementation*
- “Package privileges” in the *Administration Guide: Implementation*

- "Index privileges" in the *Administration Guide: Implementation*
- "Sequence privileges" in the *Administration Guide: Implementation*
- "Controlling access to database objects" in the *Administration Guide: Implementation*
- "Indirect privileges through a package" in the *Administration Guide: Implementation*
- "Routine privileges" in the *Administration Guide: Implementation*
- "Object creation, ownership, and privileges" in the *Administration Guide: Implementation*
- "System monitor authority (SYSMON)" in the *Administration Guide: Implementation*

Schemas

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

A schema is also an object in the database. 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, provided that the user has IMPLICIT_SCHEMA database 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.

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, all users have IMPLICIT_SCHEMA authority. This allows any user to create objects in any schema not already in existence. An implicitly created schema allows any user to create other objects in this schema.The ability to create 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.

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

Related concepts:

- "Schema privileges" in the *Administration Guide: Implementation*

Tables

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

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

A *summary table* is a table defined by a query that is also used to determine the data in the table. Summary tables can be used to improve the performance of queries. If the database manager determines that a portion of a query 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 table can define the data type of each column separately, or base the types on the attributes of a user-defined structured type. This is called a *typed table*. A user-defined structured type may be part of a type hierarchy. A *subtype* inherits attributes from its *supertype*. Similarly, a typed table can be part of a table hierarchy. A *subtable* inherits columns from its *supertable*. Note that the term *subtype* applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy. Similarly, the term *subtable* applies to a typed table and all typed tables that are below it in the table hierarchy. A *proper subtable* of a table T is a table below T in the table hierarchy.

A *declared temporary table* is created with a DECLARE GLOBAL TEMPORARY TABLE statement and is used to hold temporary data on behalf of a single application. This table is dropped implicitly when the application disconnects from the database.

Views

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 base table. All views can be used just like base tables for data retrieval. Whether a view can be used in an insert, update, or delete operation depends on its definition.

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 may 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 base 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 base table column. For example, if a view includes a foreign key of its base table, insert and update operations using that view are subject to the same referential constraints as is the base table. Also, if the base table of a view is a parent table, delete and update operations using that view are subject to the same rules as are delete and update operations on the base 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 base table is dropped); if this occurs, the view is no longer available for SQL operations.

Aliases

An *alias* is an alternative name for a table or a view. It can be used to reference a table or a view if an existing table or view *can* be referenced. An alias cannot be used in all contexts; for example, it cannot be used in the check condition of a check constraint. An alias cannot reference a declared temporary table.

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

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

Indexes

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

Indexes are used by the database manager to:

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

Keys

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

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

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

A *primary key* is a special case of a unique key. A table cannot have more than one primary key.

A *foreign key* is a key that is specified in the definition of a referential constraint.

A *partitioning key* is a key that is part of the definition of a table in a partitioned database. The partitioning key is used to determine the partition on which the row of data is stored. If a partitioning key is defined, unique keys and primary keys must include the same columns as the partitioning key, but can have additional columns. A table cannot have more than one partitioning key.

Constraints

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

There are four types of constraints:

- A *unique constraint* is a rule that forbids duplicate values in one or more columns within a table. Unique and primary keys are the supported unique constraints. For example, a unique constraint can be defined on the supplier identifier in the supplier table to ensure that the same supplier identifier is not given to two suppliers.
- A *referential constraint* is a logical rule about values in one or more columns in one or more tables. For example, a set of tables shares information about a corporation's suppliers. Occasionally, a supplier's name changes. You can define a referential constraint stating that the ID of the supplier in a table must match a supplier ID in the supplier information. This constraint prevents insert, update, or delete operations that would otherwise result in missing supplier information.
- A *table check constraint* sets restrictions on data added to a specific table. For example, a table check constraint can ensure that the salary level for an employee is at least $20,000 whenever salary data is added or updated in a table containing personnel information.
- An *informational constraint* is a rule that can be used by the SQL compiler, but that is not enforced by the database manager.

Referential and table check constraints can be turned on or off. It is generally a good idea, for example, to turn off the enforcement of a constraint when large amounts of data are loaded into a database.

Unique constraints

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

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

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

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

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

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

Referential constraints

Referential integrity is the state of a database in which all values of all foreign keys are valid. A *foreign key*is a column or a set of columns in a table whose values are required to match at least one primary key or unique key value of a row in its parent table. A *referential constraint* is the rule that the values of the foreign key are valid only if one of the following conditions is true:

- They appear as values of a parent key.
- Some component of the foreign key is null.

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

Referential constraints are optional and can be defined in the CREATE TABLE statement or the ALTER TABLE statement. Referential constraints are enforced by the database manager during the execution of INSERT, UPDATE, DELETE, ALTER TABLE, ADD CONSTRAINT, and SET INTEGRITY statements.

Referential constraints with a delete or an update rule of RESTRICT are enforced before all other referential constraints. Referential constraints with a delete or an update rule of NO ACTION behave like RESTRICT in most cases.

Note that referential constraints, check constraints, and triggers can be combined.

Referential integrity rules involve the following concepts and terminology:

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

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

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

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

Descendent table
A table is a descendent of table T if it is a dependent of T or a descendent of a dependent of T.

Dependent row
A row that has at least one parent row.

Descendent row
A row is a descendent of row r if it is a dependent of r or a descendent of a dependent of r.

Referential cycle
A set of referential constraints such that each table in the set is a descendent of itself.

Self-referencing table
A table that is a parent and a dependent in the same referential constraint. The constraint is called a *self-referencing constraint*.

Self-referencing row
A row that is a parent of itself.

Insert rule

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

Update rule

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

In the case of a parent row, when a value in a column of the parent key is updated, the following rules apply:

- If any row in the dependent table matches the original value of the key, the update is rejected when the update rule is RESTRICT.
- If any row in the dependent table does not have a corresponding parent key when the update statement is completed (excluding AFTER triggers), the update is rejected when the update rule is NO ACTION.

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

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

Delete rule

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

The delete rule of a referential constraint applies when a row of the parent table is deleted. More precisely, the rule applies when a row of the parent table is the object of a delete or propagated delete operation (defined below), and that row has dependents in the dependent table of the referential constraint. Consider an example where P is the parent table, D is the dependent table, and p is a parent row that is the object of a delete or propagated delete operation. The delete rule works as follows:

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

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

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

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

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

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

The following restrictions apply to delete-connected relationships:

- When a table is delete-connected to itself in a referential cycle of more than one table, the cycle must not contain a delete rule of either RESTRICT or SET NULL.
- A table must not both be a dependent table in a CASCADE relationship (self-referencing or referencing another table) and have a self-referencing relationship with a delete rule of either RESTRICT or SET NULL.
- When a table is delete-connected to another table through multiple relationships where such relationships have overlapping foreign keys, these relationships must have the same delete rule and none of these can be SET NULL.
- When a table is delete-connected to another table through multiple relationships where one of the relationships is specified with delete rule SET NULL, the foreign key definition of this relationship must not contain any partitioning key or MDC key column.
- When two tables are delete-connected to the same table through CASCADE relationships, the two tables must not be delete-connected to each other where the delete connected paths end with delete rule RESTRICT or SET NULL.

Table check constraints

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

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

When one or more table check constraints is defined in the ALTER TABLE statement for a table with existing data, the existing data is checked against the new condition before the ALTER TABLE statement completes. The SET INTEGRITY statement can be used to put the table in *check pending* state, which allows the ALTER TABLE statement to proceed without checking the data.

Informational constraints

An *informational constraint* is a rule that can be used by the SQL compiler to improve the access path to data. Informational constraints are not enforced by the database manager, and are not used for additional verification of data; rather, they are used to improve query performance.

Use the CREATE TABLE or ALTER TABLE statement to define a referential or table check constraint, specifying constraint attributes that determine whether or not the database manager is to enforce the constraint and whether or not the constraint is to be used for query optimization.

Related reference:
- "SET INTEGRITY statement" in the *SQL Reference, Volume 2*
- Appendix H, "Interaction of triggers and constraints," on page 707

Isolation levels

The *isolation level* associated with an application process defines the degree of isolation of that application process from other concurrently executing application processes. The isolation level of an application process therefore specifies:
- The degree to which the rows read and updated by the application are available to other concurrently executing application processes.
- The degree to which the update activity of other concurrently executing application processes can affect the application.

The isolation level for static SQL statements is specified as an attribute of a package and applies to the application processes that use the package. The isolation level is specified in the program preparation process. For dynamic SQL statements, the default isolation level is the isolation level specified for the package preparing the statement. Use of the SET CURRENT ISOLATION statement allows for alternate isolation levels to be specified for dynamic SQL issued within a session. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. (Declared temporary tables and their rows cannot be locked because they are only accessible to the application that declared them.)

The database manager supports three general categories of locks:

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

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

Exclusive
Prevents concurrent application processes from accessing the data in any way. Does not apply to application processes with an isolation level of *uncommitted read*, which can read but not modify the data.

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

Isolation levels

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

- Repeatable Read (RR)

  This level ensures that:

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

  – Any row changed by another application process cannot be read until it is committed by that application process.

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

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

- Read Stability (RS)

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

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

  – Any row changed by another application process cannot be read until it is committed by that application process.

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

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

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

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

- Cursor Stability (CS)

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

Unlike Repeatable Read, Cursor Stability only ensures that the current row of every updatable cursor is not changed by other application processes. Thus, the rows that were read during a unit of work can be changed by other application processes.

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

- Uncommitted Read (UR)

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

  – Any row read during a unit of work to be changed by other application processes.

  – Any row changed by another application process to be read, even if the change has not been committed by that application process.

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

Comparison of isolation levels

The following table summarizes information about isolation levels.

	UR	**CS**	**RS**	**RR**
Can the application see uncommitted changes made by other application processes?	Yes	No	No	No
Can the application update uncommitted changes made by other application processes?	No	No	No	No
Can the re-execution of a statement be affected by other application processes? *See phenomenon P3 (phantom) below.*	Yes	Yes	Yes	No
Can “updated” rows be updated by other application processes? See Note 1 below.	No	No	No	No
Can “updated” rows be read by other application processes that are running at an isolation level other than UR?	No	No	No	No
Can “updated” rows be read by other application processes that are running at the UR isolation level?	Yes	Yes	Yes	Yes
Can “accessed” rows be updated by other application processes? *See phenomenon P2 (nonrepeatable read) below.*	Yes	Yes	No	No
Can “accessed” rows be read by other application processes?	Yes	Yes	Yes	Yes
Can “current” row be updated or deleted by other application processes? *See phenomenon P1 (dirty-read) below.*	See Note 2 below.	See Note 2 below.	No	No

Notes:

1. The isolation level offers no protection to the application if the application is both reading and writing a table. For example, an application opens a cursor on a table and then performs an insert, update, or delete operation on the same table. The application may see inconsistent data when more rows are fetched from the open cursor.
2. If the cursor is not updatable, with CS the current row may be updated or deleted by other application processes in some cases. For example, buffering may cause the current row at the client to be different than what the current row actually is at the server.

	UR	CS	RS	RR

Examples of Phenomena:

P1 *Dirty Read.* Unit of work UW1 modifies a row. Unit of work UW2 reads that row before UW1 performs a COMMIT. If UW1 then performs a ROLLBACK, UW2 has read a nonexistent row.

P2 *Nonrepeatable Read.* Unit of work UW1 reads a row. Unit of work UW2 modifies that row and performs a COMMIT. If UW1 then re-reads the row, it might receive a modified value.

P3 *Phantom.* Unit of work UW1 reads the set of *n* rows that satisfies some search condition. Unit of work UW2 then INSERTs one or more rows that satisfies the search condition and performs a COMMIT. If UW1 then repeats the initial read with the same search condition, it obtains the original rows plus the inserted rows.

Related reference:

- "DECLARE CURSOR statement" in the *SQL Reference, Volume 2*

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.

Related reference:

- "Select-statement" on page 490
- "SQL queries" on page 449

Application processes, concurrency, and recovery

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

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

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

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

A *unit of work* is a recoverable sequence of operations within an application process. A unit of work is initiated when an application process is started, or when the previous unit of work is ended by something other than the termination of the application process. A unit of work is ended by a commit operation, a rollback operation, or the end of an application process. A commit or rollback operation affects only the database changes made within the unit of work it is ending.

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

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

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

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

The initiation and the termination of a unit of work define points of consistency within an application process. For example, a banking transaction may involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and then added to the second account. Following the subtraction step, the data is inconsistent. Only after the funds have been added to the second account is consistency reestablished. When both steps are complete, the commit operation can be used to end the unit of work, thereby making the changes available to other application processes. If a failure occurs before the unit of work ends, the database manager will roll back uncommitted changes to restore the data consistency that it assumes existed when the unit of work was initiated.

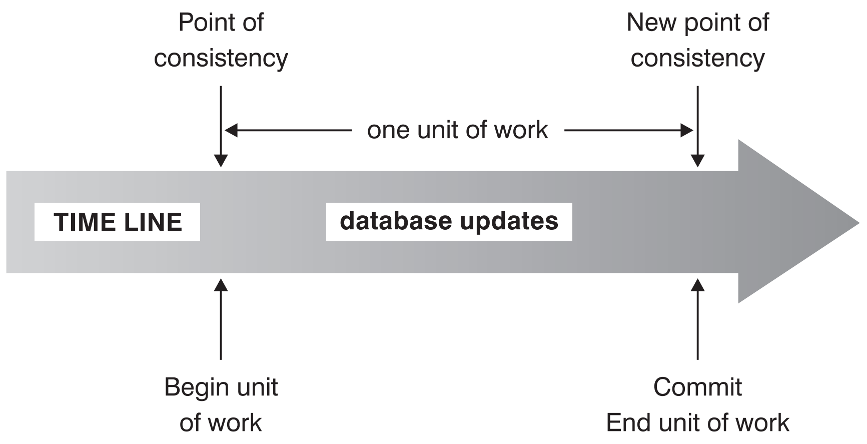


Figure 2. Unit of Work with a COMMIT Statement

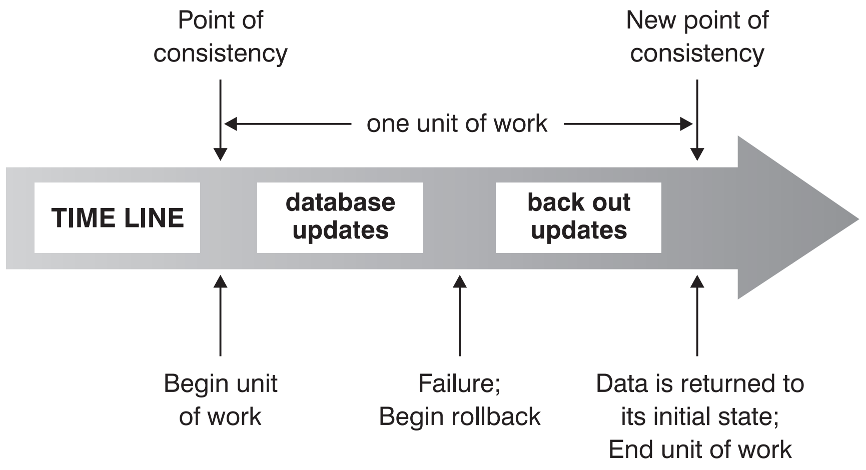


Figure 3. Unit of Work with a ROLLBACK Statement

Related concepts:

- "Isolation levels" on page 13

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

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

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

- CLI provides function calls that support a way of querying database catalogs that is consistent across the DB2 family. This reduces the need to write catalog queries that must be tailored to specific database servers.

- CLI provides the ability to scroll through a cursor:
  - Forward by one or more rows
  - Backward by one or more rows
  - Forward from the first row by one or more rows
  - Backward from the last row by one or more rows
  - From a previously stored location in the cursor.
- Stored procedures called from application programs that were written using CLI can return result sets to those programs.

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

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

- JDBC calls are translated into DB2 CLI calls through Java native methods. JDBC requests flow from the DB2 client through DB2 CLI to the DB2 server. JDBC cannot use static SQL.
- SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. An SQLJ source file must be translated by the SQLJ translator before the resulting Java source code can be compiled.

Packages

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

Catalog views

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

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

Related reference:

- "System catalog views" on page 523

Character conversion

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

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

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

Following is a list of terms used when discussing character conversion:

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

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

code point
: A unique bit pattern that represents a character.

encoding scheme
: A set of rules used to represent character data, for example:
  - Single-Byte ASCII
  - Single-Byte EBCDIC
  - Double-Byte ASCII
  - Mixed single- and double-byte ASCII

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

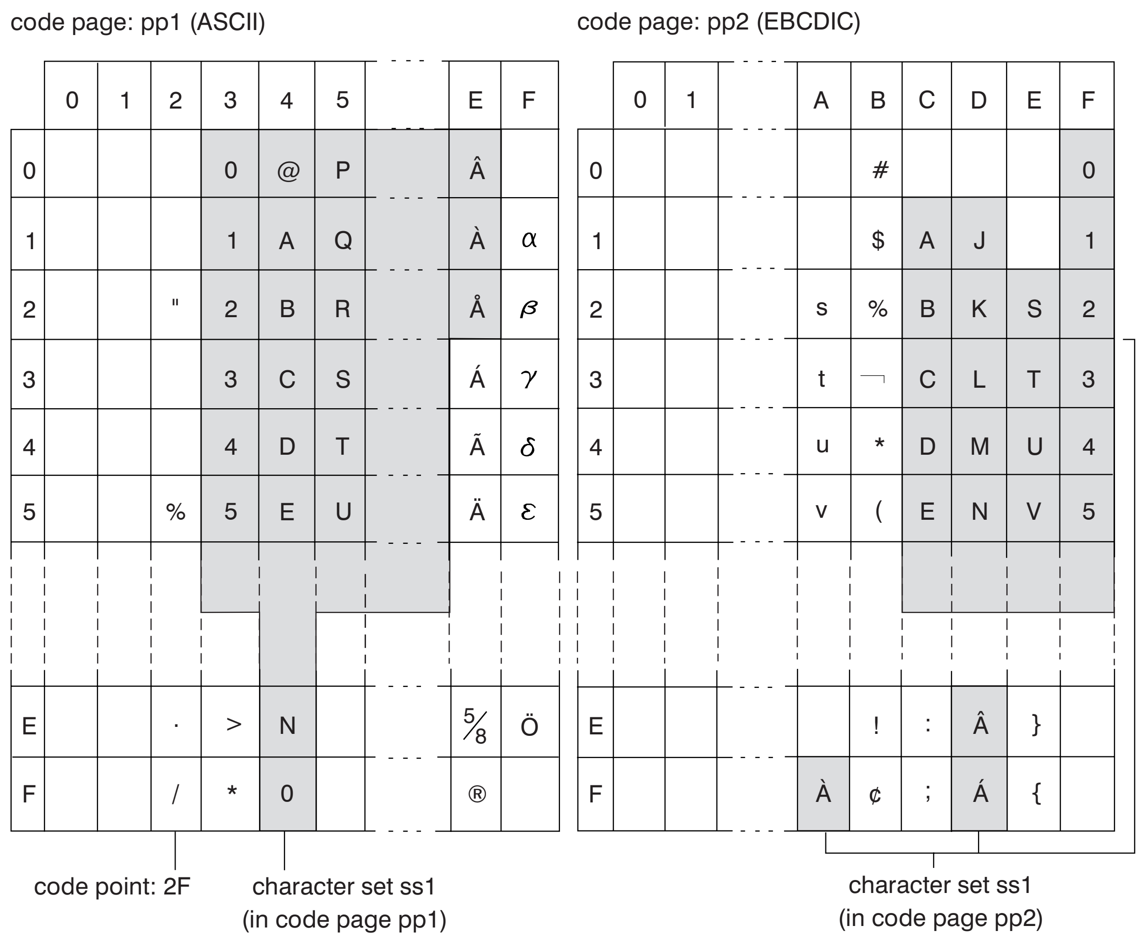

code page: pp1 (ASCII)
code page: pp2 (EBCDIC)
code point: 2F
character set ss1 (in code page pp1)
character set ss1 (in code page pp2)

Figure 4. 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 is not necessarily the same code page under which the application was bound.

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

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

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

Character string code pages have the following attributes:

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

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

Related reference:
- "Assignments and comparisons" on page 99
- "Rules for string conversions" on page 117

Event monitors

Event monitors are used to collect information about the database and any connected applications when specified events occur. Events represent transitions in database activity: for instance, connections, deadlocks, statements, and 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.

Note: By default, all databases have an event monitor named DB2DETAILDEADLOCK defined, which keeps track of DEADLOCKS WITH DETAILS. The DB2DETAILDEADLOCK event monitor starts automatically when the database starts.

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

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

When the CREATE EVENT MONITOR SQL statement is executed, the definition of the event monitor it creates is stored in the following database system catalog tables:
- SYSCAT.EVENTMONITORS: event monitors defined for the database.
- SYSCAT.EVENTS: events monitored for the database.
- SYSCAT.EVENTTABLES: target tables for table event monitors.

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

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

Related concepts:
- "Database system monitor" in the *System Monitor Guide and Reference*

Related tasks:
- "Collecting information about database system events" in the *System Monitor Guide and Reference*
- "Creating an event monitor" in the *System Monitor Guide and Reference*

Related reference:
- "Event monitor sample output" in the *System Monitor Guide and Reference*
- "Event types" in the *System Monitor Guide and Reference*

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 SQL statements that are executed whenever the trigger is activated. The SQL 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 SQLstates. If the trigger activation time is after the trigger event, triggered actions can include statements that select, insert, update, delete, or signal SQLstates.

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, and separate transition tables can be defined for old and new values.

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

The activation of a trigger may cause *trigger cascading*, which is the result of the activation of one trigger that executes SQL statements that cause the activation of other triggers or even the same trigger again. The triggered actions may also cause updates 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.

Related reference:

- Appendix H, “Interaction of triggers and constraints,” on page 707

Table spaces and other storage structures

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

Database managed space (DMS)
A table space that is managed by the database manager.

System managed space (SMS)
A table space that is managed by the operating system.

All table spaces consist of containers. A *container* describes where objects are stored. A subdirectory in a file system is an example of a container.

When data is read from table space containers, it is placed in an area of memory called a buffer pool. A *buffer pool* is associated with a specific table space, thereby allowing control over which data will share the same memory areas for data buffering.

In a partitioned database, data is spread across different database partitions. Exactly which partitions are included is determined by the database partition group that is assigned to the table space. A *database partition group* is a group of one or more partitions that are defined as part of the database. A table space includes one or more containers for each partition in the database partition group. A *partitioning map*, associated with each database partition group, is used by the database manager to determine on which partition a given row of data is to be stored. The partitioning map is an array of 4096 partition numbers. The partitioning map index produced by the partitioning function for each row in a table is used as an index into the partitioning map to determine the partition on which a row is to be stored. As an example, the following figure shows how a row with partitioning key value (c1, c2, c3) is mapped to partitioning map index 2 which, in turn, references partition p5.

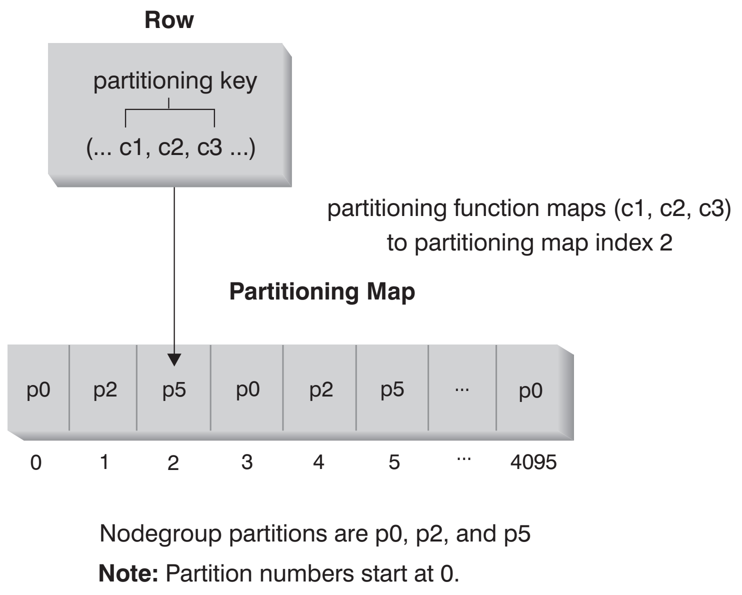

Row
partitioning key
(... c1, c2, c3 ...)
partitioning function maps (c1, c2, c3)
to partitioning map index 2
Partitioning Map
p0
p2
p5
p0
p2
p5
...
p0
0
1
2
3
4
5
...
4095
Nodegroup partitions are p0, p2, and p5
Note: Partition numbers start at 0.

Figure 5. Data Distribution

The partitioning map can be changed, allowing the data distribution to be changed without modifying the partitioning key or the actual data. The new partitioning map is specified as part of the REDISTRIBUTE DATABASE PARTITION GROUP command or the sqludrdt application programming interface (API), which use it to redistribute the tables in the database partition group.

The DB2® Data Links Manager product provides functionality that supports additional storage capabilities. A normal user table can include columns (defined with the DATALINK data type) that register links to data stored in external files. DATALINK values point to data files that are stored on an external file server.

Related concepts:

- "Data partitioning across multiple partitions" on page 26

Related reference:

- "CREATE BUFFERPOOL statement" in the *SQL Reference, Volume 2*
- "CREATE DATABASE PARTITION GROUP statement" in the *SQL Reference, Volume 2*
- "CREATE TABLESPACE statement" in the *SQL Reference, Volume 2*

Data partitioning across multiple partitions

DB2® allows great flexibility in spreading data across multiple partitions (nodes) of a partitioned database. Users can choose how to partition their data by declaring partitioning keys, and can determine which and how many partitions their table data can be spread across by selecting the database partition group and table space in which the data should be stored. In addition, a partitioning map (which is updatable) specifies the mapping of partitioning key values to partitions. This makes it possible for flexible workload parallelization across a partitioned database for large tables, while allowing smaller tables to be stored on one or a small number of partitions if the application designer so chooses. Each local partition may have local indexes on the data it stores to provide high performance local data access.

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

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

The partitioning 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 partitioning key for a table is created by default from the first column of the primary key. If no primary key is defined, the default partitioning key is the first column defined in that table that has a data type other than a long or a LOB data type. Partitioned tables must have at least one column that is neither a long nor a LOB data type. A table in a table space that is in a single partition database partition group will have a partitioning key only if it is explicitly specified.

Hash partitioning is used to place a row in a partition as follows:

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

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

DB2 has the capability of recognizing when data being accessed for a join or a subquery is located at the same partition in the same database partition group. This is known as *table collocation*. Rows in collocated tables with the same partitioning key values are located on the same partition. DB2 can choose to perform join or subquery processing at the partition in which the data is stored. This can have significant performance advantages.

Collocated tables must:

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

Related reference:

- "Partition-compatible data types" on page 119

Distributed relational databases

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

Distributed relational databases are built on formal requester-server protocols and functions. An *application requester* supports the application end of a connection. It transforms a database request from the application into communication protocols suitable for use in the distributed database network. These requests are received and processed by 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

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.

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 may 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.

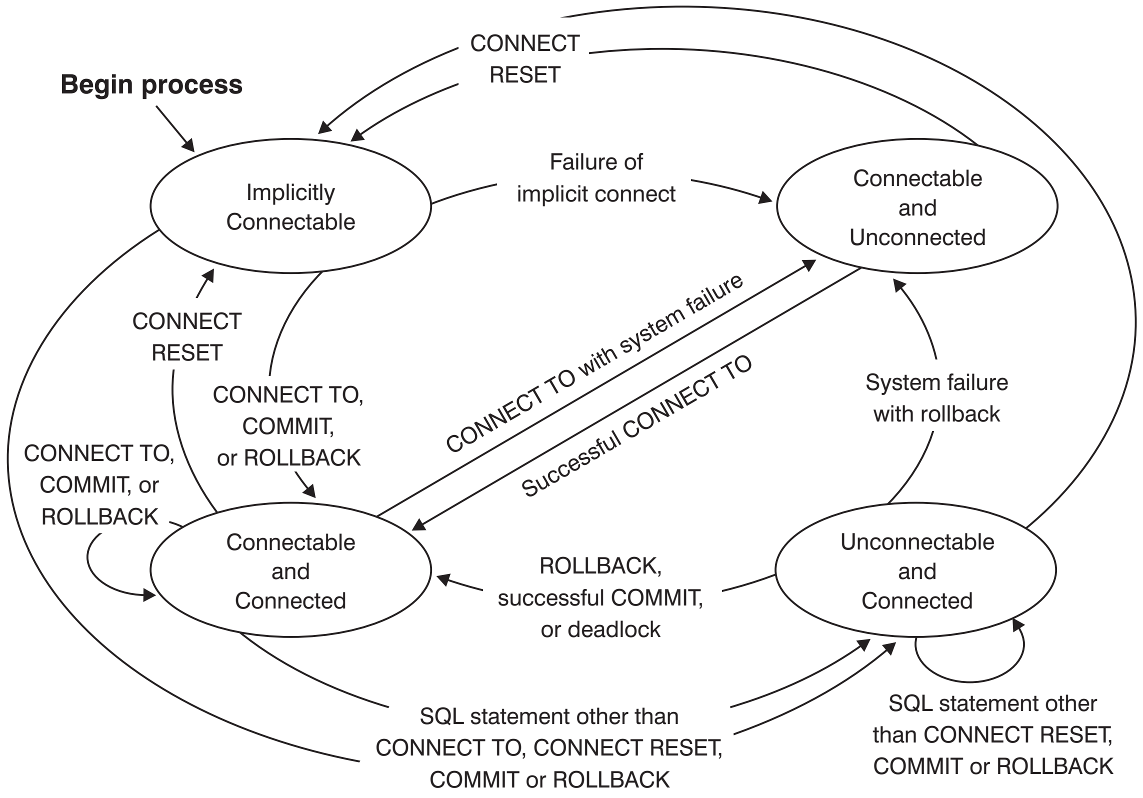

Begin process
Implicitly Connectable
Connectable and Unconnected
Connectable and Connected
Unconnectable and Connected
CONNECT RESET
Failure of implicit connect
CONNECT RESET
CONNECT TO, COMMIT, or ROLLBACK
CONNECT TO, COMMIT, or ROLLBACK
CONNECT TO with system failure
Successful CONNECT TO
System failure with rollback
ROLLBACK, successful COMMIT, or deadlock
SQL statement other than CONNECT TO, CONNECT RESET, COMMIT or ROLLBACK
SQL statement other than CONNECT RESET, COMMIT or ROLLBACK

Figure 6. Connection State Transitions If Implicit Connect Is Available

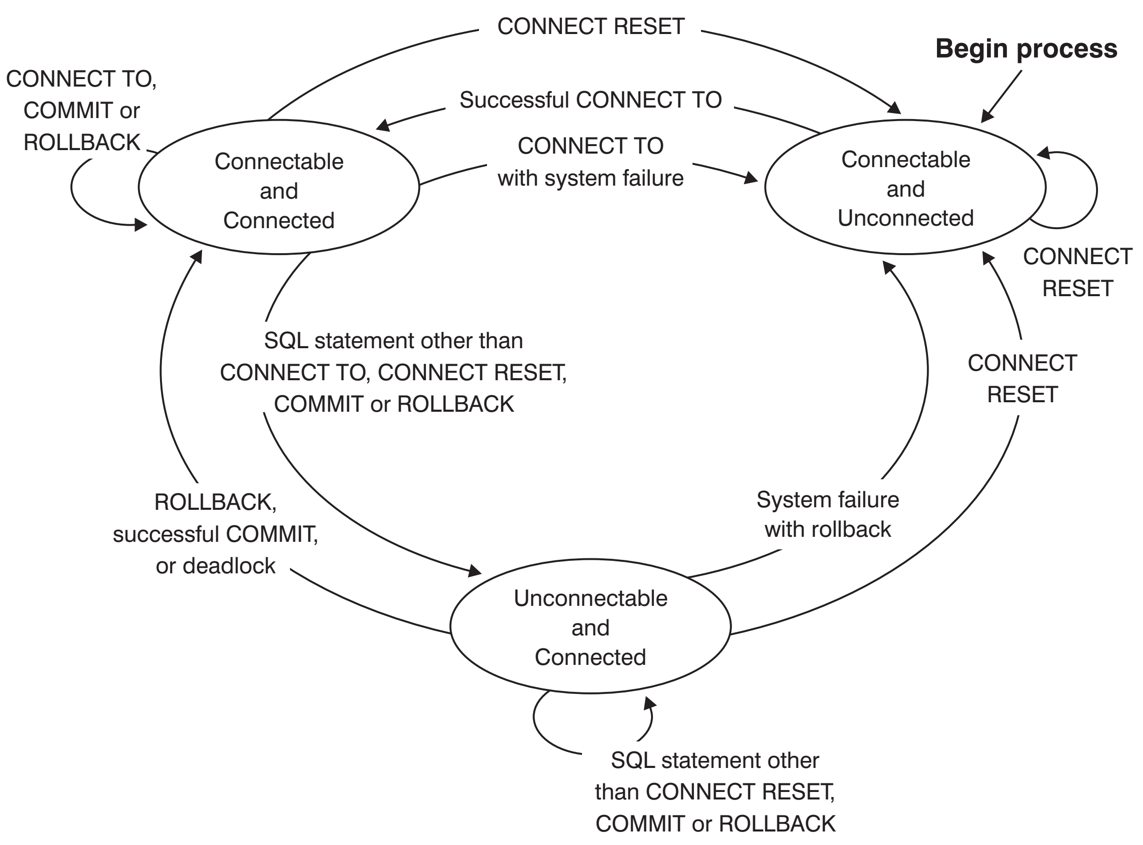

CONNECT RESET
Begin process
CONNECT TO, COMMIT or ROLLBACK
Successful CONNECT TO
CONNECT TO with system failure
Connectable and Connected
Connectable and Unconnected
CONNECT RESET
SQL statement other than CONNECT TO, CONNECT RESET, COMMIT or ROLLBACK
CONNECT RESET
ROLLBACK, successful COMMIT, or deadlock
System failure with rollback
Unconnectable and Connected
SQL statement other than CONNECT RESET, COMMIT or ROLLBACK

Figure 7. Connection State Transitions If Implicit Connect Is Not Available

Application-directed distributed unit of work

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

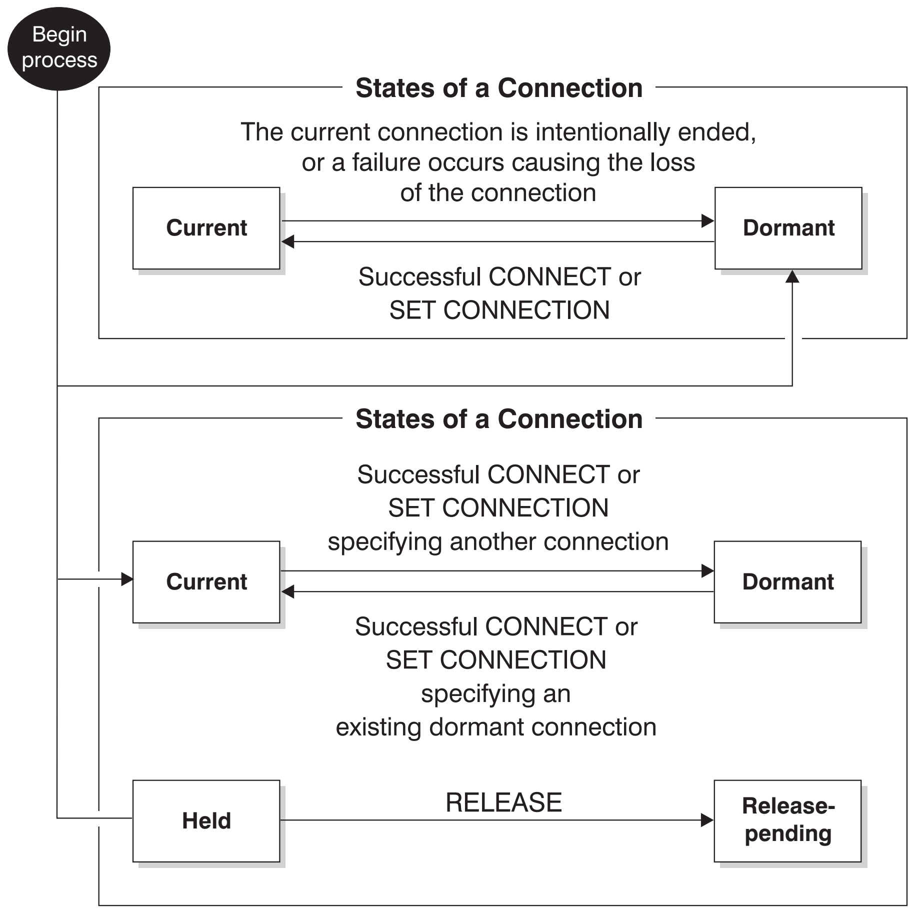

Begin process
States of a Connection
The current connection is intentionally ended, or a failure occurs causing the loss of the connection
Current
Dormant
Successful CONNECT or SET CONNECTION
States of a Connection
Successful CONNECT or SET CONNECTION specifying another connection
Current
Dormant
Successful CONNECT or SET CONNECTION specifying an existing dormant connection
Held
RELEASE
Release-pending

Figure 8. Application-Directed Distributed Unit of Work Connection State Transitions

Application process connection states

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 distributed unit of work semantics" on page 34.

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 has been 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

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:

- The current connection is placed into the *dormant connection state*, and
- 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 may still be disconnected by a commit operation if the commit operation satisfies the conditions of the DISCONNECT precompiler option.

Current® and dormant states. Regardless of whether a connection is in the held state or the release-pending state, 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 distributed unit of work semantics

The semantics of type 2 connection management are determined by a set of precompiler options. These options are summarized below 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:
  – 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 above 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.

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*
- "CONNECT (Type 2) statement" in the *SQL Reference, Volume 2*

DB2 federated systems

Federated systems

A DB2® *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 Universal Database™ 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.

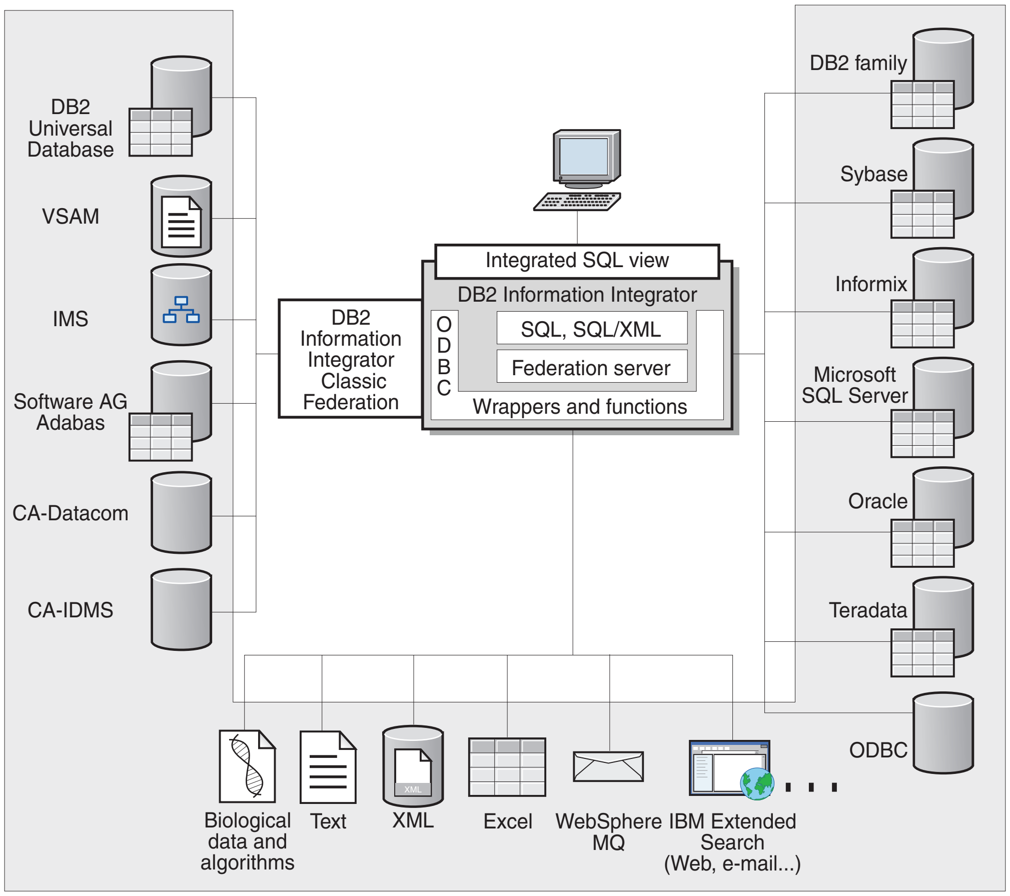


Figure 9. The components of a federated system

The power of a DB2 federated system is in its ability to:

- Join 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
- Replicate 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

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.

Related concepts:

- "What is a data source?" on page 37

What is a data source?

In a federated system, a *data source* can be a relational DBMS instance (such as Oracle or Sybase) or a nonrelational data source (such as BLAST search algorithm or an XML tagged file). Through some data sources you can access other data sources. For example, through the Extended Search data source you can access data sources such as Lotus® Notes databases, Microsoft® Access, Microsoft Index Server, Web search engines, and Lightweight Directory Access Protocol (LDAP) directories.

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™ and OS/390® data sources and the Documentum Client API/Library is used to access Documentum data sources.

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

Related concepts:

- "The federated database" on page 39

Related reference:

- "Supported data sources" on page 37

Supported data sources

There are many data sources that you can access using a federated system. The following table lists the supported data sources:

Table 1. Supported data source versions and access methods.

Data source	Supported versions	Access method
DB2 Universal Database™ for Linux, UNIX, and Windows®	7.2, 8.1, 8.2	DRDA®
DB2 Universal Database for z/OS™ and OS/390®	6.1, 7.1 with the following APARs applied: • PQ62695 • PQ55393 • PQ56616 • PQ54605 • PQ46183 • PQ62139 8.1	DRDA
DB2 Universal Database for iSeries™	5.1 • with the following APARs applied: – SE06003 – SE06872 – II13348 • with the following PTFs applied: – SI05990 SI05991 5.2 with PTF SI0735 applied.	DRDA
DB2 Server for VM and VSE	7.1 (or later) with fixes for APARs for schema functions applied.	DRDA
Informix™	7.31, 8.32, 8.4, 9.3, 9.4	Informix Client SDK V2.7 (or later)
ODBC	3.x	ODBC driver for the data source, such as Redbrick ODBC Driver to access Redbrick.
OLE DB	2.7, 2.8	OLE DB 2.0 (or later)
Oracle	8.0.6, 8.1.6, 8.1.7, 9.0, 9.1, 9.2, 9i, 10g	Oracle net client or NET8 client software
Microsoft SQL Server	7.0, 2000 SP3 and later service packs on that release	On Windows, the Microsoft SQL Server Client ODBC 3.0 (or later) driver. On UNIX, the DataDirect Technologies (formerly MERANT) Connect ODBC 3.7 (or later) driver.
Sybase	11.9.2, 12.x	Sybase Open Client ctlib interface

Table 1. Supported data source versions and access methods. (continued)

Data source	Supported versions	Access method
Teradata	V2R3, V2R4, V2R5	Teradata Call-Level Interface, Version 2 (CLIv2) Release 04.06 (or later)
BLAST	2.2.3 and later 2.2 fixpacks supported	BLAST daemon (supplied with the wrapper)
BioRS	v5.0.14	None
Documentum	3.x, 4.x	Documentum Client library/APL3.1.7a (or later)
Entrez (PubMed and GenBank data sources)	1.0	None
HMMER	2.2g, 2.3	HMMER daemon (supplied with the wrapper)
IBM Lotus Extended Search	4.0.1, 4.0.2	Extended Search Client Library (supplied with the wrapper)
Microsoft Excel	97, 2000, 2002, 2003	Excel 97, 2000, 2002, or 2003 installed on the federated server
PeopleSoft	8.x	IBM WebSphere Business Integration Adapter for PeopleSoft v2.3.1, 2.4
SAP	3.x, 4.x	IBM WebSphere Business Integration Adapter for mySAP.com v2.3.1, 2.4
Siebel	7, 7.5, 2000	IBM WebSphere Business Integration Adapter for Siebel eBusiness Applications v2.3.1, 2.4
Table-structured files		None
User-defined functions for KEGG	Supported	
User-defined functions for Life Sciences	Supported	
Web services	SOAP 1.0., 1.1, WSDL 1.0, 1.1 specifications	HTTP
XML	1.0 specification	None

Related concepts:
- "What is a data source?" on page 37

The federated database

To end users and client applications, data sources appear as a single collective database in DB2®. Users and applications interface with the *federated database* managed by the federated server. The federated database contains a system catalog. 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 sources were ordinary relational tables or views within the federated database. As a result:

- The federated system can join 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:
  - Suppose the code page used by the federated server is different than the code page used by the data source. 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.
  - Suppose the collating sequence used by the federated server is different than the collating sequence used by the data source. Any sort operations on character data are performed at the federated server instead of at the data source.

Related concepts:

- "The SQL Compiler" on page 41
- "The federated database system catalog" on page 40

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, and index 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 data sources, the information stored in the global catalog includes both remote and local information.

For nonrelational data sources, the information stored in the global catalog varies from data source to data source.

To see the data source table information that is stored in the global catalog, query the SYSCAT.TABLES, SYSCAT.TABOPTIONS, SYSCAT.INDEXES, 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.

Related concepts:
- "The SQL Compiler" on page 41

Related reference:
- "Views in the global catalog table containing federated information" in the *Federated Systems Guide*

The SQL Compiler

To obtain data from data sources, users and applications submit queries in DB2® 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 attributes, mappings, index information, and processing statistics.

Related concepts:
- "Wrappers and wrapper modules" on page 44
- "The query optimizer" on page 41

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

DB2® UDB 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. DB2 UDB 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 local and remote access plans for processing a query fragment, based on resource cost. DB2 UDB then chooses the plan it believes will process the query with the least resource cost.

If any of the fragments are to be processed by data sources, DB2 UDB submits these fragments to the data sources. After the data sources process the fragments, the results are retrieved and returned to DB2 UDB. If DB2 UDB performed any part of the processing, it combines its results with the results retrieved from the data source. DB2 UDB then returns all results to the client.

Related concepts:
- "The SQL Compiler" on page 41
- "Compensation" on page 42
- "Tuning query processing" in the *Federated Systems Guide*

Compensation

The DB2® federated server does not push down a query fragment if the data source cannot process it, or if the federated server can process it faster than the data source can process it. For example, suppose that the SQL dialect of a data source does not support a CUBE grouping in the GROUP BY clause. A query that contains a CUBE grouping and references a table in that data source is submitted to the federated server. DB2 Information Integrator does not pushdown the CUBE grouping to the data source, but processes the CUBE itself. The ability by DB2 Information Integrator to process SQL that is not supported by a data source is called *compensation*.

The federated server compensates for lack of functionality at the data source in two ways:
- It can ask the data source to use one or more operations that are equivalent to the DB2 function stated in the query. Suppose a data source does not support the cotangent (COT(x)) function, but supports the tangent (TAN(x)) function. DB2 Information Integrator can ask the data source to perform the calculation (1/TAN(x)), which is equivalent to the cotangent (COT(x)) function.
- It can return the set of data to the federated server, and perform the function locally.

For relational data sources, each type of RDBMS supports a subset of the international SQL standard. In addition, some types of RDBMSs support SQL constructs that exceed this standard. An *SQL dialect*, is the totality of SQL that a type of RDBMS supports. If an SQL construct is found in the DB2 SQL dialect, but not in the relational data source dialect, the federated server can implement this construct on behalf of the data source.

DB2 Information Integrator can compensate for differences in SQL dialects. An example of this ability is the common-table-expression clause. DB2 SQL includes the clause common-table-expression. In this clause, a name can be specified by which all FROM clauses in a fullselect can reference a result set. The federated server will process a common-table-expression for a data source, even though the SQL dialect used by the data source does not include common-table-expression.

With compensation, the federated server can support the full DB2 SQL dialect for queries of data sources. Even data sources with weak SQL support or no SQL support will benefit from compensation. You must use the DB2 SQL dialect with a federated system, except in a pass-through session.

Related concepts:

- "Pass-through sessions" on page 43

Pass-through sessions

You can submit SQL statements directly to data sources by using a special mode called *pass-through*. You submit SQL statements in the SQL dialect used by the data source. Use a pass-through session when you want to perform an operation that is not possible with the DB2® SQL/API. For example, use a pass-through session to create a procedure, create an index, or perform queries in the native dialect of the data source.

Currently, the data sources that support pass-through, support pass-through using SQL. In the future, it is possible that data sources will support pass-though using a data source language other than SQL.

Similarly, you can use a pass-through session to perform actions that are not supported by SQL, such as certain administrative tasks. However, you cannot use a pass-through session to perform all administrative tasks. For example, you can create or drop tables at the data source, but you cannot start or stop the remote database.

You can use both static and dynamic SQL in a pass-through session.

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

SET PASSTHRU
: Opens a pass-through session. When you issue another SET PASSTHRU statement to start a new pass-through session, the current pass-through session is terminated.

SET PASSTHRU RESET
: Terminates the current pass-through session.

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

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

The following restrictions apply to pass-through sessions:

- You must use the SQL dialect or language commands of the data source — you cannot use the DB2 SQL dialect. As a result, you do not query a nickname, but the data source objects directly.
- When performing UPDATE or DELETE operations in a pass-through session, you cannot use the WHERE CURRENT OF CURSOR condition.
- | LOBs are not supported in pass-through sessions.

Related concepts:

- "Wrappers and wrapper modules" on page 44

- “Querying data sources directly with pass-through” in the *Federated Systems Guide*

Wrappers and wrapper modules

Wrappers are mechanisms by which the federated server interacts with data sources. The federated server uses routines stored in a library called a *wrapper module* to implement a wrapper. These routines allow the federated server to perform operations such as connecting to a data source and retrieving data from it iteratively. Typically, the DB2® 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, suppose that you want to access three DB2 for z/OS™ database tables, one DB2 for iSeries™ table, two Informix® tables, and one Informix view. You need to create one wrapper for the DB2 data source objects and one wrapper for the Informix data source objects. Once 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 for Linux, UNIX®, and Windows®, DB2 for z/OS and OS/390®, DB2 for iSeries, 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 server 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 server queries about the default function mappings for a data source. The wrapper contains information that the federated server 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 DB2 Information Integrator uses the wrapper.

Related concepts:
- “Server definitions and server options” on page 46

Related tasks:

- "Trusted and fenced mode process environments" in the *IBM DB2 Information Integrator Wrapper Developer's Guide*
- "Altering a wrapper" in the *Federated Systems Guide*

Related reference:

- "ALTER WRAPPER statement" in the *SQL Reference, Volume 2*
- "Wrapper options for federated systems" on page 663
- "Default wrapper names" on page 45

Default wrapper names

There are wrappers for each supported data source. Some wrappers have default wrapper names. When you use the default name to create the wrapper, the federated server automatically picks up the data source library associated with the wrapper.

Table 2. Default wrapper names for each data source.

Data source	Default wrapper names
DB2 Universal Database™ for Linux, UNIX and Windows®	DRDA
DB2 Universal Database for z/OS and OS/390®	DRDA
DB2 Universal Database for iSeries	DRDA
DB2 Server for VM and VSE	DRDA
Informix	INFORMIX
Microsoft® SQL Server	MSSQLODBC3
ODBC	ODBC
OLE DB	OLEDB
Oracle	NET8
Sybase	CTLIB
Teradata	TERADATA
BLAST	None
BioRS	None
Documentum	None
Entrez	None
Extended Search	None
HMMER	None
Microsoft Excel	None
Table-structured files	None
Web Services	None
WebSphere Business Integration	None
XML	None

Related concepts:

- "Wrappers and wrapper modules" on page 44

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. Some server options configure the wrapper and some affect the way DB2 Information Integrator uses the wrapper.

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

Related concepts:

- "User mappings" on page 46

Related reference:

- "Server options for federated systems" on page 648

User mappings

When a federated server needs to pushdown a request to a data source, the server must first establish a connection to the data source.

For most data sources, the federated server does this by using a valid user ID and password to that data source. When a user ID and password is required to connect to a data source, you can define an association between the federated server authorization ID and the data source user ID and password. This association can be created for each user ID that will be using the federated system to send distributed requests. This association is called a *user mapping*.

In some cases, you do not need to create a user mapping if the user ID and password you use to connect to the federated database are the same as those you use to access the remote data source.

Related tasks:

- “Registering user mappings for a data source” in the *IBM DB2 Information Integrator Data Source Configuration Guide*

Nicknames and data source objects

After you create the server definitions and user mappings, the federated instance owner creates the nicknames. A *nickname* is an identifier that is used to reference the object located at the data sources that you want to access. The objects that nicknames identify are referred to as *data source objects*.

Nicknames are not alternative names for data source objects in the same way that aliases are alternative names. They are pointers by which the federated server references these objects. Nicknames are typically defined with the CREATE NICKNAME statement along with specific nickname column options and nickname options.

When an end user or a client application 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 end user or the client application.

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, and the information in the wrapper, to facilitate access to the data source object. For example, if the nickname is for a table that has an index, the global catalog contains information about the index. The wrapper contains the mappings between the DB2® data types and the data source data types.

Currently, you cannot execute some DB2 UDB utility operations on nicknames.

You cannot use the Cross Loader utility to cross load into a nickname.

Related concepts:
- “Nickname column options” on page 48

Related reference:
- “Nickname column options for federated systems” on page 641
- “Nickname options for federated systems” in the *Federated Systems Guide*
- “Valid data source objects” on page 47

Valid data source objects

Nicknames identify objects at the data source that you want to access. The following table lists the types of objects that you can create a nickname for in a federated system.

Table 3. Valid data source objects

Data source	Valid objects
DB2 for Linux, UNIX, and Windows	Nicknames, materialized query tables, tables, views
DB2 for z/OS and OS/390	Tables, views
DB2 for iSeries	Tables, views
DB2 for VM and VSE	Tables, views
Informix	Tables, views, synonyms
Microsoft SQL Server	Tables, views
ODBC	Tables, views
Oracle	Tables, views, synonyms
Sybase	Tables, views
Teradata	Tables, views
BLAST	FASTA files indexed for BLAST search algorithms
BioRS	BioRS databanks
Documentum	Objects and registered tables in a Documentum Docbase
Entrez	Entrez databases
Extended Search	Files from data sources such as Lotus Notes databases, Microsoft Access, Microsoft Index Server, Web search engines, and LDAP directories.
HMMER	HMM database files (libraries of Hierarchical Markov Models, such as PFAM), that can be searched by HMMER’s hmmpfam or hmmsearch programs.
Microsoft Excel	.xls files (only the first sheet in the workbook is accessed)
Table-structured files	Text files that meet a specific format.
Websphere Business Integration adapters	Websphere Business Integration business objects that map to BAPIs in SAP, business components in Siebel, and component interfaces in PeopleSoft
Web Services	Operations in a Web services description language file
XML-tagged files	Sets of items in an XML document

Related concepts:
- “Nicknames and data source objects” on page 47
- “Nickname column options” on page 48

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

Related concepts:
- "Data type mappings" on page 49

Related tasks:
- "Working with nicknames" in the *Federated Systems Guide*

Related reference:
- "Nickname column options for federated systems" on page 641

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 for Linux, UNIX®, and

Windows® 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.

| You should create new type mappings or modify the default type mappings before
| you create nicknames. Otherwise nicknames created before the type mapping
| changes will not reflect the new mappings.

Related concepts:
- "Data type mappings in a federated system" in the *Federated Systems Guide*

Function mappings

For the federated server to recognize a data source function, the function must be mapped to an existing counterpart function in DB2® for Linux, UNIX® and Windows®. DB2 Information Integrator supplies default mappings between existing built-in data source functions and built-in DB2 counterpart functions. For most data sources, the default function mappings are in the wrappers. The default function mappings to DB2 for z/OS™ and OS/390® functions are in the DRDA® wrapper. The default function mappings to Sybase functions are in the CTLIB wrapper, and so forth.

| For relational data sources, you can create a function mapping when you want to
| use a data source function that the federated server does not recognize. The
| mapping that you create is between the data source function and a DB2
| counterpart function at the federated database. Function mappings are typically
| used when a new built-in function or a new user-defined function become
| available at the data source. Function mappings are also used when a DB2
| counterpart function does not exist. In this case, you must also create a function
| template.

Related concepts:
- "Function mappings in a federated system" in the *Federated Systems Guide*
- "Index specifications" on page 50

Index specifications

When you create a nickname for a data source table, information about any indexes that the data source table has is added to the global catalog. The query optimizer uses this information to expedite the processing of distributed requests. The catalog information about a data source index is a set of metadata, and is called an *index specification*. A federated server does not create an index specification when you create a nickname for:
- A table that has no indexes
- A view, which typically does not have any index information stored in the remote catalog
- A data source object that does not have a remote catalog from which the federated server can obtain the index information

Suppose that a table acquires a new index, in addition to the ones it had when the nickname was created. Because index information is supplied to the global catalog at the time the nickname is created, the federated server is unaware of the new index. Similarly, when a nickname is created for a view, the federated server is unaware of the underlying table (and its indexes) from which the view was generated. In these circumstances, you can supply the necessary index information to the global catalog. You can create an index specification for tables that have no indexes. The index specification tells the query optimizer which column or columns in the table to search on to find data quickly.

Related concepts:

- "Index specifications in a federated system" in the *Federated Systems Guide*

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.

Suppose that database ALPHA uses the default collating sequence of the EBCDIC code page, and that database BETA uses the default collating sequence of the

ASCII code page. Sort orders for character strings at these two databases would differ, as shown in the following example:

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

EBCDIC-Based Sort            ASCII-Based Sort

COL2                         COL2
----                         ----
V1G                          7AB
Y2W                          V1G
7AB                          Y2W
```

Similarly, character comparisons in a database depend on the collating sequence defined for that database. In this 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. Character comparisons at these two databases would yield different results, as shown in the following example:

```
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 API. Through this API, you can specify one of the following sequences:

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

Suppose that the data source is DB2 for z/OS and OS/390. 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 and OS/390 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.

Related concepts:

- "Server characteristics affecting pushdown opportunities" in the *Federated Systems Guide*
- "Collating sequences in a federated system" in the *IBM DB2 Information Integrator Data Source Configuration Guide*

Related tasks:

- "Creating a federated database" in the *IBM DB2 Information Integrator Data Source Configuration Guide*

Related reference:

- "National language versions" in the *Administration Guide: Planning*
- "Federated database national language considerations" in the *IBM DB2 Information Integrator Data Source Configuration Guide*

Chapter 2. Language elements

This chapter describes the language elements that are common to many SQL statements:

- "Characters"
- "Tokens" on page 57
- "Identifiers" on page 58
- "Data types" on page 79
- "Constants" on page 121
- "Special registers" on page 124
- "Functions" on page 152
- "Methods" on page 160
- "Expressions" on page 168
- "Predicates" on page 204

Characters

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

A *letter* is any of the 26 uppercase (A through Z) or 26 lowercase (a through z) letters, plus the three characters $, #, and @, which are included for compatibility with host database products. For example, in code page 850, $ is at X'24', # is at X'23', and @ is at X'40'). Letters also include the 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 below:

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

Characters

Character	Description	Character	Description
|	vertical bar[1]	^	caret
!	exclamation mark	[	left bracket
{	left brace	]	right bracket
}	right brace		

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

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

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.

Related reference:

- "How SQL statements are invoked" in the *SQL Reference, Volume 2*
- "PREPARE statement" in the *SQL Reference, Volume 2*

Identifiers

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

- SQL identifiers

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

  – An *ordinary identifier* is a letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier should not be identical to a reserved word.

    Examples

    ```
    WKLYSAL    WKLY_SAL
    ```

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

    Examples

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

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

- Host identifiers

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

Naming conventions and implicit object name qualifications

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

In dynamic SQL statements, a schema-qualified object name implicitly uses the CURRENT SCHEMA special register value as the qualifier for unqualified object name references. By default it is set to the current authorization ID. If the dynamic SQL statement is contained in a package that exhibits bind, define, or invoke behaviour, the CURRENT SCHEMA special register is not used for qualification. In a bind behaviour package, the package default qualifier is used as the value for implicit qualification of unqualified object references. In a define behaviour package, the authorization ID of the routine definer is used as the value for implicit qualification of unqualified object references within that routine. In an invoke behaviour package, the statement authorization ID in effect when the routine is invoked is used as the value for implicit qualification of unqualified object references within dynamic SQL statements within that routine. For more information, see "Dynamic SQL characteristics at run time" on page 64.

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

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

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

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

alias-name	A schema-qualified name that designates an alias.
attribute-name	An identifier that designates an attribute of a structured data type.
authorization-name	An identifier that designates a user or a group: • Valid characters are A through Z, a through z, 0 through 9, #, @, $, and _. • The name must not begin with the characters 'SYS', 'IBM', or 'SQL'. • The name must not be: ADMINS, GUESTS, LOCAL, PUBLIC, or USERS. • A delimited authorization ID must not contain lowercase letters.
bufferpool-name	An identifier that designates a bufferpool.
column-name	A qualified or unqualified name that designates a column of a table or view. The qualifier is a table name, a view name, a nickname, or a correlation name.
condition-name	An identifier that designates a condition in an SQL procedure.
constraint-name	An identifier that designates a referential constraint, primary key constraint, unique constraint, or a table check constraint.
correlation-name	An identifier that designates a result table.
cursor-name	An identifier that designates an SQL cursor. For host compatibility, a hyphen character may be used in the name.
data-source-name	An identifier that designates a data source. This identifier is the first part of a three-part remote object name.
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 72. Note that a descriptor name never includes an indicator variable.

distinct-type-name	A qualified or unqualified name that designates a distinct type. An unqualified distinct type name in an SQL statement is implicitly qualified by the database manager, depending on context.
event-monitor-name	An identifier that designates an event monitor.
function-mapping-name	An identifier that designates a function mapping.
function-name	A qualified or unqualified name that designates a function. An unqualified function name in an SQL statement is implicitly qualified by the database manager, depending on context.
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 72.
index-name	A schema-qualified name that designates an index or an index specification.
label	An identifier that designates a label in an SQL procedure.
method-name	An identifier that designates a method. The schema context for a method is determined by the schema of the subject type (or a supertype of the subject type) of the method.
nickname	A schema-qualified name that designates a federated server reference to a table or a view.
db-partition-group-name	An identifier that designates a database partition group.
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.
procedure-name	A qualified or unqualified name that designates a procedure. An unqualified procedure name in an SQL statement is implicitly qualified by the database manager, depending on context.
remote-authorization-name	An identifier that designates a data source user. The rules for authorization names vary from data source to data source.
remote-function-name	A name that designates a function registered to a data source database.
remote-object-name	A three-part name that designates a data source table or view, and that identifies the data source in

	which the table or view resides. The parts of this name are data-source-name, remote-schema-name, and remote-table-name.
remote-schema-name	A name that designates the schema to which a data source table or view belongs. This name is the second part of a three-part remote object name.
remote-table-name	A name that designates a table or view at a data source. This name is the third part of a three-part remote object name.
remote-type-name	A data type supported by a data source database. Do not use the long form for built-in types (use CHAR instead of CHARACTER, for example).
savepoint-name	An identifier that designates a savepoint.
schema-name	An identifier that provides a logical grouping for SQL objects. A schema name used as a qualifier for the name of an object may be implicitly determined: • from the value of the CURRENT SCHEMA special register • from the value of the QUALIFIER precompile/bind option • on the basis of a resolution algorithm that uses the CURRENT PATH special register • on the basis of the schema name for another object in the same SQL statement. To avoid complications, it is recommended that the name SESSION not be used as a schema, except as the schema for declared global temporary tables (which *must* use the schema name SESSION).
server-name	An identifier that designates an application server. In a federated system, the server name also designates the local name of a data source.
specific-name	A qualified or unqualified name that designates a specific name. An unqualified specific name in an SQL statement is implicitly qualified by the database manager, depending on context.
SQL-variable-name	The name of a local variable in an SQL procedure statement. SQL variable names can be used in other SQL statements where a host variable name is allowed. The name can be qualified by the label of the compound statement that declared the SQL variable.
statement-name	An identifier that designates a prepared SQL statement.
supertype-name	A qualified or unqualified name that designates the supertype of a type. An unqualified supertype name in an SQL statement is implicitly qualified by the database manager, depending on context.
table-name	A schema-qualified name that designates a table.

tablespace-name	An identifier that designates a table space.
trigger-name	A schema-qualified name that designates a trigger.
type-mapping-name	An identifier that designates a data type mapping.
type-name	A qualified or unqualified name that designates a type. An unqualified type name in an SQL statement is implicitly qualified by the database manager, depending on context.
typed-table-name	A schema-qualified name that designates a typed table.
typed-view-name	A schema-qualified name that designates a typed view.
view-name	A schema-qualified name that designates a view.
wrapper-name	An identifier that designates a wrapper.

Aliases

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

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

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

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

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined by the time that the SQL statement is compiled, is replaced at statement compilation time by the qualified base table or view name. For example, if PBIRD.SALES is an alias for DSPN014.DIST4_SALES_148, then at compilation time:

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

effectively becomes

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

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

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

Authorization IDs and authorization names

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

Authorization IDs are used by the database manager to provide:

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

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

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

For more information, see "Dynamic SQL characteristics at run time" on page 64.

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 *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 58.

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

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

  ```
  DROP TABLE TDEPT
  ```

  Removes the SMITH.TDEPT table.

  ```
  DROP TABLE SMITH.TDEPT
  ```

  Removes the SMITH.TDEPT table.

  ```
  DROP TABLE KEENE.TDEPT
  ```

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

  ```
  CREATE SCHEMA PAYROLL AUTHORIZATION KEENE
  ```

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

Dynamic SQL characteristics at run time

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

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

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

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

Run behavior
DB2 uses the authorization ID of the user (the ID that initially connected to DB2) executing the package as the value to be used for authorization

checking of dynamic SQL statements and for the initial value used for implicit qualification of unqualified object references within dynamic SQL statements.

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

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

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

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

Restricted statements when run behavior does not apply

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

Considerations regarding the DYNAMICRULES option

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

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

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

Authorization IDs and statement preparation

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

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

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

Column names

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

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In a column function, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a GROUP BY or ORDER BY clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.

  – In an expression, a search condition, or a scalar function, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause.

Qualified column names

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

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

Where a qualifier is optional, it can serve two purposes. They are described under "Column name qualifiers to avoid ambiguity" on page 69 and "Column name qualifiers in correlated references" on page 70.

Correlation names

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

```
   FROM X.MYTABLE Z
```

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

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

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table, view, nickname, or alias. In the case of a nested table expression or table function, a correlation name is required to identify the result table. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

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

Example

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

The qualified reference to PROJECT should instead use the correlation name, ″E″, as shown below:

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

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

```
FROM EMPLOYEE E, DEPARTMENT
```

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

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

1. Given the FROM clause:
   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name “E1” (E1.PROJECT).
2. Given the FROM clause:
   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name “E2” (E2.PROJECT).
3. Given the FROM clause:
   ```
   FROM EMPLOYEE, EMPLOYEE
   ```
   the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same. This is allowed, but references to specific column names would be ambiguous (SQLSTATE 42702).
4. Given the following statement:
   ```
   SELECT *
     FROM EMPLOYEE E1, EMPLOYEE E2            * incorrect *
     WHERE EMPLOYEE.PROJECT = 'ABC'
   ```
   the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).
5. Given the FROM clause:
   ```
   FROM EMPLOYEE, X.EMPLOYEE
   ```
   a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). If X is the CURRENT SCHEMA

special register value in dynamic SQL or the QUALIFIER precompile/bind option in static SQL, then the columns cannot be referenced since any such reference would be ambiguous.

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

Given the FROM clause:

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

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

Column name qualifiers to avoid ambiguity

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

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 host variables

A *host variable* is either:

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

or:

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

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

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

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

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

Host variables in dynamic SQL

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

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

This example shows a dynamic SQL statement using parameter markers:

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

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

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

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

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

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

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

Similarly, if :HV1:HV2 is specified in a VALUES INTO clause or in a FETCH or SELECT INTO statement, and if the value returned is null, HV1 is not changed, and HV2 is set to a negative value. If the database is configured with DFT_SQLMATHWARN yes (or was during binding of a static SQL statement), HV2 could be -2. If HV2 is -2, a value for HV1 could not be returned because of an error converting to the numeric type of HV1, or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2 Universal Database Version 5, HV2 will be -1 for arithmetic exceptions. If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

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

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

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

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

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

References to BLOB, CLOB, and DBCLOB host variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see "References to locator variables"), and LOB file reference variables (see "References to BLOB, CLOB, and DBCLOB file reference variables" on page 75) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

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

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

References to locator variables

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

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

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

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

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

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

References to BLOB, CLOB, and DBCLOB file reference variables

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

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

A file reference variable has the following properties:

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

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

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

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

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

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

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

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

- Input (from client to server)

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

- Output (from server to client)

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

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

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

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

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

Example of an output file reference variable (in C): Given a declare section coded as:

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

Following preprocessing this would be:

```
EXEC SQL BEGIN DECLARE SECTION
   /* SQL TYPE IS CLOB_FILE  hv_text_file; */
   struct {
       unsigned long  name_length; //  File Name Length
       unsigned long  data_length; //  Data Length
```

```
        unsigned long  file_options; // File Options
        char           name[255];   // File Name
     } hv_text_file;
     char  hv_patent_title[64];
  EXEC SQL END DECLARE SECTION
```

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

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

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

Example of an input file reference variable (in C): Given the same declare section as above, the following code can be used to insert the data from a regular file referenced by :hv_text_file into a CLOB column.

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

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

References to structured type host variables

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

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

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

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

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

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

Example

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

Related reference:

- "CREATE ALIAS statement" in the *SQL Reference, Volume 2*
- "PREPARE statement" in the *SQL Reference, Volume 2*
- "SET SCHEMA statement" in the *SQL Reference, Volume 2*
- Appendix A, "SQL limits," on page 501
- Appendix C, "SQLDA (SQL descriptor area)," on page 513
- Appendix G, "Reserved schema names and reserved words," on page 703
- Appendix N, "Japanese and traditional-Chinese extended UNIX code (EUC) considerations," on page 755
- "SQL queries" on page 449
- "Large objects (LOBs)" on page 84

Data types

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
- Host variables
- Functions
- Expressions
- Special registers.

DB2 supports a number of built-in data types. It also provides support for user-defined data types. Figure 10 shows the supported built-in data types.

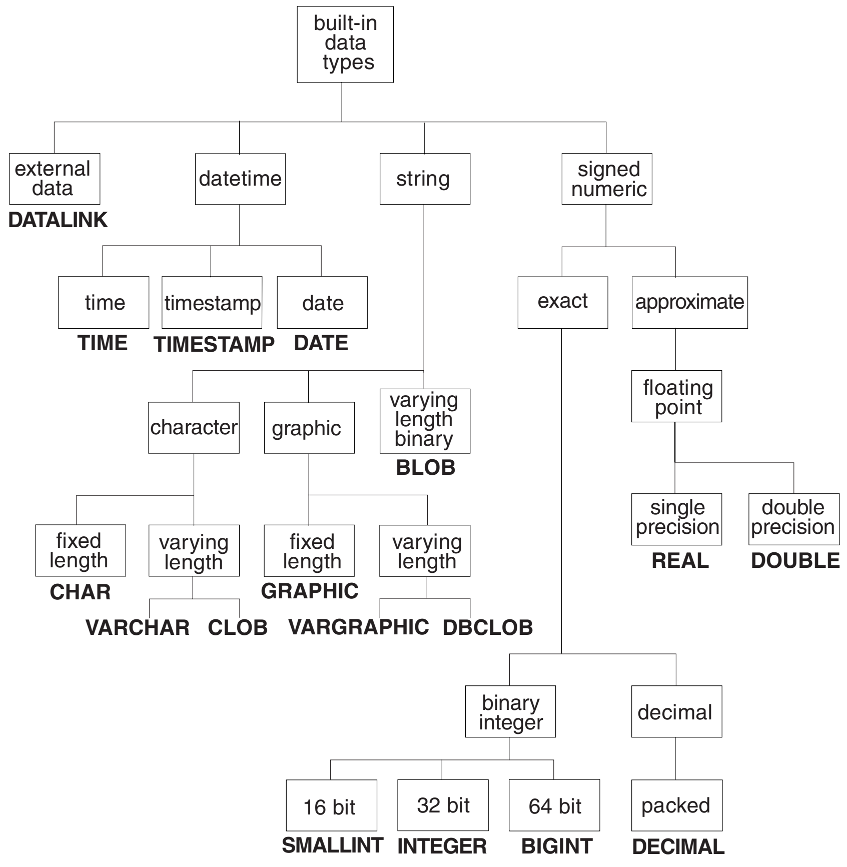


Figure 10. The DB2 Built-in Data Types

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

Related reference:

- "User-defined types" on page 91

Numbers

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

See 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.

Single-precision floating-point (REAL)

A *single-precision floating-point* number is a 32-bit approximation of a real number. The number can be zero or can range from -3.402E+38 to -1.175E-37, or from 1.175E-37 to 3.402E+38.

Double-precision floating-point (DOUBLE or FLOAT)

A *double-precision floating-point* number is a 64-bit approximation of a real number. The number can be zero or can range from -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

Decimal (DECIMAL or NUMERIC)

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

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.

Related reference:

- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- Appendix C, "SQLDA (SQL descriptor area)," on page 513

Character strings

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

Fixed-length character string (CHAR)

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

Varying-length character strings

There are three types of varying-length character string:

- A VARCHAR value can be up to 32 672 bytes long.
- A LONG VARCHAR value can be up to 32 700 bytes long.
- A CLOB (character large object) value can be up to 2 gigabytes (2 147 483 647 bytes) long. A CLOB is used to store large SBCS or mixed (SBCS and MBCS) character-based data (such as documents written with a single character set) and, therefore, has an SBCS or mixed code page associated with it.

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

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

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

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

Each character string is further defined as one of:

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

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

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

Graphic strings

A *graphic string* is a sequence of bytes that represents double-byte character data. The length of the string is the number of double-byte characters in the sequence. If the length is zero, the value is called the *empty string*. This value should not be confused with the null value.

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

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

Fixed-length graphic strings (GRAPHIC)

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

Varying-length graphic strings

There are three types of varying-length graphic string:

- A VARGRAPHIC value can be up to 16 336 double-byte characters long.
- A LONG VARGRAPHIC value can be up to 16 350 double-byte characters long.
- A DBCLOB (double-byte character large object) value can be up to 1 073 741 823 double-byte characters long. A DBCLOB is used to store large DBCS character-based data (such as documents written with a single character set) and, therefore, has a DBCS code page associated with it.

Special restrictions apply to an expression that results in a varying-length graphic string whose maximum length is greater than 127 bytes. These restrictions are the same as those specified in "Varying-length character strings" on page 81.

Binary strings

A *binary string* is a sequence of bytes. Unlike character strings, which usually contain text data, binary strings are used to hold non-traditional data such as pictures, voice, or mixed media. Character strings of the FOR BIT DATA subtype may be used for similar purposes, but the two data types are not compatible. The BLOB scalar function can be used to cast a FOR BIT DATA character string to a binary string. Binary strings are not associated with a code page. They have the same restrictions as character strings (for details, see "Varying-length character strings" on page 81).

Binary large object (BLOB)

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

Large objects (LOBs)

The term *large object* and the generic acronym LOB refer to the BLOB, CLOB, or DBCLOB data type. LOB values are subject to restrictions that apply to LONG VARCHAR values, as described in "Varying-length character strings" on page 81. These restrictions apply even if the length attribute of the LOB string is 254 bytes or less.

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

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

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

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

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

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

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

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

Datetime values

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

Date

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

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

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

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24. The range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications are zero.

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

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

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) designating a date and time as defined above, except that the time includes a fractional specification of microseconds.

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

The length of a TIMESTAMP column, as described in the SQLDA, is 26 bytes, which is the appropriate length for the character string representation of the value.

String representations of datetime values

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

String representations of datetime values

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

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

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

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

Valid string formats for dates are listed in the following table. Each format is identified by name and associated abbreviation.

Table 5. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization	ISO	yyyy-mm-dd	1991-10-27
IBM USA standard	USA	mm/dd/yyyy	10/27/1991
IBM European standard	EUR	dd.mm.yyyy	27.10.1991
Japanese Industrial Standard Christian Era	JIS	yyyy-mm-dd	1991-10-27
Site-defined	LOC	Depends on the territory code of the application	—

Time strings: A string representation of a time is a string that starts with a digit and has a length of at least 4 characters. Trailing blanks may be included; a leading zero may be omitted from the hour part of the time, and seconds may be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13:30 is equivalent to 13:30:00.

Valid string formats for times are listed in the following table. Each format is identified by name and associated abbreviation.

Table 6. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization[2]	ISO	hh.mm.ss	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh.mm.ss	13.30.05
Japanese Industrial Standard Christian Era	JIS	hh:mm:ss	13:30:05
Site-defined	LOC	Depends on the territory code of the application	—

Notes:

1. In ISO, EUR, and JIS format, .ss (or :ss) is optional.
2. The International Standards Organization changed the time format so that it is identical with 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 may be omitted, indicating an implicit specification of 00 minutes. Thus 1 PM is equivalent to 1:00 PM.
4. In the USA time format, the hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM. There is a single space before the AM and PM. Using the JIS format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

   12:01 AM through 12:59 AM corresponds to 00:01:00 through 00:59:00.

   01:00 AM through 11:59 AM corresponds to 01:00:00 through 11:59:00.

   12:00 PM (noon) through 11:59 PM corresponds to 12:00:00 through 23:59:00.

   12:00 AM (midnight) corresponds to 24:00:00 and 00:00 AM (midnight) corresponds to 00:00:00.

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

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

DATALINK values

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

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

data location
The location of a file linked with a reference within DB2, in the form of a URL. The allowed scheme names for this URL are:

- HTTP
- FILE
- UNC

The other parts of the URL are:

- the file server name for the HTTP, FILE, and UNC schemes
- the full file path name within the file server

comment
Up to 200 bytes of descriptive information, *including* the data location attribute. This is intended for application-specific uses, such as further or alternative identification of the location of the data.

Leading and trailing blank characters are trimmed while parsing data location attributes as URLs. Also, the scheme names ('http', 'file', 'unc') and host are case-insensitive and are always stored in the database in uppercase. When a DATALINK value is fetched from a database, an access token is embedded within the URL attribute, if the DATALINK column is defined with READ PERMISSION DB or WRITE PERMISSION ADMIN. The token is generated dynamically, and is not a permanent part of the DATALINK value stored in the database.

It is possible for a DATALINK value to have only a comment attribute and an empty data location attribute. Such a value may even be stored in a column but, of course, no file will be linked to such a column. The total length of the comment and the data location attribute of a DATALINK value is currently limited to 200 bytes.

It is important to distinguish between DATALINK references to files and LOB file reference variables. The similarity is that they both contain a representation of a file. However:

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

Use built-in scalar functions to build a DATALINK value (DLVALUE, DLNEWCOPY, DLPREVIOUSCOPY, and DLREPLACECONTENT) and to extract the encapsulated values from a DATALINK value (DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLSERVER, DLURLCOMPLETEONLY, DLURLCOMPLETEWRITE, and DLURLPATHWRITE).

Related reference:

- "Identifiers" on page 58

- Appendix O, “Backus-Naur form (BNF) specifications for DATALINKs,” on page 761

XML values

The XML data type is an internal representation of XML, and can only be used as input to functions that accept this data type as input. XML is a transient data type that cannot be stored in the database, or returned to an application.

Valid values for the XML data type include:
- An element
- A forest of elements
- The textual content of an element
- An empty XML value

Currently, the only supported operation is to serialize (by using the XML2CLOB function) the XML value into a string that is stored as a CLOB value.

User-defined types

There are three types of user-defined data type:

- Distinct type
- Structured type
- Reference type

Each of these types is described in the following sections.

Distinct types

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for most operations. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

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

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

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

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

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

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

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type. This can be done by creating user-defined functions that are sourced on functions defined on the source type of the distinct type. The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, or DATALINK as the source type. In addition, functions are generated to support casting from the source type to the distinct type, and from the distinct type to the source type.

Structured types

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

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

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

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

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

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

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

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

Reference types

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

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

Related reference:
- "DROP statement" in the *SQL Reference, Volume 2*
- "CURRENT PATH" on page 140
- "Character strings" on page 81
- "Assignments and comparisons" on page 99

Promotion of data types

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

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

Table 7 shows the precedence list (in order) for each data type and can be used to determine the data types to which a given data type can be promoted. The table shows that the best choice is always the same data type instead of choosing to promote to another data type.

Table 7. Data Type Precedence Table

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

Table 7. Data Type Precedence Table (continued)

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

Notes:

1. The lowercase types above are defined as follows:
   - decimal = DECIMAL(p,s) or NUMERIC(p,s)
   - real = REAL or FLOAT(*n*), where *n* is not greater than 24
   - double = DOUBLE, DOUBLE-PRECISION, FLOAT or FLOAT(*n*), where *n* is greater than 24
   - udt = a user-defined type

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

Related reference:

- "Functions" on page 152
- "Casting between data types" on page 96
- "Assignments and comparisons" on page 99

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.

Casting between data types can be done explicitly using the CAST specification but may also occur implicitly during assignments involving user-defined types. Also, when creating sourced user-defined functions, the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 8 on page 97. The first column represents the data type of the cast operand (source data type), and the data types across the top represent the target data type of the CAST specification.

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

The following casts involving distinct types are supported:

- Cast from distinct type *DT* to its source data type *S*
- Cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- Cast from distinct type *DT* to the same distinct type *DT*
- Cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT*
- Cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- Cast from a DOUBLE to distinct type *DT* with a source data type REAL
- Cast from a VARCHAR to distinct type *DT* with a source data type CHAR
- Cast from a VARGRAPHIC to distinct type *DT* with a source data type GRAPHIC
- For a Unicode database, cast from a VARCHAR or a VARGRAPHIC to distinct type *DT* with a source data type CHAR or GRAPHIC.

It is not possible to specify FOR BIT DATA when performing a cast to a character type.

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

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

The following casts involving reference types are supported:

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

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

Table 8. Supported Casts between Built-in Data Types

Target Data Type → / Source Data Type ↓	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	CHAR	VARCHAR	LONG VARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONG VARG	DBCLOB	DATE	TIME	TIMESTAMP	BLOB
SMALLINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y[1]	Y[1]	-	-	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	Y[1]	Y[1]	-	-	Y	Y	Y	Y
LONG VARCHAR	-	-	-	-	-	-	Y	Y	Y	Y	-	-	Y[1]	Y[1]	-	-	-	Y
CLOB	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y[1]	-	-	-	Y
GRAPHIC	-	-	-	-	-	-	Y[1]	Y[1]	-	-	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y
VARGRAPHIC	-	-	-	-	-	-	Y[1]	Y[1]	-	-	Y	Y	Y	Y	Y[1]	Y[1]	Y[1]	Y
LONG VARGRAPHIC	-	-	-	-	-	-	-	-	Y[1]	Y[1]	Y	Y	Y	Y	-	-	-	Y
DBCLOB	-	-	-	-	-	-	-	-	-	Y[1]	Y	Y	Y	Y	-	-	-	Y
DATE	-	Y	Y	Y	-	-	Y	Y	-	-	Y[1]	Y[1]	-	-	Y	-	-	-
TIME	-	Y	Y	Y	-	-	Y	Y	-	-	Y[1]	Y[1]	-	-	-	Y	-	-
TIMESTAMP	-	-	Y	Y	-	-	Y	Y	-	-	Y[1]	Y[1]	-	-	Y	Y	Y	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

Notes

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- Only a DATALINK type can be cast to a DATALINK type.
- It is not possible to cast a structured type value to anything else.

[1] Cast is only supported for Unicode databases.

Related reference:

- “Expressions” on page 168
- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*
- “CURRENT PATH” on page 140
- “Promotion of data types” on page 94
- “Assignments and comparisons” on page 99

Assignments and comparisons

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

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

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

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

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

Table 9. Data Type Compatibility for Assignments and Comparisons

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

Table 9. Data Type Compatibility for Assignments and Comparisons (continued)

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

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

- Datetime values can be assigned to string columns and to string variables.
- A valid string representation of a date can be assigned to a date column or compared with a date.
- A valid string representation of a time can be assigned to a time column or compared with a time.
- A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.

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

[2] A user-defined distinct type value is only comparable to a value defined with the same user-defined distinct type. In general, assignments are supported between a distinct type value and its source data type. A user-defined structured type is not comparable and can only be assigned to an operand of the same structured type or one of its supertypes. For additional information see “User-defined type assignments” on page 106.

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

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

[5] For information on assignment and comparison of reference types, see “Reference type assignments” on page 107 and “Reference type comparisons” on page 112.

[6] Only supported for Unicode databases.

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

Numeric assignments

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

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

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

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

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

Floating-point to decimal: When a floating-point number is converted to decimal, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number less than $0.5*10^{-31}$ is reduced to 0. The scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

String assignments

There are two types of assignments:

- *storage assignment* is when a value is assigned to a column or parameter of a routine
- *retrieval assignment* is when a value is assigned to a host variable.

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

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

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

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

Furthermore, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

If a character string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of 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 specified with the PREP or BIND command.

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

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

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

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

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

DBCS considerations for graphic string assignments: Graphic string assignments are processed in a manner analogous to that for character strings. For non-Unicode databases, graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types. For Unicode databases, graphic string data types are compatible with character string data types.

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

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

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

The indicator variable, if specified, will contain the original length (in double-byte characters) of the source string. In the case of DBCLOB, however, the indicator variable does not contain the original length.

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

Datetime assignments

The basic rule for datetime assignments is that a DATE, TIME, or TIMESTAMP value can only be assigned to a column with a matching data type (whether DATE, TIME, or TIMESTAMP) or to a fixed- or varying-length string variable or string column. The assignment must not be to a LONG VARCHAR, CLOB, LONG VARGRAPHIC, 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 a host variable, the following rules apply:

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

  If ISO or JIS formats are used, and if the length of the host variable is less than 8 characters, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.
- **For a TIMESTAMP:** If the host variable is less than 19 characters, an error occurs. If the length is less than 26 characters, but greater than or equal to 19 characters, trailing digits of the microseconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

DATALINK assignments

The assignment of a value to a DATALINK column results in the establishment of a link to a file unless the linkage attributes of the value are empty or the column is defined with NO LINK CONTROL. In cases where a linked value already exists in the column, that file is unlinked. Assigning a null value where a linked value already exists also unlinks the file associated with the old value.

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

- The comment is being changed.
- If the table is placed in Datalink Reconcile Not Possible (DRNP) state, the links in the table can be reinstated by providing linkage attributes identical to the ones in the column.
- If the column is defined with WRITE PERMISSION ADMIN and the file content is changed, a new version of the link can be established by providing a DATALINK value constructed by using the DLURLNEWCOPY function with the same data location.

- If the column is defined with WRITE PERMISSION ADMIN and the file content is changed, but the change needs to be backed out, the existing version of the link can be reinstated by providing a DATALINK value constructed by using the DLURLPREVIOUSCOPY function with the same data location.
- The content of the referenced file is being replaced by another file specified in the DLURLREPLACECONTENT scalar function.

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

- The DLVALUE scalar function can be used to create a new DATALINK value and assign it to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.
- A DATALINK value can be constructed in a CLI parameter using the CLI function SQLBuildDataLink. This value can then be assigned to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.
- The DLURLNEWCOPY scalar function can be used to construct a DATALINK value and assign it to a column. The data location referenced by the constructed DATALINK value must be the same as the one that already exists in the column. The act of assignment by using an UPDATE statement will re-establish the link to the file. A file backup will be taken if the column is defined with RECOVERY YES. This type of assignment is used to notify the database that the file has been updated. The database is thus made aware of the new file and will re-establish a new link to the file.
- The DLURLPREVIOUSCOPY scalar function can be used to construct a DATALINK value and assign it to a column. The data location referenced by the constructed DATALINK value must be the same as the one that already exists in the column. The act of assignment by using an UPDATE statement will reinstate the link. It will also restore the file to the previous version from the archive if the column is defined with RECOVERY YES. This type of assignment is used to back out any change to the file that was made since the previous committed version.
- The DLURLREPLACECONTENT scalar function can be used to create a new DATALINK value and assign it to a column. The act of assignment will not only link the file, but also replace the content with another file specified in the DLURLREPLACECONTENT scalar function.

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

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

  Note that the size of a URL parameter or function result is the same on both input or output, and the size is bound by the length of the DATALINK column. However, in some cases the URL value is returned with an access token attached. In situations where this is possible, the output location must have sufficient storage space for the access token and the length of the DATALINK column. Hence, the actual length of both the comment and the URL (in its fully expanded form) provided on input should be restricted to accommodate the output storage space. If the restricted length is exceeded, this error is raised.
- Input data location does not contain a valid write token (reason code 32).

  The assignment requires a valid write token to be embedded in the data location. This requirement only applies when the column is defined with WRITE

PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE, and the DATALINK value is constructed by the DLURLNEWCOPY or DLURLPREVIOUSCOPY scalar function. On the other hand, a user has the option to provide a write token for a DATALINK column defined with WRITE PERMISSION ADMIN NOT REQUIRING TOKEN FOR UPDATE. However, if the token is not valid, the same error is raised.

This error can also occur when constructing a DATALINK value using the DLURLNEWCOPY or DLURLPREVIOUSCOPY scalar function with value '1' specified in the second argument, but the value does not contain a valid write token.

- The DATALINK value constructed by the DLURLPREVIOUSCOPY scalar function can be assigned only to a DATALINK column defined with WRITE PERMISSION ADMIN and RECOVERY YES (reason code 33).
- The DATALINK value constructed by the DLURLNEWCOPY or DLURLPREVIOUSCOPY scalar function does not match the value that already exists in the column (reason code 34).
- The DATALINK value constructed by the DLURLNEWCOPY or DLURLPREVIOUSCOPY scalar function cannot be used in an INSERT statement to assign a new value (reason code 35).
- The DATALINK value constructed by the DLURLNEWCOPY scalar function cannot be assigned to a DATALINK column defined with WRITE PERMISSION BLOCKED (reason code 39).
- The same DATALINK value constructed by the DLURLNEWCOPY or DLURLPREVIOUSCOPY scalar function cannot be assigned multiple times within the same transaction (reason code 41).
- The DATALINK value constructed by the DLURLREPLACECONTENT scalar function can be assigned to a DATALINK column defined with NO LINK CONTROL, only if the second argument is an empty string or a null value (reason code 42).
- The unlink operation of the replacement file specified in the DLREPLACECONTENT scalar function has not committed (reason code 43).
- The replacement file specified in the DLREPLACECONTENT scalar function is being used in another replacement process (reason code 44).
- The DATALINK-referenced file is being used as the replacement file in another operation (reason code 45).
- The format of the replacement file specified in the DLREPLACECONTENT scalar function is not valid (reason code 46).
- The replacement file value specified in the DLREPLACECONTENT scalar function cannot be a directory or a symbolic link (reason code 47).
- The replacement file specified in the DLREPLACECONTENT scalar function is being linked to a database (reason code 48).
- The replacement file specified in the DLREPLACECONTENT scalar function cannot be found by a Data Links File Manager (reason code 49).
- The DATALINK value constructed by the DLURLNEWCOPY scalar function with a write token contained in the data location value can be assigned only to a DATALINK column with WRITE PERMISSION ADMIN (reason code 50).

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

- File server not currently available (SQLSTATE 57050).
- File does not exist (SQLSTATE 428D1, reason code 24).
- File already linked to another column (SQLSTATE 428D1, reason code 25).

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

- Referenced file cannot be accessed for linking (reason code 26).
- The write token embedded in the data location does not match the write token used to open the file (SQLSTATE 428D1, reason code 36).
- DATALINK value referenced file is in the update-in-progress state (SQLSTATE 428D1, reason code 37).
- The previous archive copy of the DATALINK value referenced file is not available (SQLSTATE 428D1, reason code 40).

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

- File server not currently available (SQLSTATE 57050).
- File with referential integrity control is not in a correct state according to the Data Links File Manager (SQLSTATE 58004).
- DATALINK value referenced file is in the update-in-progress state (SQLSTATE 428D1, reason code 37).

If, when retrieving a DATALINK value for write access (by using the DLURLCOMPLETEWRITE or DLURLPATHWRITE scalar function), the DATALINK column is defined with WRITE PERMISSION ADMIN, the directory access privilege is checked at the file server (DataLink File Manager). The user who issues the query must have the authority to write to the files under the given directory before a write token is generated and embedded in the return DATALINK value. If the user does not have write authority, no write token will be generated, and the SELECT statement will fail (SQLSTATE 42511, reason code 1).

Portions of a DATALINK value can be assigned to host variables following the application of scalar functions such as DLLINKTYPE or DLURLPATH.

Note that usually no attempt is made to access the file server at retrieval time. It is therefore possible that subsequent attempts to access the file server through file system commands might fail. It may be necessary to access the file server to determine the prefix name associated with a path. This can be changed at the file server when the mount point of a file system is moved. First access of a file on a server will cause the required values to be retrieved from the file server and cached at the database server for the subsequent retrieval of DATALINK values for that file server. An error is returned if the file server cannot be accessed (SQLSTATE 57050).

If using the scalar functions DLURLCOMPLETEWRITE or DLURLPATHWRITE to retrieve a DATALINK value, it may be necessary to access the file server to determine the directory access privilege on a path for a user. An error is returned if the file server cannot be accessed (SQLSTATE 57050).

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

User-defined type assignments

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

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

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

If the target of the assignment is a column based on a distinct type, the source data type must be castable to the target data type.

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

> A value of a structured type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the declared type of the host variable is the structured type or a supertype of the structured type.

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

Reference type assignments

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

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

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

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

Numeric comparisons

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

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

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

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

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

String comparisons

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

When comparing character strings of unequal lengths and the collating sequence specified is not of the UCA (Unicode Collation Algorithm) type, the comparison is made using a logical copy of the shorter string, which is padded on the right with single-byte blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings, including those tagged as FOR BIT DATA. If the collating sequence is of the UCA type, the shorter string is not padded on the right, because padding with any character may affect the result of the comparison. Comparisons using UCA are done on one or more weights of each character in the strings, and not on the code points. Details of UCA can be found in the Unicode Technical Standard #10, which is available at the Unicode Consortium Web site. The following information pertains to collating sequences that are not of the UCA type.

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

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

Long strings and LOB strings are not supported in any comparison operations that use the basic comparison operators (=, <>, <, >, <=, and >=). They are supported in comparisons using the LIKE predicate and the POSSTR function.

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

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

then the following is valid:

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

Examples:

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

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

```
’a’ < ’A’ < ’á’ < ’Á’
```

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

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

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

```
 'A' < 'a' < 'Á' < 'á'
```

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

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

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

```
'A' < 'a' < 'Á' < 'á'
```

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

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

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

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

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

Conversion rules for comparison: When two strings are compared, one of the strings is first converted, if necessary, to the 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 108. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

MBCS considerations for string comparisons: Mixed SBCS/MBCS character strings are compared according to the collating sequence specified when the database was created. For databases created with default (SYSTEM) collation sequence, all single-byte ASCII characters are sorted in correct order, but double-byte characters are not necessarily in code point sequence. For databases created with IDENTITY sequence, all double-byte characters are correctly sorted in their code point order, but single-byte ASCII characters are sorted in their code point order as well. For databases created with COMPATIBILITY sequence, a compromise order is used that sorts properly for most double-byte characters, and is almost correct for ASCII. This was the default collation table in DB2 Version 2.

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

Example:

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

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

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

and

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

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

Graphic string comparisons are valid between all graphic string data types except LONG VARGRAPHIC. LONG VARGRAPHIC and DBCLOB data types are not allowed in a comparison operation.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

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

and

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

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

Datetime comparisons

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

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

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

Example:

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

User-defined type comparisons

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

Example:

Given the following YOUTH distinct type and CAMP_DB2_ROSTER table:

```
CREATE DISTINCT TYPE YOUTH AS INTEGER WITH COMPARISONS

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

The following comparison is valid:

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

The following comparison is not valid:

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

However, AGE can be compared to ATTENDEE_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE INTEGER(AGE) > ATTENDEE_NUMBER

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

SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > YOUTH(ATTENDEE_NUMBER)

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

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Reference type comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the function path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

Related reference:

- "Identifiers" on page 58
- "LIKE predicate" on page 216
- "POSSTR" on page 384
- "Datetime values" on page 85
- "Casting between data types" on page 96
- "Rules for result data types" on page 113
- "Rules for string conversions" on page 117

Rules for result data types

The data types of a result are determined by rules which are applied to the operands in an operation. This section explains those rules.

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

These rules are applied subject to other restrictions on long strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 characters. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

Character strings

Character strings are compatible with other character strings. Character strings include data types CHAR, VARCHAR, LONG VARCHAR, and CLOB.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
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)
LONG VARCHAR	CHAR(y), VARCHAR(y), or LONG VARCHAR	LONG VARCHAR
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where z = max(x,y)
CLOB(x)	LONG VARCHAR	CLOB(z) where z = max(x,32700)

The code page of the result character string will be derived based on the rules for string conversions.

Graphic strings

Graphic strings are compatible with other graphic strings. Graphic strings include data types GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
LONG VARGRAPHIC	GRAPHIC(y), VARGRAPHIC(y), or LONG VARGRAPHIC	LONG VARGRAPHIC
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max (x,y)
DBCLOB(x)	LONG VARGRAPHIC	DBCLOB(z) where z = max (x,16350)

The code page of the result graphic string will be derived based on the rules for string conversions.

Character and graphic strings in a Unicode database

In a Unicode database, character strings and graphic strings are compatible.

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)
LONG VARGRAPHIC	CHAR(y) or VARCHAR(y) or LONG VARCHAR	LONG VARGRAPHIC
LONG VARCHAR	GRAPHIC(y) or VARGRAPHIC(y)	LONG VARGRAPHIC
DBCLOB(x)	CHAR(y) or VARCHAR(y) or CLOB(y)	DBCLOB(z) where z = max(x,y)
DBCLOB(x)	LONG VARCHAR	DBCLOB(z) where z = max(x,16350)
CLOB(x)	GRAPHIC(y) or VARGRAPHIC(y)	DBCLOB(z) where z = max(x,y)
CLOB(x)	LONG VARGRAPHIC	DBCLOB(z) where z = max(x,16350)

Binary large object (BLOB)

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

Numeric

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

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
BIGINT	BIGINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	SMALLINT	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = x+max(w-x,5)[1]
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where p = x+max(w-x,11)[1]
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where p = x+max(w-x,19)[1]
DECIMAL(w,x)	DECIMAL(y,z)	DECIMAL(p,s) where p = max(x,z)+max(w-x,y-z)[1]s = max(x,z)
REAL	REAL	REAL
REAL	DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric	DOUBLE

[1] Precision cannot exceed 31.

DATE

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

TIME

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

TIMESTAMP

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

DATALINK

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

User-defined types

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

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

Related reference:

- "BLOB" on page 286
- "Rules for string conversions" on page 117

Rules for string conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This section explains those rules.

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

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to ’W’ in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:
- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- 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 (or VALUE) scalar function
- The selected result expression of the CASE expression
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:
- The code pages are different
- Neither string is BIT DATA

- The string is neither null nor empty

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.

Partition-compatible data types

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

Table 10 shows the compatibility of data types in partitions.

Partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with CHAR.
- Partition compatibility is not affected by columns with NOT NULL or FOR BIT DATA definitions.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base data type of the UDT is used to analyze partition compatibility.
- Decimals of the same value in the partitioning key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.
- CHAR or VARCHAR of different lengths are compatible data types.
- REAL or DOUBLE values that are equal are treated identically even though their precision differs.

Table 10. Partition Compatibilities

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

Table 10. Partition Compatibilities (continued)

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	Distinct Type	Structured Type

Note:

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

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

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

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the 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. This means that a user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a user-defined type and a constant is only possible if the user-defined type has been cast to the constant's built-in type, or if the constant has been cast to the user-defined type. For example, using the table and distinct type in "User-defined type comparisons" on page 111, the following comparisons with the constant 14 are valid:

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

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

The following comparison is not valid:

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

Integer constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2 147 483 647, and not -2 147 483 648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9 223 372 036 854 775 807, and not -9 223 372 036 854 775 808, which is the limit for big integer values.

Examples:

```
64     -15     +100     32767     720176      12345678901
```

In syntax diagrams, the term 'integer' is used for a large integer constant that must not include a sign.

Floating-point constants

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double-precision. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of characters in the constant must not exceed 30.

Examples:

```
15E1     2.E5     2.2E-1     +5.E+2
```

Decimal constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be 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
```

Character string constants

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

Examples:

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

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.

Hexadecimal constants

| A *hexadecimal constant* specifies a varying-length character string in the section code
| page.

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

Examples:

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

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

Graphic string constants

A *graphic string constant* specifies a varying-length graphic string consisting of a sequence of double-byte characters that starts and ends with a single-byte apostrophe ('), and that is preceded by a single-byte G or N. The characters between the apostrophes must represent an even number of bytes, and the length of the graphic string must not exceed 16 336 bytes.

Examples:

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

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

Related reference:

- "Expressions" on page 168
- "Assignments and comparisons" on page 99

Special registers

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:

```
|
►►──┬─┬─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 DEFAULT TRANSFORM GROUP─────────────────────┤
    ├─CURRENT DEGREE──────────────────────────────────────┤
    ├─CURRENT EXPLAIN MODE────────────────────────────────┤
    ├─CURRENT EXPLAIN SNAPSHOT────────────────────────────┤
    ├─CURRENT ISOLATION───────────────────────────────────┤
    ├─CURRENT LOCK TIMEOUT────────────────────────────────┤
    ├─CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION─────┤
    ├─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 TIME──────┬───────────────────────────────┤
    │ │             (1)   │                               │
    │ └─CURRENT_TIME──────┘                               │
    ├─┬─CURRENT TIMESTAMP──────┬──────────────────────────┤
    │ │                  (1)   │                          │
    │ └─CURRENT_TIMESTAMP──────┘                          │
    ├─┬─CURRENT TIMEZONE──────┬───────────────────────────┤
    │ │                 (1)   │                           │
    │ └─CURRENT_TIMEZONE──────┘                           │
    ├─┬─CURRENT USER──────┬───────────────────────────────┤
    │ │             (1)   │                               │
    │ └─CURRENT_USER──────┘                               │
    ├─┬─SESSION_USER─┬────────────────────────────────────┤
    │ └─USER─────────┘                                    │
    └─SYSTEM_USER─────────────────────────────────────────┘
```

Notes:

1 The SQL 1999 Core standard uses the form with the underscore.

Some special registers can be updated using the SET statement. The following table shows which of the special registers can be updated.

Table 11. Special Registers

Special Register	Updatable
CURRENT CLIENT_ACCTNG	No
CURRENT CLIENT_APPLNAME	No
CURRENT CLIENT_USERID	No
CURRENT CLIENT_WRKSTNNAME	No
CURRENT DATE	No
CURRENT DBPARTITIONNUM	No
CURRENT DEFAULT TRANSFORM GROUP	Yes
CURRENT DEGREE	Yes
CURRENT EXPLAIN MODE	Yes
CURRENT EXPLAIN SNAPSHOT	Yes
CURRENT ISOLATION	Yes
CURRENT LOCK TIMEOUT	Yes
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	Yes
CURRENT PACKAGE PATH	Yes
CURRENT PATH	Yes
CURRENT QUERY OPTIMIZATION	Yes
CURRENT REFRESH AGE	Yes
CURRENT SCHEMA	Yes
CURRENT SERVER	No
CURRENT TIME	No
CURRENT TIMESTAMP	No
CURRENT TIMEZONE	No
CURRENT USER	No
SESSION_USER	Yes
SYSTEM_USER	No
USER	Yes

When a special register is referenced in a routine, the value of the special register in the routine depends on whether the special register is updatable or not. For non-updatable special registers, the value is set to the default value for the special register. For updatable special registers, the initial value is inherited from the invoker of the routine and can be changed with a subsequent SET statement inside the routine.

CURRENT CLIENT_ACCTNG

The CURRENT CLIENT_ACCTNG (or CLIENT ACCTNG) special register contains the value of the accounting string from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the accounting string can be changed by using the Set Client Information (**sqleseti**) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the current value of the accounting string for this connection.

```
VALUES (CURRENT CLIENT_ACCTNG)
  INTO :ACCT_STRING
```

CURRENT CLIENT_APPLNAME

The CURRENT CLIENT_APPLNAME (or CLIENT APPLNAME) special register contains the value of the application name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the application name can be changed by using the Set Client Information (**sqleseti**) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Select which departments are allowed to use the application being used in this connection.

```
SELECT DEPT
  FROM DEPT_APPL_MAP
  WHERE APPL_NAME = CURRENT CLIENT_APPLNAME
```

CURRENT CLIENT_USERID

The CURRENT CLIENT_USERID (or CLIENT USERID) special register contains the value of the client user ID from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the client user ID can be changed by using the Set Client Information (**sqleseti**) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Find out in which department the current client user ID works.

```
SELECT DEPT
  FROM DEPT_USERID_MAP
  WHERE USER_ID = CURRENT CLIENT_USERID
```

CURRENT CLIENT_WRKSTNNAME

The CURRENT CLIENT_WRKSTNNAME (or CLIENT WRKSTNNAME) special register contains the value of the workstation name from the client information specified for this connection. The data type of the register is VARCHAR(255). The default value of this register is an empty string.

The value of the workstation name can be changed by using the Set Client Information (**sqleseti**) API.

Note that the value provided via the sqleseti API is in the application code page, and the special register value is stored in the database code page. Depending on the data values used when setting the client information, truncation of the data value stored in the special register may occur during code page conversion.

Example: Get the workstation name being used for this connection.

```
VALUES (CURRENT CLIENT_WRKSTNNAME)
  INTO :WS_NAME
```

CURRENT DATE

The CURRENT DATE (or CURRENT_DATE) special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT DATE is not inherited from the invoking statement.

In a federated system, CURRENT DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT DATE register at the federated server, not from the data sources.

Example: Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
  SET PRENDATE = CURRENT DATE
  WHERE PROJNO = 'MA2111'
```

CURRENT DBPARTITIONNUM

The CURRENT DBPARTITIONNUM special register specifies an INTEGER value that identifies the coordinator node number for the statement. For statements issued from an application, the coordinator is the partition to which the application connects. For statements issued from a routine, the coordinator is the partition 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 partitioning. (In other words, if there is no db2nodes.cfg file. For partitioned databases, the db2nodes.cfg file exists and contains partition, or node, definitions.)

CURRENT DBPARTITIONNUM can be changed through the CONNECT statement, but only under certain conditions.

For compatibility with versions earlier than Version 8, the keyword NODE can be substituted for DBPARTITIONNUM.

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

```
VALUES CURRENT DBPARTITIONNUM
  INTO :APPL_NODE
```

Related reference:

- “CONNECT (Type 1) statement” in the *SQL Reference, Volume 2*

CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR(18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs. This special register does not specify the transform groups used in static SQL statements, or in the exchange of parameters and results with external functions or methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a length of zero).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using the SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Examples:

Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

```
SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

Retrieve the name of the default transform group assigned to this special register.

```
VALUES (CURRENT DEFAULT TRANSFORM GROUP)
```

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition parallelism for the execution of dynamic SQL statements. (For static SQL, the DEGREE bind option provides the same control.) The data type of the register is CHAR(5). Valid values are ANY or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32 767 when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is ANY when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:
- The value of the maximum query degree (*max_querydegree*) configuration parameter
- The application runtime degree
- The SQL statement compilation degree.

If the *intra_parallel* database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intra-partition parallelism.

The value can be changed by invoking the SET CURRENT DEGREE statement.

The initial value of CURRENT DEGREE is determined by the *dft_degree* database configuration parameter.

Related reference:
- "SET CURRENT DEGREE statement" in the *SQL Reference, Volume 2*

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables. This information does not include the Explain snapshot. Possible values are YES, EXPLAIN, NO, REOPT, RECOMMEND INDEXES, and EVALUATE INDEXES. (For static SQL, the EXPLAIN bind option provides the same control. In the case of the PREP and BIND commands, the EXPLAIN option values are: YES, NO, and ALL.)

YES Enables the Explain facility and causes Explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN
Enables the facility, but dynamic statements are not executed.

NO Disables the Explain facility.

REOPT
Enables the Explain facility and causes Explain information for a dynamic (or incremental-bind) SQL statement to be captured only when the statement is reoptimized using real values for the input variables (host variables, special registers, or parameter markers).

RECOMMEND INDEXES
Recommends a set of indexes for each dynamic query. Populates the ADVISE_INDEX table with the set of indexes.

EVALUATE INDEXES
Explains dynamic queries as though the recommended indexes existed. The indexes are picked up from the ADVISE_INDEX table.

The initial value is NO. The value can be changed by invoking the SET CURRENT EXPLAIN MODE statement.

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked. The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option. RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
  INTO :EXPL_MODE
```

Related reference:

- “SET CURRENT EXPLAIN MODE statement” in the *SQL Reference, Volume 2*
- Appendix J, “Explain register values,” on page 735

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: DELETE, INSERT, SELECT, SELECT INTO, 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, 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
```

Related reference:

- "SET CURRENT EXPLAIN SNAPSHOT statement" in the *SQL Reference, Volume 2*
- Appendix J, "Explain register values," on page 735

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
```

Related concepts:

- "Isolation levels" on page 13

Related reference:

- "SET CURRENT ISOLATION statement" in the *SQL Reference, Volume 2*

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, and MDC block locks. The data type of the register is INTEGER.

Valid values for the CURRENT LOCK TIMEOUT special register are integers between -1 and 32767, inclusive. This special register can also be set to the null value. A value of -1 specifies that timeouts are not to take place, and that the application is to wait until the lock is released or a deadlock is detected. A value of 0 specifies that the application is not to wait for a lock; if a lock cannot be obtained, an error is to be returned immediately.

The value of the CURRENT LOCK TIMEOUT special register can be changed by invoking the SET CURRENT LOCK TIMEOUT statement. Its initial value is null; in this case, the current value of the *locktimeout* database configuration parameter is used when waiting for a lock, and this value will be returned for the special register.

Related reference:

- "SET CURRENT LOCK TIMEOUT statement" in the *SQL Reference, Volume 2*

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.

Related reference:

- "SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION statement" in the *SQL Reference, Volume 2*
- "dft_mttb_types - Default maintained table types for optimization configuration parameter" in the *Administration Guide: Performance*

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
```

Related reference:

- "CURRENT PATH" on page 140
- "SET CURRENT PACKAGE PATH statement" in the *SQL Reference, Volume 2*

CURRENT PATH

The CURRENT PATH (or CURRENT_PATH) special register specifies a VARCHAR(254) value that identifies the SQL path to be used when resolving function references and data type references in dynamically prepared SQL statements. CURRENT FUNCTION PATH is a synonym for CURRENT PATH. CURRENT PATH is also used to resolve stored procedure references in CALL statements. The initial value is the default value specified below. For static SQL, the FUNCPATH bind option provides an SQL path that is used for function and data type resolution.

The CURRENT PATH special register contains a list of one or more schema names that are enclosed by double quotation marks and separated by commas. For example, an SQL path specifying that the database manager is to look first in the FERMAT schema, then in the XGRAPHIC schema, and finally in the SYSIBM schema, is returned in the CURRENT PATH special register as:

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

The default value is “SYSIBM”,“SYSFUN”,“SYSPROC”,*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 254 characters 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 DISTINCT TYPE, CREATE FUNCTION, COMMENT, and DROP.

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

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

Related reference:

- “Functions” on page 152
- “SET PATH statement” in the *SQL Reference, Volume 2*

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
```

Related reference:

- "SET CURRENT QUERY OPTIMIZATION statement" in the *SQL Reference, Volume 2*

CURRENT REFRESH AGE

The CURRENT REFRESH AGE special register specifies a timestamp duration value with a data type of DECIMAL(20,6). It is the maximum duration since a particular timestamped event occurred to a cached data object (for example, a REFRESH TABLE statement processed on a system-maintained REFRESH DEFERRED materialized query table), such that the cached data object can be used to optimize the processing of a query. If CURRENT REFRESH AGE has a value of 99 999 999 999 999 (ANY), and the query optimization class is 5 or more, the types of tables specified in CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION are considered when optimizing the processing of a dynamic SQL query.

The initial value of CURRENT REFRESH AGE is zero. The value can be changed by invoking the SET CURRENT REFRESH AGE statement.

Related reference:

- "SET CURRENT REFRESH AGE statement" in the *SQL Reference, Volume 2*

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 OS/390, CURRENT SQLID (or CURRENT_SQLID) is a synonym for CURRENT SCHEMA.

The initial value of CURRENT SCHEMA is the authorization ID of the current session user. The value can be changed by invoking the SET SCHEMA statement.

The QUALIFIER bind option controls the schema name used to qualify database object references, where applicable, for static SQL statements.

Example: Set the schema for object qualification to 'D123'.

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

CURRENT SERVER

The CURRENT SERVER (or CURRENT_SERVER) special register specifies a VARCHAR(18) value that identifies the current application server. The register contains the actual name of the application server, not an alias.

CURRENT SERVER can be changed through the CONNECT statement, but only under certain conditions.

When used in an SQL statement inside a routine, CURRENT SERVER is not inherited from the invoking statement.

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

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

Related reference:

- "CONNECT (Type 1) statement" in the *SQL Reference, Volume 2*

CURRENT TIME

The CURRENT TIME (or CURRENT_TIME) special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

When used in an SQL statement inside a routine, CURRENT TIME is not inherited from the invoking statement.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Example: 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.

When used in an SQL statement inside a routine, CURRENT TIMESTAMP is not inherited from the invoking statement.

In a federated system, CURRENT TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT TIMESTAMP register at the federated server, not from the data sources.

Example: Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
  VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

CURRENT TIMEZONE

The CURRENT TIMEZONE (or CURRENT_TIMEZONE) special register specifies the difference between UTC (Coordinated Universal Time, formerly known as GMT) and local time at the application server. The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. (The CURRENT TIMEZONE value is determined from C runtime functions.)

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used; for example, in time and timestamp arithmetic.

When used in an SQL statement inside a routine, CURRENT TIMEZONE is not inherited from the invoking statement.

Example: Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```
INSERT INTO IN_TRAY VALUES (
  CURRENT TIMESTAMP - CURRENT TIMEZONE,
  :source,
  :subject,
  :notetext )
```

CURRENT USER

The CURRENT USER (or CURRENT_USER) special register specifies the authorization ID that is to be used for statement authorization. For static SQL statements, the value represents the authorization ID that is used when the package is bound. For dynamic SQL statements, the value is the same as the value of the SESSION_USER special register for packages bound with the DYNAMICRULES(RUN) bind option. The data type of the register is VARCHAR(128).

Example: Select table names whose schema matches the value of the CURRENT USER special register.

```
SELECT TABNAME FROM SYSCAT.TABLES
  WHERE TABSCHEMA = CURRENT USER AND TYPE = 'T'
```

If this statement is executed as a static SQL statement, it returns the tables whose schema name matches the binder of the package that includes the statement. If this statement is executed as a dynamic SQL statement, it returns the tables whose schema name matches the current value of the SESSION_USER special register.

SESSION_USER

The SESSION_USER special register specifies the authorization ID that is to be used for the current session. The value of this register is used for authorization checking of dynamic SQL statements when DYNAMICRULES run behavior is in effect for the package. The data type of the register is VARCHAR(128).

The initial value of SESSION_USER for a new connection is the same as the value of the SYSTEM_USER special register. Its value can be changed by invoking the SET SESSION AUTHORIZATION statement.

SESSION_USER is a synonym for the USER special register.

Example: Determine what routines can be executed using dynamic SQL. Assume DYNAMICRULES run behavior is in effect for the package that will issue the dynamic SQL statement that invokes the routine.

```
SELECT SCHEMA, SPECIFICNAME FROM SYSCAT.ROUTINEAUTH
  WHERE GRANTEE = SESSION_USER
  AND EXECUTEAUTH IN ('Y', 'G')
```

Related reference:

- “SET SESSION AUTHORIZATION statement” in the *SQL Reference, Volume 2*

SYSTEM_USER

The SYSTEM_USER special register specifies the authorization ID of the user that connected to the database. The value of this register can only be changed by connecting as a user with a different authorization ID. The data type of the register is VARCHAR(128).

See "Example" in the description of the SET SESSION AUTHORIZATION statement.

Related reference:

- "SET SESSION AUTHORIZATION statement" in the *SQL Reference, Volume 2*

USER

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

When used in an SQL statement inside a routine, USER is not inherited from the invoking statement.

Example: Select all notes from the `IN_TRAY` table that were placed there by the user.

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

Functions

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

- *Built-in functions* are provided with the database manager. They return a single result value and are identified as part of the SYSIBM schema. Such functions include column functions (for example, AVG), operator functions (for example, +), and casting functions (for example, DECIMAL).
- *User-defined functions* are functions that are registered to a database in SYSCAT.ROUTINES (using the CREATE FUNCTION statement). User-defined functions are never part of the SYSIBM schema. One set of such functions is provided with the database manager in a schema called SYSFUN.

  User-defined functions extend the capabilities of the database system by adding function definitions (provided by users or third party vendors) that can be applied in the database engine itself. Extending database functions lets the database exploit the same functions in the engine that an application uses, providing more synergy between application and database.

External, SQL, and sourced user-defined functions

A user-defined function can be an external function, an SQL function, or a sourced function. An *external function* is defined to the database with a reference to an object code library, and a function within that library that will be executed when the function is invoked. External functions cannot be column functions. An *SQL function* is defined to the database using only the SQL RETURN statement. It can return either a scalar value, a row, or a table. SQL functions cannot be column functions. A *sourced function* is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or column functions. They are useful for supporting existing functions with user-defined types.

Scalar, column, row, and table user-defined functions

Each user-defined function is also categorized as a scalar, column, or table function. A *scalar function* is a function that returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

A *column function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. These are also sometimes called *aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF, which is sourced upon one of the built-in column functions, can be defined. This is useful for distinct types. For example, if there is a distinct type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE), which is sourced on the built-in function AVG(INTEGER), could be defined, and it would be a column function.

A *row function* is a function that returns one row of values. It may only be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.

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

Function signatures

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

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

Access to functions is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific function or a set of functions. The EXECUTE privilege (or DBADM authority) is needed to invoke a function. The definer of the function automatically receives the EXECUTE privilege. The definer of an external function or an SQL function having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the function. The definer (or SYSADM or DBADM) must then grant it to the user who wants to invoke the function from any SQL statement, reference the function in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint), or create another function sourced on this function. If the EXECUTE privilege is not granted to a user, the function will not be considered by the function resolution algorithm, even if it is a better match. Built-in functions (SYSIBM functions) and SYSFUN functions have the EXECUTE privilege implicitly granted to PUBLIC.

Function resolution

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

An *argument* is a value passed to a function upon invocation. When a function is invoked in SQL, it is passed a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a function. When a function is defined to the database, either internally (a built-in function) or by a user (a user-defined function), its parameters (zero or more) are specified, and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a function. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the function given in the invocation, EXECUTE privilege on the function, the number and data types of the arguments, all the functions with the same name in the SQL path, and the data types of their corresponding parameters as the basis for deciding whether or not to select a function. The following are the possible outcomes of the decision process:

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

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

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

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

  then, the second RISK will be chosen.

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

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

  would choose the first RISK, because SMALLINT can be promoted to INTEGER and is a better match than DOUBLE which is further down the precedence list.

  When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.
- No function is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

  ```
  SELECT ... RISK(C) ...
  ```

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

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

  and an SQL path of:

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

  the following function reference:

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

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

Determining the best fit

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

Function resolution is performed using the following steps:

1. First, find all functions from the catalog (SYSCAT.ROUTINES), and built-in functions, such that all of the following are true:
   - For invocations where the schema name was specified (a qualified reference), the schema name and the function name match the invocation name.
   - For invocations where the schema name was not specified (an unqualified reference), the function name matches the invocation name and has a schema name that matches one of the schemas in the SQL path.
   - The invoker has the EXECUTE privilege on the function.
   - The number of defined parameters matches the invocation.
   - Each invocation argument matches the function’s corresponding defined parameter in data type, or is “promotable” to it.
2. Next, consider each argument of the function invocation, from left to right. For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list corresponding to the argument data type for which there exists a function with a parameter of that data type. Lengths, precisions, scales and the FOR BIT DATA attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

   The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).
3. If more than one candidate function remains after Step 2, all remaining candidate functions must have identical signatures but be in different schemas. Choose the function whose schema is earliest in the user’s SQL path.
4. If there are no candidate functions remaining after step 2, an error is returned (SQLSTATE 42884).

Function path considerations for built-in functions

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN and SYSPROC schemas, but are not considered built-in functions because they are developed as user-defined functions and have no special processing considerations. Users cannot define additional functions in the SYSIBM, SYSFUN, or SYSPROC schemas (or in any other schema whose name begins with the letters ’SYS’).

As already stated, the built-in functions participate in the function resolution process exactly as do the user-defined functions. One difference between built-in and user-defined functions, from a function resolution perspective, is that the built-in functions must always be considered during function resolution. Therefore, omission of SYSIBM from the path results in the assumption (for function and data type resolution) that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN","SYSIBM","SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN","SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH, because SYSIBM implicitly appears first in the path.

To minimize potential problems in this area:

- Never use the names of built-in functions for user-defined functions.
- If, for some reason, it is necessary to create a user-defined function with the same name as a built-in function, be sure to qualify any references to it.

Example of function resolution

Following is an example of successful function resolution. (Note that not all required keywords are shown.)

There are seven ACT functions, in three different schemas, registered as:

```
CREATE FUNCTION AUGUSTUS.ACT (CHAR(5), INT, DOUBLE) SPECIFIC ACT_1 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_2 ...
CREATE FUNCTION AUGUSTUS.ACT (INT, INT, DOUBLE, INT) SPECIFIC ACT_3 ...
CREATE FUNCTION JULIUS.ACT (INT, DOUBLE, DOUBLE) SPECIFIC ACT_4 ...
CREATE FUNCTION JULIUS.ACT (INT, INT, DOUBLE) SPECIFIC ACT_5 ...
CREATE FUNCTION JULIUS.ACT (SMALLINT, INT, DOUBLE) SPECIFIC ACT_6 ...
CREATE FUNCTION NERO.ACT (INT, INT, DEC(7,2)) SPECIFIC ACT_7 ...
```

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_7 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, and ACT_5 are an exact match with the argument type. No functions can be eliminated from consideration; therefore the next argument must be examined.
- For this second argument, ACT_2 and ACT_5 are exact matches, but ACT_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation between ACT_2 and ACT_5.

- For the third and last argument, neither ACT_2 nor ACT_5 match the argument type exactly, but both are equally good.
- There are two functions remaining, ACT_2 and ACT_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis, ACT_5 is the function chosen.

Function invocation

Once the function is selected, there are still possible reasons why the use of the function may not be permitted. Each function is defined to return a result with a specific data type.If this result data type is not compatible with the context in which the function is invoked, an error will occur. For example, given functions named STEP defined, this time, with different data types as the result:

```
STEP(SMALLINT) returns CHAR(5)
STEP(DOUBLE) returns INTEGER
```

and the following function reference (where S is a SMALLINT column):

```
SELECT ... 3 + STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen.An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

A couple of other examples where this can happen are as follows, both of which will result in an error on the statement:

- The function was referenced in a FROM clause, but the function selected by the function resolution step was a scalar or column function.
- The reverse case, where the context calls for a scalar or column function, and function resolution selects a table function.

In cases where the arguments of the function invocation were not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns. This includes the case where precision, scale, or length differs between the argument and the parameter.

Conservative binding semantics

There are instances in which routines and data types are resolved when a statement is processed, and the database manager must be able to repeat this resolution. This is true in:

- Static DML statements in packages
- Views
- Triggers
- Check constraints
- SQL routines

For static DML statements in packages, the routine and data type references are resolved during a bind operation. Routine and data type references in views, triggers, SQL routines, and check constraints are resolved when the database object is created.

If routine resolution is performed again on any routine references in these objects, it could change the behavior if:

- A new routine has been added with a signature that is a better match, but the actual executable performs different operations.
- The definer has been granted the execute privilege on a routine with a signature that is a better match, but the actual executable performs different operations.

Similarly, if resolution is performed again on any data type in these objects, it could change the behavior if a new data type has been added with the same name in a different schema that is also on the SQL path. To avoid this, the database manager applies *conservative binding semantics* wherever necessary. This ensures that routine and data type references will be resolved using the same SQL path and the set of routines to which it previously resolved when it was bound. The creation timestamp of routines and data types considered during resolution is not later than the time when the statement was bound. (Built-in functions added starting with Version 6.1 have a creation timestamp that is based on the time of database creation or migration.) In this way, only the routines and data types that were considered during routine and data type resolution when the statement was originally processed will be considered. Hence, newly created routines, newly authorized routines, and data types are not considered when conservative binding semantics are applied.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). Assume the SQL path contains both schemas SCHEMA1 and SCHEMA2 (although their order within the SQL path does not matter). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that calls BAR(INT_VAL). This will resolve to the function SCHEMA2.BAR(DOUBLE). The view will always use SCHEMA2.BAR(DOUBLE), even if someone grants USER1 the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created.

For static DML in packages, the packages can rebind implicitly, or by explicitly issuing the REBIND command (or corresponding API), or the BIND command (or corresponding API). The implicit rebind is always performed to resolve routines and data types with conservative binding semantics. The REBIND command provides the option to resolve with conservative binding semantics (RESOLVE CONSERVATIVE) or to resolve by considering any new routines and data types (RESOLVE ANY, the default option).

Implicit rebind of a package always resolves the same routine. Even if EXECUTE privilege on a better-matched routine was granted, that routine will not be considered. Explicit rebind of a package can result in a different routine being selected. (But if RESOLVE CONSERVATIVE is specified, routine resolution will follow conservative binding semantics).

If a routine is specified during the creation of a view, trigger, constraint, or SQL routine body, the specific instance of the routine to be used is determined by routine resolution at the time the object is created. Subsequent granting of the EXECUTE privilege after the object has been created will not change the specific routine that the object uses.

Consider a database with two functions that have the signatures SCHEMA1.BAR(INTEGER) and SCHEMA2.BAR(DOUBLE). USER1 has been granted the EXECUTE privilege on the function SCHEMA2.BAR(DOUBLE). Suppose USER1 creates a view that calls BAR(INT_VAL). This will resolve to the function SCHEMA2.BAR(DOUBLE). The view will always use

SCHEMA2.BAR(DOUBLE), even if someone grants USER1 the EXECUTE privilege on SCHEMA1.BAR(INTEGER) after the view has been created.

The same behavior occurs in other database objects. For example, if a package is implicitly rebound (perhaps after dropping an index), the package will refer to the same specific routine both before and after the implicit rebind. An *explicit* rebind of a package, however, can result in a different routine being selected.

Related reference:

- "CURRENT PATH" on page 140
- "Promotion of data types" on page 94
- "Assignments and comparisons" on page 99

Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or *subject argument*) has the same type, or is a subtype of the subject type (also called the *subject parameter*), of the method. For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR, and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. *Observer methods* are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. *Mutator methods* are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described above is an example of a mutator method for the type ADDRESS.

Explicitly defined methods, or *user-defined methods*, are methods that are registered to a database in SYSCAT.ROUTINES, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

User-defined methods for structured types extend the function of the database system by adding method definitions (provided by users or third party vendors) that can be applied to structured type instances in the database engine. Defining database methods lets the database exploit the same methods in the engine that an application uses, providing more synergy between application and database.

External and SQL user-defined methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, because they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as *type preserving* (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method signatures

A method is identified by its subject type, a method name, the number of parameters, and the data types of its parameters. This is called a *method signature*, and it must be unique within the database.

There can be more than one method with the same name for a structured type, provided that:

- The number of parameters or the data types of the parameters are different, or

- The methods are part of the same method hierarchy (that is, the methods are in an overriding relationship or override the same original method), or
- The same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name that has multiple method instances is called an *overloaded method*. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy. These methods must have different parameter types.

A method can be invoked by referring (in an allowable context) to the method name, preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. A list of arguments enclosed in parentheses must follow. Which method is actually invoked depends on the static type of the subject type, using the method resolution process described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution apply.

If function resolution results in a method defined WITH FUNCTION ACCESS, all subsequent steps of method invocation are processed.

Access to methods is controlled through the EXECUTE privilege. GRANT and REVOKE statements are used to specify who can or cannot execute a specific method or a set of methods. The EXECUTE privilege (or DBADM authority) is needed to invoke a method. The definer of the method automatically receives the EXECUTE privilege. The definer of an external method or an SQL method having the WITH GRANT option on all underlying objects also receives the WITH GRANT option with the EXECUTE privilege on the method. The definer (or SYSADM or DBADM) must then grant it to the user who wants to invoke the method from any SQL statement, or reference the method in any DDL statement (such as CREATE VIEW, CREATE TRIGGER, or when defining a constraint). If the EXECUTE privilege is not granted to a user, the method will not be considered by the method resolution algorithm, even if it is a better match.

Method resolution

After method invocation, the database manager must decide which of the possible methods with the same name is the "best fit". Functions (built-in or user-defined) are not considered during method resolution.

An *argument* is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a method. When a method is defined to the database, either implicitly (system-generated for a type) or by a user (a user-defined method), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input to a method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

Method resolution

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.

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, 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 161), or a method has been specified in a DDL statement, this method is known as the "most specific applicable authorized method". If the subject is of a structured type, that method could have one or more overriding methods. DB2 must then determine which of these methods to invoke, based on the dynamic type (most specific type) of the subject at run time. This determination is called "determining the most specific dispatchable method". That process is described here.

1. Find the original method in the method hierarchy that the most specific applicable authorized method is part of. This is called the *root method*.
2. Create the set of dispatchable methods, which includes the following:
   - The most specific applicable authorized method.
   - Any method that overrides the most specific applicable authorized method, and which is defined for a type that is a subtype of the subject of this invocation.
3. Determine the most specific dispatchable method, as follows:
   a. Start with an arbitrary method that is an element of the set of dispatchable methods and that is a method of the dynamic type of the subject, or of one of its supertypes. This is the initial most specific dispatchable method.
   b. Iterate through the elements of the set of dispatchable methods. For each method: If the method is defined for one of the proper subtypes of the type for which the most specific dispatchable method is defined, and if it is defined for one of the supertypes of the most specific type of the subject, then repeat step 2 with this method as the most specific dispatchable method; otherwise, continue iterating.
4. Invoke the most specific dispatchable method.

Example:

Given are three types, "Person", "Employee", and "Manager". There is an original method "income", defined for "Person", which computes a person's income. A person is by default unemployed (a child, a retiree, and so on). Therefore, "income" for type "Person" always returns zero. For type "Employee" and for type "Manager", different algorithms have to be applied to calculate the income. Hence, the method "income" for type "Person" is overridden in "Employee" and "Manager".

Create and populate a table as follows:

```
CREATE TABLE aTable (id integer, personColumn Person);
  INSERT INTO aTable VALUES (0, Person()), (1, Employee()), (2, Manager());
```

List all persons who have a minimum income of $40000:

```
SELECT id, person, name
  FROM aTable
  WHERE person..income() >= 40000;
```

The method ″income″ for type ″Person″ is chosen, using method resolution, to be the most specific applicable authorized method.

1. The root method is ″income″ for ″Person″ itself.
2. The second step of the algorithm above is carried out to construct the set of dispatchable methods:
   - The method ″income″ for type ″Person″ is included, because it is the most specific applicable authorized method.
   - The method ″income″ for type ″Employee″, and ″income″ for ″Manager″ is included, because both methods override the root method, and both ″Employee″ and ″Manager″ are subtypes of ″Person″.

   Therefore, the set of dispatchable methods is: {″income″ for ″Person″, ″income″ for ″Employee″, ″income″ for ″Manager″}.
3. Determine the most specific dispatchable method:
   - For a subject whose most specific type is ″Person″:
     a. Let the initial most specific dispatchable method be ″income″ for type ″Person″.
     b. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of ″Person″ and for a supertype of the most specific type of the subject, ″income″ for type ″Person″ is the most specific dispatchable method.
   - For a subject whose most specific type is ″Employee″:
     a. Let the initial most specific dispatchable method be ″income″ for type ″Person″.
     b. Iterate through the set of dispatchable methods. Because method ″income″ for type ″Employee″ is defined for a proper subtype of ″Person″ and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method ″income″ for type ″Employee″ is a better match for the most specific dispatchable method. Repeat this step with method ″income″ for type ″Employee″ as the most specific dispatchable method.
     c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of ″Employee″ and for a supertype of the most specific type of the subject, method ″income″ for type ″Employee″ is the most specific dispatchable method.
   - For a subject whose most specific type is ″Manager″:
     a. Let the initial most specific dispatchable method be ″income″ for type ″Person″.
     b. Iterate through the set of dispatchable methods. Because method ″income″ for type ″Manager″ is defined for a proper subtype of ″Person″ and for a supertype of the most specific type of the subject (Note: A type is its own super- and subtype.), method ″income″ for type ″Manager″ is a better match for the most specific dispatchable method. Repeat this step with method ″income″ for type ″Manager″ as the most specific dispatchable method.
     c. Because there is no other method in the set of dispatchable methods that is defined for a proper subtype of ″Manager″ and for a supertype of the most specific type of the subject, method ″income″ for type ″Manager″ is the most specific dispatchable method.

4. Invoke the most specific dispatchable method.

Related reference:
- “Promotion of data types” on page 94
- “Assignments and comparisons” on page 99

Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. When repeatedly using similar complex expressions, an SQL function to encapsulate a common expression can be considered.

In a Unicode database, an expression that accepts a character or graphic string will accept any string types for which conversion is supported.

expression:

```
     ┌─operator──────────────────────────────┐
     ↓                                       │
├──────┬───┬──┬─function───────────────────┬─┴──────────────────────────────────┤
       ├ + ┤  ├─(expression)───────────────┤
       └ - ┘  ├─constant───────────────────┤
              ├─column-name────────────────┤
              ├─host-variable──────────────┤
              ├─special-register───────────┤
              │                     (1)    │
              ├─(scalar-fullselect)────────┤
              │                  (2)       │
              ├─labeled-duration───────────┤
              │                 (3)        │
              ├─case-expression────────────┤
              │                    (4)     │
              ├─cast-specification─────────┤
              │                       (5)  │
              ├─dereference-operation──────┤
              │               (6)          │
              ├─OLAP-function──────────────┤
              │              (7)           │
              ├─XML-function───────────────┤
              │                  (8)       │
              ├─method-invocation──────────┤
              │                  (9)       │
              ├─subtype-treatment──────────┤
              │                    (10)    │
              └─sequence-reference─────────┘
```

operator:

```
             (11)
├──┬─CONCAT──────┬───────────────────────────────────────────────────────┤
   ├─ / ─────────┤
   ├─ * ─────────┤
   ├─ + ─────────┤
   └─ - ─────────┘
```

Notes:

1 See "Scalar fullselect" on page 174 for more information.

2 See "Labeled durations" on page 174 for more information.

3 See "CASE expressions" on page 179 for more information.

4 See "CAST specifications" on page 181 for more information.

5 See “Dereference operations” on page 183 for more information.

6 See “OLAP functions” on page 184 for more information.

7 See “XML functions” on page 189 for more information.

8 See “Method invocation” on page 198 for more information.

9 See “Subtype treatment” on page 199 for more information.

10 See “Sequence reference” on page 199 for more information.

11 `||` may be used as a synonym for CONCAT.

Expressions without operators

If no operators are used, the result of the expression is the specified value.

Examples:

```
SALARY:SALARY'SALARY'MAX(SALARY)
```

Expressions with the concatenation operator

The concatenation operator (CONCAT) links two string operands to form a *string expression*.

The operands of concatenation must be compatible strings. Note that a binary string cannot be concatenated with a character string, including character strings defined as FOR BIT DATA (SQLSTATE 42884).

In a Unicode database, concatenation involving both character string operands and graphic string operands will first convert the character operands to graphic operands. Note that in a non-Unicode database, concatenation cannot involve both character and graphic operands.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second. Note that no check is made for improperly formed mixed data when doing concatenation.

The length of the result is the sum of the lengths of the operands.

The data type and length attribute of the result is determined from that of the operands as shown in the following table:

Table 12. Data Type and Length of Concatenated Operands

Operands	Combined Length Attributes	Result
CHAR(A) CHAR(B)	<255	CHAR(A+B)
CHAR(A) CHAR(B)	>254	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
CHAR(A) LONG VARCHAR	-	LONG VARCHAR
VARCHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
VARCHAR(A) VARCHAR(B)	>4000	LONG VARCHAR

Table 12. Data Type and Length of Concatenated Operands (continued)

Operands	Combined Length Attributes	Result
VARCHAR(A) LONG VARCHAR	-	LONG VARCHAR
LONG VARCHAR LONG VARCHAR	-	LONG VARCHAR
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) LONG VARCHAR	-	CLOB(MIN(A+32K, 2G))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 2G))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>127	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
GRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
VARGRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
VARGRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
VARGRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
LONG VARGRAPHIC LONG VARGRAPHIC	-	LONG VARGRAPHIC
DBCLOB(A) GRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) VARGRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) LONG VARGRAPHIC	-	DBCLOB(MIN(A+16K, 1G))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 1G))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2G))

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands.

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

Example 1: If FIRSTNME is Pierre and LASTNAME is Fermat, then the following:

```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

returns the value `Pierre Fermat`.

Example 2: Given:

- `COLA` defined as VARCHAR(5) with value `'AA'`
- `:host_var` defined as a character host variable with length 5 and value `'BB   '`
- `COLC` defined as CHAR(5) with value `'CC'`
- `COLD` defined as CHAR(5) with value `'DDDDD'`

The value of `COLA` **`CONCAT`** `:host_var` **`CONCAT`** `COLC` **`CONCAT`** `COLD` is `'AABB   CC   DDDDD'`

The data type is VARCHAR, the length attribute is 17 and the result code page is the database code page.

Example 3: Given:

`COLA` defined as CHAR(10)

`COLB` defined as VARCHAR(5)

The parameter marker in the expression:

```
COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15), because `COLA CONCAT COLB` is evaluated first, giving a result that is the first operand of the second CONCAT operation.

User-defined types

A user-defined type cannot be used with the concatenation operator, even if it is a distinct type with a source data type that is a string type. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE_DESCRIPTION, both of which had VARCHAR(25) data types, the following user-defined function, ATTACH, could be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator could be overloaded using a user-defined function to add the new data types.

```
CREATE FUNCTION CONCAT (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Expressions with arithmetic operators

If arithmetic operators are used, the result of the expression is a value derived from the application of the operators to the values of the operands.

If any operand can be null, or the database is configured with DFT_SQLMATHWARN set to yes, the result can be null.

If any operand has the null value, the result of the expression is the null value.

Arithmetic operators can be applied to signed numeric types and datetime types (see "Datetime arithmetic in SQL" on page 175). For example, USER+2 is invalid.

Sourced functions can be defined for arithmetic operations on distinct types with a source type that is a signed numeric type.

The prefix operator + (unary plus) does not change its operand. The prefix operator − (unary minus) reverses the sign of a nonzero operand; and if the data type of A is small integer, the data type of −A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, −, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero. These operators can also be treated as functions. Thus, the expression *"+"(a,b)* is equivalent to the expression *a+b*. "operator" function.

Arithmetic errors

If an arithmetic error such as zero divide or a numeric overflow occurs during the processing of an expression, an error is returned and the SQL statement processing the expression fails with an error (SQLSTATE 22003 or 22012).

A database can be configured (using DFT_SQLMATHWARN set to yes) so that arithmetic errors return a null value for the expression, issue a warning (SQLSTATE 01519 or 01564), and proceed with processing of the SQL statement. When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of a column function causes the row to be ignored in the determining the result of the column function. If the arithmetic error was an overflow, this may significantly impact the result values.
- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows to not be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Some examples are:

- add a case expression to check for zero divide and set the desired value for such a situation
- add additional predicates to handle nulls (like a check constraint on not nullable columns could become:

  ```
  check (c1*c2 is not null and c1*c2>5000)
  ```

  to cause the constraint to be violated on an overflow).

Two-integer operands

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a *large integer* unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

Integer and decimal operands

If one operand is an integer and the other is a decimal, the operation is performed in decimal using a temporary copy of the integer 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, the operation is performed in floating-point, the operands having first been converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

User-defined types as operands

A user-defined type cannot be used with arithmetic operators, even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Scalar fullselect

A *scalar fullselect*, as supported in an expression, is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name is based on the name of the column.

Datetime operations and durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called *durations*. Following is a definition of durations and a specification of the rules for datetime arithmetic.

A duration is a number representing an interval of time. There are four types of durations.

Labeled durations

labeled-duration:

```
├──┬─function──────┬──┬─YEAR─────────┬──────────────────────────────────────┤
   ├─(expression)──┤  ├─YEARS────────┤
   ├─constant──────┤  ├─MONTH────────┤
   ├─column-name───┤  ├─MONTHS───────┤
   └─host-variable─┘  ├─DAY──────────┤
                      ├─DAYS─────────┤
                      ├─HOUR─────────┤
                      ├─HOURS────────┤
                      ├─MINUTE───────┤
                      ├─MINUTES──────┤
                      ├─SECOND───────┤
                      ├─SECONDS──────┤
                      ├─MICROSECOND──┤
                      └─MICROSECONDS─┘
```

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS. (The singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.) The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

Date duration

A *date duration* represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd.*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. (The period in the format indicates a DECIMAL data type.) The result of subtracting one date value from another, as in the expression HIREDATE − BRTHDATE, is a date duration.

Time duration

A *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.

Timestamp duration

A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyymmddhhmmss.zzzzzz*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *zzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one timestamp value from another is a timestamp duration.

Datetime arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of YEARS, MONTHS, or DAYS.
- If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.
- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of YEARS, MONTHS, or DAYS.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of HOURS, MINUTES, or SECONDS.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be a parameter marker.

Date arithmetic

Dates can be subtracted, incremented, or decremented.

Subtracting Dates: The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = DATE1 − DATE2.

```
If DAY(DATE2) <= DAY(DATE1)
then DAY(RESULT) = DAY(DATE1) - DAY(DATE2).

If DAY(DATE2) > DAY(DATE1)
then DAY(RESULT) = N + DAY(DATE1) - DAY(DATE2)
where N = the last day of MONTH(DATE2).
MONTH(DATE2) is then incremented by 1.

If MONTH(DATE2) <= MONTH(DATE1)
then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).

If MONTH(DATE2) > MONTH(DATE1)
then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
YEAR(DATE2) is then incremented by 1.

YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).
```

For example, the result of DATE('3/15/2000') − '12/31/1999' is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

Incrementing and decrementing dates: The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.
```

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 − X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.
```

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting time values: The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TIME1 − TIME2.

```
If SECOND(TIME2) <= SECOND(TIME1)
then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).

If SECOND(TIME2) > SECOND(TIME1)
then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
MINUTE(TIME2) is then incremented by 1.

If MINUTE(TIME2) <= MINUTE(TIME1)
then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).
```

```
If MINUTE(TIME1) > MINUTE(TIME1)
then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
HOUR(TIME2) is then incremented by 1.
HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
```

For example, the result of TIME('11:02:26') – '00:32:56' is 102930. (a duration of 10 hours, 29 minutes, and 30 seconds).

Incrementing and decrementing time values: The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. `TIME1 + X`, where "X" is a DECIMAL(6,0) number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Note: Although the time '24:00:00' is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (for example, time('24:00:00')±0 seconds = '00:00:00').

Timestamp arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting timestamps: The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is DECIMAL(20,6).

If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TS1 – TS2:

```
If MICROSECOND(TS2) <= MICROSECOND(TS1)
then MICROSECOND(RESULT) = MICROSECOND(TS1) -
MICROSECOND(TS2).
If MICROSECOND(TS2) > MICROSECOND(TS1)
then MICROSECOND(RESULT) = 1000000 +
MICROSECOND(TS1) - MICROSECOND(TS2)
and SECOND(TS2) is incremented by 1.
```

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

```
If HOUR(TS2) <= HOUR(TS1)
then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).
If HOUR(TS2) > HOUR(TS1)
then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
and DAY(TS2) is incremented by 1.
```

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

Incrementing and decrementing timestamps: The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. Microseconds overflow into seconds.

Precedence of operations

Expressions within parentheses and dereference operations are evaluated first from left to right. (Parentheses are also used in subselect statements, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.) When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

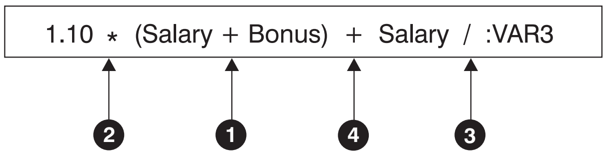


Figure 11. Precedence of Operations

CASE expressions

case-expression:

```
                                          ┌─ELSE NULL──────────────────┐
├──CASE──┬─searched-when-clause─┬─────────┼────────────────────────────┼──END──────────────┤
         └─simple-when-clause───┘         └─ELSE──result-expression────┘
```

searched-when-clause:

```
     ┌───────────────────────────────────────────────────────┐
     ▼                                                       │
├────WHEN──search-condition──THEN──┬─result-expression─┬─────┴──────────────────────┤
                                   └─NULL──────────────┘
```

simple-when-clause:

```
                    ┌───────────────────────────────────────────────────────┐
                    ▼                                                       │
├──expression───────WHEN──expression──THEN──┬─result-expression─┬──────────┴────────────┤
                                            └─NULL──────────────┘
```

CASE expressions allow an expression to be selected based on the evaluation of one or more conditions. In general, the value of the case-expression is the value of the *result-expression* following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is present then the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a case evaluates to unknown (because of NULLs), the case is not true and hence is treated the same way as a case that evaluates to false.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the *search-condition* in a searched-when-clause cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the *simple-when-clause*, the value of the *expression* prior to the first *WHEN* keyword is tested for equality with the value of the *expression* following the WHEN keyword. The data type of the *expression* prior to the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the WHEN keyword(s). The *expression* prior to the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is variant or has an external action (SQLSTATE 42845).

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All result 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 13 shows the equivalent expressions using CASE or these functions.

Table 13. Equivalent CASE Expressions

Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

CAST specifications

cast-specification:

```
├──CAST──(──┬─expression───────┬──AS──data-type──────────────────────────►
            ├─NULL─────────────┤
            └─parameter-marker─┘

►──┬─────────────────────────────────────┬──)────────────────────────────┤
   │       (1)                           │
   └─SCOPE──────────┬─typed-table-name─┬─┘
                    └─typed-view-name──┘
```

Notes:

1 The SCOPE clause only applies to the REF data type.

The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data type*. If the cast is not supported, an error is returned (SQLSTATE 42846).

expression
: If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data type*.

  When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

NULL
: If the cast operand is the keyword NULL, the result is a null value that has the specified *data type*.

parameter-marker
: A parameter marker (specified as a question mark character) is normally considered an expression, but is documented separately in this case because it

has a special meaning. If the cast operand is a *parameter-marker*, the specified *data type* is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

data type

The name of an existing data type. If the type name is not qualified, the SQL path is used to do data type resolution. A data type that has an associated attributes like length or precision and scale should include these attributes when specifying *data type* (CHAR defaults to a length of 1 and DECIMAL defaults to a precision of 5 and scale of 0 if not specified). Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, the supported target data types depend on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, any existing data type can be used.
- For a cast operand that is a parameter marker, the target data type can be any existing data type. If the data type is a user-defined distinct type, the application using the parameter marker will use the source data type of the user-defined distinct type. If the data type is a user-defined structured type, the application using the parameter marker will use the input parameter type of the TO SQL transform function for the user-defined structured type.

SCOPE

When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.

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 function 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)
```

Dereference operations

dereference-operation:

```
├──scoped-ref-expression── -> ──name1──┬──────────────────────┬──────────┤
                                       └─(─┬────────────────┬─)─┘
                                           │ ┌─,──────────┐ │
                                           │ ▼            │ │
                                           └───expression─┴─┘
```

The scope of the scoped reference expression is a table or view called the *target* table or view. The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

scoped-ref-expression
: An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a CAST specification with a SCOPE clause is required to give the reference a scope.

name1
: Specifies an unqualified identifier.

  If no parentheses follow *name1*, and *name1* matches the name of an attribute of the target type, then the value of the dereference operation is the value of the named column in the target row. In this case, the data type of the column (made nullable) determines the result type of the dereference operation. If no target row exists whose object identifier matches the reference expression, then the result of the dereference operation is null. If the dereference operation is used in a select list and is not included as part of an expression, *name1* becomes the result column name.

  If parentheses follow *name1*, or if *name1* does not match the name of an attribute of the target type, then the dereference operation is treated as a method invocation. The name of the invoked method is *name1*. The subject of the method is the target row, considered as an instance of its structured type. If no target row exists whose object identifier matches the reference expression, the subject of the method is a null value of the target type. The expressions inside parentheses, if any, provide the remaining parameters of the method invocation. The normal process is used for resolution of the method invocation. The result type of the selected method (made nullable) determines the result type of the dereference operation.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the *scoped-ref-expression* (SQLSTATE 42501).

A dereference operation can never modify values in the database. If a dereference operation is used to invoke a mutator method, the mutator method modifies a copy of the target row and returns the copy, leaving the database unchanged.

Examples:

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

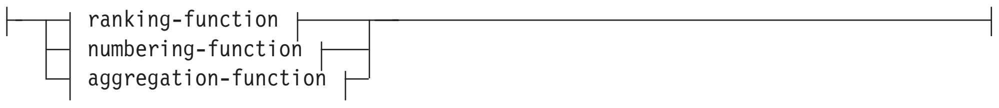
```
SELECT EMPNO, DEPTREF->DEPTNAME
  FROM EMPLOYEE
```

- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

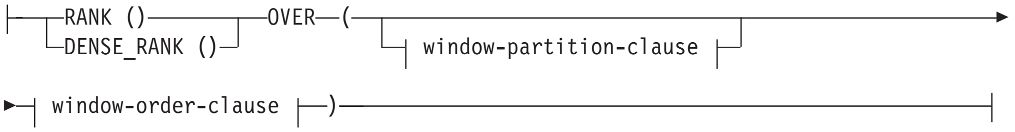
```
SELECT EMPNO, DEPTREF->BUDGET('1997') AS DEPTBUDGET97
  FROM EMPLOYEE
```

OLAP functions

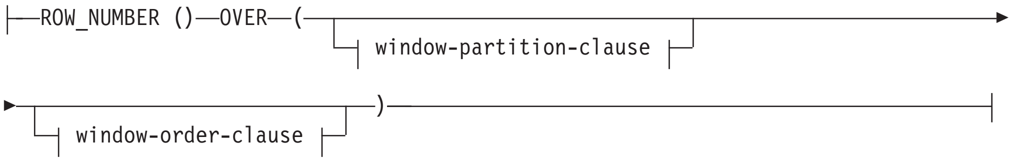
OLAP-function:



ranking-function:



numbering-function:



aggregation-function:

```
├──column-function──OVER──(──┬─────────────────────────────┬──────────────────────────►
                             └─┤ window-partition-clause ├─┘

►──┬──────────────────────────────────────────────────────────────────────┬──────────►
   └─┤ window-order-clause ├──┬───────────────────────────────────────┬──┘
                              └─┤ window-aggregation-group-clause ├──┘

►──┬─RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING─┬──)──────────┤
   └─┤ window-aggregation-group-clause ├────────────────────────┘
```

window-partition-clause:

```
                  ┌─,──────────────────────┐
                  ▼                        │
├──PARTITION BY──────partitioning-expression─┴─────────────────────────────────┤
```

window-order-clause:

```
              ┌─,──────────────────────────────────────────────┐
              ▼                            ┌─┤ asc option ├──┐ │
├──ORDER BY──────┬─sort-key-expression──┼──────────────────┼─┴─┬──────────────┤
                 │                         └─┤ desc option ├─┘   │
                 └─ORDER OF──table-designator────────────────────┘
```

asc option:

```
         ┌─NULLS LAST──┐
├──ASC───┼─────────────┼──────────────────────────────────────────────┤
         └─NULLS FIRST─┘
```

desc option:

```
          ┌─NULLS FIRST─┐
├──DESC───┼─────────────┼─────────────────────────────────────────────┤
          └─NULLS LAST──┘
```

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─┘
```

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing column function information as a scalar value in a query result. An OLAP function can be included in expressions in a select-list or the ORDER BY clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used as an argument of a column function (SQLSTATE 42607). The query result to which the OLAP function is applied is the result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows over which the function is applied, and in what order. When used with a column function, the applicable rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three month period.

The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If RANK is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

If DENSE_RANK (or DENSERANK) is specified, the rank of a row is defined as 1 plus the number of preceding rows that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

The ROW_NUMBER (or ROWNUMBER) function computes the sequential row number of the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order, as returned by the subselect (not according to any ORDER BY clause in the select-statement).

The data type of the result of RANK, DENSE_RANK or ROW_NUMBER is BIGINT. The result cannot be null.

PARTITION BY (*partitioning-expression***,...)**
: Defines the partition within which the function is applied. A *partitioning-expression* is an expression used in defining the partitioning of the result set. Each *column-name* referenced in a partitioning-expression must unambiguously reference a result set column of the OLAP function subselect statement (SQLSTATE 42702 or 42703). A partitioning-expression cannot include a scalar-fullselect (SQLSTATE 42822) or any function that is not deterministic or has an external action (SQLSTATE 42845).

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 (SQLSTATE 42822) or any function that is not deterministic or that has an external action (SQLSTATE 42845). This clause is required for the RANK and DENSE_RANK functions (SQLSTATE 42601).

ASC
: Uses the values of the sort-key-expression in ascending order.

DESC
: Uses the values of the sort-key-expression in descending order.

NULLS FIRST
: The window ordering considers null values *before* all non-null values in the sort order.

NULLS LAST
: The window ordering considers null values *after* all non-null values in the sort order.

ORDER OF *table-designator*
: Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data (SQLSTATE 428FI). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

window-aggregation-group-clause
: The aggregation group of a row R is a set of rows defined in relation to R (in

the ordering of the rows of R's partition). This clause specifies the aggregation group. If this clause is not specified, the default is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, providing a cumulative aggregation result.

ROWS
Indicates the aggregation group is defined by counting rows.

RANGE
Indicates the aggregation group is defined by an offset from a sort key.

group-start
Specifies the starting point for the aggregation group. The aggregation group end is the current row. Specification of the group-start clause is equivalent to a group-between clause of the form ″BETWEEN group-start AND CURRENT ROW″.

group-between
Specifies the aggregation group start and end based on either ROWS or RANGE.

group-end
Specifies the ending point for the aggregation group. The aggregation group start is the current row. Specification of the group-end clause is equivalent to a group-between clause of the form ″BETWEEN CURRENT ROW AND group-end″.

UNBOUNDED PRECEDING
Includes the entire partition preceding the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

UNBOUNDED FOLLOWING
Includes the entire partition following the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

CURRENT ROW
Specifies the start or end of the aggregation group based on the current row. If ROWS is specified, the current row is the aggregation group boundary. If RANGE is specified, the aggregation group boundary includes the set of rows with the same values for the *sort-key-expressions* as the current row. This clause cannot be specified in *group-bound2* if *group-bound1* specifies *value* FOLLOWING.

value **PRECEDING**
Specifies either the range or number of rows preceding the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow subtraction. This clause cannot be specified in *group-bound2* if *group-bound1* is CURRENT ROW or *value* FOLLOWING.

value **FOLLOWING**
Specifies either the range or number of rows following the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the

window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow addition.

Examples:

- Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than $30,000.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
    RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
  FROM EMPLOYEE WHERE SALARY+BONUS > 30000
  ORDER BY LASTNAME
```

  Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

```
ORDER BY RANK_SALARY
```

  or

```
ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
```

- Rank the departments according to their average total salary.

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
    RANK() OVER (ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  ORDER BY RANK_AVG_SAL
```

- Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value.

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL,
    DENSE_RANK() OVER
      (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

- Provide row numbers in the result of a query.

```
SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
    LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

- List the top five wage earners.

```
SELECT EMPNO, LASTNAME, 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.

XML functions

XML-function:

```
├──┬─XMLSERIALIZE──(──CONTENT──┤ XML-value-function ├──AS──┤ data-type ├──)─┬──────────┤
   └─XML2CLOB──(──┤ XML-value-function ├──)─────────────────────────────────┘
```

XML functions

XML-value-function:

```
├──┬─┤ xmlagg-function ├─────┬──────────────────────────────────────────┤
   ├─┤ xmlelement-function ├─┤
   ├─┤ xmlforest-function ├──┤
   └─┤ xmlconcat-function ├──┘
```

data-type:

```
├──┬─┬─CHARACTER─┬──┬───────────┬─────────┬──────────────────────────────┤
   │ └─CHAR──────┘  └─(integer)─┘         │
   ├─┬─VARCHAR────────────────────┬─(──integer──)─────┤
   │ └─┬─CHARACTER─┬──VARYING─────┘                   │
   │   └─CHAR──────┘                                  │
   └─┬─CLOB──────────────────────────┬─(──integer──┬───┬──)─┘
     └─┬─CHARACTER─┬──LARGE OBJECT───┘             ├─K─┤
       └─CHAR──────┘                               ├─M─┤
                                                   └─G─┘
```

xmlagg-function:

```
├──XMLAGG──────────────────────────────────────────────────────────────►

►──(──┤ xmlelement-function ├──┬───────────────────────────────────┬──)──┤
                               │           ┌─,──────────────────┐  │
                               │           ▼          ┌─ASC──┐  │  │
                               └─ORDER BY──┤ sort-key ├─┴─DESC─┴──┴──┘
```

sort-key:

```
├──┬─column-name──────────┬────────────────────────────────────────────┤
   └─sort-key-expression──┘
```

xmlconcat-function:

```
                        ┌─,─────────────────────────┐
                        ▼                           │
├──XMLCONCAT──(──────┤ XML-value-function ├─────┴──)────────────────────┤
```

xmlelement-function:

```
├──XMLELEMENT──(──NAME──element-name───────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────────┬──►
   └─,─┤ xmlnamespaces-function ├─,─┤ xmlattributes-function ├────────┘

►──┬─────────────────────────────┬──)────────────────────────────────────┤
   │     ┌─,───────────────────┐ │
   │     ▼                     │ │
   └─,───┤ element-content ├───┴─┘
```

xmlnamespaces-function:

```
                          ┌─,────────────────────────────────────────┐
                          ▼                                          │
├──XMLNAMESPACES──(──────┬─namespace-uri──AS──namespace-prefix─┬────┴──)──────────────┤
                          │                              (1)    │
                          └─┬─DEFAULT────┬─namespace-uri─────────┘
                            └─NO DEFAULT─┘
```

xmlattributes-function:

```
                          ┌─,────────────────────────────────────┐
                          ▼                                      │
├──XMLATTRIBUTES──(──────attribute-value──┬─────────────────────┬┴──)──────────────┤
                                          └─AS──attribute-name──┘
```

element-content:

```
├──┬─expression───────────────┬──────────────────────────────────────────────────┤
   └─┤ XML-value-function ├───┘
```

xmlforest-function:

```
├──XMLFOREST──────────────────────────────────────────────────────────────────────►

                                        ┌─,───────────────────────────────────────┐
                                        ▼                                         │
►──(──┬──────────────────────────────┬──┤ element-content ├──┬──────────────────┬─┴──)──────┤
      └─┤ xmlnamespaces-function ├─,─┘                       └─AS──element-name─┘
```

Notes:

1 DEFAULT or NO DEFAULT can only be specified once in arguments of XMLNAMESPACES.

XMLSERIALIZE
Returns the argument as a value of the data type specified as part of the function arguments. The schema is SYSIBM. The argument must be an expression of data type XML. The result has the data type specified by the user. If the result of the *XML-value-function* is null, the result is the null value.

CONTENT
Specifies that the value of *XML-value-function* can consist of more than one top-level element.

AS *data-type*
Supported data types are CHAR, VARCHAR, and CLOB. If no length is specified for CHAR, the length defaults to 1. A length value must be specified for VARCHAR. If no length is specified for CLOB, the length defaults to 1M.

XML2CLOB
Returns the argument as a CLOB value. The schema is SYSIBM. The argument must be an expression of data type XML. The result has the CLOB data type.

The XML2CLOB function is provided for compatibility with previous releases. The XMLSERIALIZE function should be used instead.

XMLAGG
Returns the concatenation of a set of XML values. The schema is SYSIBM. The function name cannot be specified as a qualified name. The data type of the result is XML, and its length is set to 1 073 741 823. If the XMLAGG function is applied to an empty set, the result is a null value. Otherwise, the result is the concatenation of the values in the set.

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.

Restrictions on using the XMLAGG function are:

- Column functions cannot be used as direct input (SQLSTATE 42607).
- XMLAGG cannot be used as a column function of an OLAP aggregate function (SQLSTATE 42601).

XMLCONCAT
Returns the concatenation of a variable number of XML arguments. The schema is SYSIBM. The arguments must be expressions of data type XML. The result has the same internal XML data type as the arguments. A null value returned from an input argument of the XMLCONCAT function is ignored. If all arguments of XMLCONCAT are null, the result is the null value. Otherwise, the result is the concatenation of the XML values.

XMLELEMENT
Constructs an XML element from the arguments. The schema is SYSIBM. The function name cannot be specified as a qualified name. This function takes an element name, an optional collection of namespace declarations, an optional collection of attributes, and zero or more arguments that will make up the element's content. If the element content is null, the result will be an empty element. The result data type is XML.

NAME
This keyword precedes the name of the XML element.

element-name
The name of the XML element. It should be an XML QName. See the W3C XML namespace specifications for more details on valid names.

xmlnamespaces-function
XML namespace declarations that are the result of the XMLNAMESPACES function.

xmlattributes-function
XML attributes that are the result of the XMLATTRIBUTES function.

element-content
The content of the generated elements is specified by an expression or a list of expressions. The data type of the result of the expression must be one of SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE, CHAR, VARCHAR, LONG VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB, DATE, TIME,

TIMESTAMP, XML, or any distinct type whose source type is one of the preceding data types. Character string data defined as FOR BIT DATA is not allowed. The expressions can be any SQL expression, but cannot include a scalar fullselect, or a subquery.

XMLATTRIBUTES

Constructs XML attributes from the arguments. The schema is SYSIBM. The function name cannot be specified as a qualified name. The result has the same internal XML data type as the arguments.

attribute-value

The attribute value is an expression. The data type of the result of the expression must be one of: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE, CHAR, VARCHAR, LONG VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB, DATE, TIME, TIMESTAMP, or any distinct type whose source type is one of the preceding data types. Character string data defined as FOR BIT DATA is not allowed. The expression can be any SQL expression but cannot include a scalar fullselect or a subquery. If the expression is not a simple column reference, an attribute name must be specified. Duplicate attribute names are not allowed (SQLSTATE 42713).

attribute-name

The attribute name is an SQL identifier. It 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. The attribute name cannot be `xmlns` or prefixed with `xmlns:`. A namespace is declared using the function XMLNAMESPACES.

XMLFOREST

Constructs a sequence (forest) of XML elements from the arguments. The schema is SYSIBM. The result has the same internal XML data type as the arguments. A null value returned from an input argument of the XMLFOREST function is ignored (no element generated). If all arguments of XMLFOREST are null, the result is the null value. Otherwise, the result is the sequence of XML elements returned.

xmlnamespaces-function

XML namespace declarations that are the result of the XMLNAMESPACES function.

element-content

The element content is an expression. The data type of the result of the expression must be one of: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE, CHAR, VARCHAR, LONG VARCHAR, CLOB, GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB, DATE, TIME, TIMESTAMP, or any distinct type whose source type is one of the preceding data types. Character string data defined as FOR BIT DATA is not allowed. The expression can be any SQL expression but cannot include a scalar fullselect or a subquery. If the expression is not a simple column reference, an element name must be specified.

element-name

The element name is 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.

XMLNAMESPACES

Constructs XML namespace declarations from the arguments. The schema is SYSIBM. The result data type is XML. The namespace declarations are in the scope of the elements that are generated by the XMLELEMENT and XMLFOREST functions. You can use the predefined namespace prefix, `xml`, without explicitly defining it. The `xml` prefix cannot be redefined.

namespace-uri
The namespace universal resource identifier (URI) is a character string constant. This character string constant cannot be empty.

namespace-prefix
The namespace prefix is an SQL identifier. It must be of the form of an XML NCName (SQLSTATE 42635). See the W3C XML namespace specifications for more details on valid names. The prefix cannot be `xml` or `xmlns`. The prefix must be unique within the list of namespace declarations.

DEFAULT *namespace-uri*
Specifies the default namespace to use for this XML element, and for all XML elements that are contained in this XML element, except for those *xmlnamespaces-functions* that specify NO DEFAULT for the namespace.

NO DEFAULT
Specifies that no default namespace is to be used for this XML element, and for all XML elements that are contained in this XML element, except for those *xmlnamespaces-functions* that specify a DEFAULT for the namespace.

Examples:

- Construct a CLOB from the expression returned by the XMLELEMENT function. The query

```
SELECT e.empno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Emp", e.firstnme || ' ' ||
    e.lastname) AS CLOB)
  AS "Result" FROM employee e
  WHERE e.edlevel = 12
```

  produces the following result:

```
EMPNO      Result
000290     <Emp>JOHN PARKER</Emp>
000310     <Emp>MAUDE SETRIGHT</Emp>
```

- Produce a department element (for each department) with a list of employees, sorted by employee last name:

```
SELECT XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Department",
                           XMLATTRIBUTES(e.workdept AS "name"),
                           XMLAGG(XMLELEMENT(NAME "emp", e.lastname)
                             ORDER BY e.lastname
                                 )
                          ) AS CLOB ) AS "dept_list"
  FROM employee e
  WHERE e.workdept IN ('C01','E21')
  GROUP BY workdept
```

  This query produces the following output. Note that no spaces or newline characters are actually produced in the result; the following output has been formatted for clarity.

```
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>
```

- For each department that reports to department A00, create an empty XML element named Mgr with an ID attribute equal to the MGRNO. The query

  ```
  SELECT d.deptno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Mgr",
    XMLATTRIBUTES(d.mgrno AS "ID")) AS CLOB)
    AS "Result" FROM department d
    WHERE d.admrdept = 'A00'
  ```

  produces the following result:

  ```
  DEPTNO     Result
  A00        <Mgr ID="000010"/>
  B01        <Mgr ID="000020"/>
  C01        <Mgr ID="000030"/>
  D01        <Mgr/>
  E01        <Mgr ID="000050">
  ```

- Produce an XML element named Emp for each employee, with nested elements for the employee's full name and the date on which the employee was hired. The query

  ```
  SELECT e.empno, XMLSERIALIZE
   (CONTENT XMLELEMENT(NAME "Emp",
    XMLELEMENT(NAME "name", e.firstnme || ' ' || e.lastname),
    XMLELEMENT(NAME "hiredate", e.hiredate)) AS CLOB)
    AS "Result" FROM employee e
    WHERE e.edlevel = 12
  ```

  produces the following result (formatted here for convenience; the output XML fragment has no extraneous white space characters):

  ```
  EMPNO      Result
  000290     <Emp>
             <name>JOHN PARKER</name>
             <hiredate>1980-05-30</hiredate>
             </Emp>
  000310     <Emp>
             <name>MAUDE SETRIGHT</name>
             <hiredate>1964-09-12</hiredate>
             </Emp>
  ```

- Using the XMLATTRIBUTES function, along with the XMLSERIALIZE and XMLELEMENT functions, construct the XML attributes. The query

  ```
  SELECT XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Emp_Exempt",
    XMLATTRIBUTES(e.firstnme, e.lastname AS "name:last", e.midinit)) AS CLOB)
      AS "Result"
    FROM employee e
    WHERE e.lastname='GEYER'
  ```

  produces the following result:

  ```
  <Emp_Exempt
     FIRSTNME="JOHN"
     name_last="GEYER"
     MIDINIT="B">
  </Emp_Exempt>
  ```

- Produce an Emp element for each qualified employee, containing a sequence of sub-elements produced from arguments of XMLFOREST. The query

  ```
  SELECT e.empno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "Emp",
      XMLFOREST(e.firstnme || ' ' || e.lastname AS "Name" , e.hiredate)) AS CLOB)
    AS "Result" FROM employee e
    WHERE e.edlevel = 12
  ```

produces the following result (formatted here for convenience; the output XML has no extraneous white space characters):

```
EMPNO      Result
000290     <Emp>
             <Name>JOHN PARKER</Name>
             <HIDREDATE>1980-05-30</HIREDATE>
           </Emp>
000310     <Emp>
             <Name>MAUDE SETRIGHT</Name>
             <HIREDATE>1964-09-12</HIREDATE>
           </Emp>
```

- Produce the last name and first name for each qualified employee. The query

  ```
  SELECT e.empno, XMLSERIALIZE(CONTENT XMLCONCAT(XMLELEMENT(NAME "firstname",
        e.firstnme),
      XMLELEMENT(NAME "lastname", e.lastname)) AS CLOB)
    AS "Result" FROM employee e
    WHERE e.edlevel = 12
  ```

  produces the following output:

  ```
  EMPNO      Result
  000290     <firstname>JOHN</firstname><lastname>PARKER</lastname>
  000310     <firstname>MAUDE</firstname><lastname>SETRIGHT</lastname>
  ```

- Produce an XML element named `adm:employee` and an XML attribute `adm:department`, both associated with a namespace whose prefix is `adm`.

  ```
  SELECT empno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "adm:employee",
    XMLNAMESPACES('http://www.adm.com' AS "adm"),
    XMLATTRIBUTES(workdept AS "adm:department"), lastname) AS CLOB) AS "Result"
    FROM employee
    WHERE job = 'ANALYST'
  ```

  This query generates an XML value with textual representation, as in the following output (formatted here for convenience; the output XML fragment has no extraneous white space characters, and the output generally appears as one line):

```
EMPNO  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>
```

- Generate a sequence of elements produced from arguments of XMLFOREST. Also, declare a default namespace to be associated with the first element, and a namespace whose prefix is `d` associated with the second element.

  ```
  SELECT empno, XMLSERIALIZE(CONTENT XMLFOREST
    (XMLNAMESPACES(DEFAULT 'http://hr.org',
      'http://fed.gov' AS "d"),
      lastname, job AS "d:job") AS CLOB) AS "Result"
    FROM employee
    WHERE edlevel = 12
  ```

  This query generates an XML value with a textual representation as in the following output (formatted here for convenience; the output XML fragment has no extraneous white space characters, and the output generally appears as one line):

```
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>
```

- Produce an XML element named employee associated with a default namespace and a sub-element named department which does not use the default namespace.

  ```
  SELECT emp.empno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "employee",
    XMLNAMESPACES(DEFAULT 'http://hr.org'), emp.lastname,
    XMLELEMENT(NAME "department",
      XMLNAMESPACES(NO DEFAULT),
        emp.workdept)) AS CLOB) AS "Result"
    FROM employee emp
    WHERE emp.edlevel = 12
  ```

  This query generates an XML value, whose serialized form looks like the following output (formatted here for convenience; the output XML fragment has no extraneous white space characters, and the output generally appears as one line):

  ```
  EMPNO  Result
  000290 <employee xmlns="http://hr.org">PARKER
                <department xmlns="">E11
                </department>
         </employee>
  000310 <employee xmlns="http://hr.org">SETRIGHT
                <department xmlns="">E11
                </department>
         </employee>
  ```

- Produce an XML element named employee associated with a default namespace, and a sub-element named job which does not use the default namespace and whose sub-element department uses a default namespace.

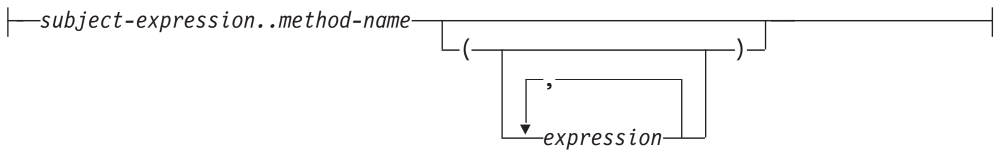
  ```
  SELECT emp.empno, XMLSERIALIZE(CONTENT XMLELEMENT(NAME "employee",
    XMLNAMESPACES(DEFAULT 'http://hr.org'), emp.lastname,
    XMLELEMENT(NAME "job",
      XMLNAMESPACES(NO DEFAULT), emp.job,
      XMLELEMENT(NAME "department",
        XMLNAMESPACES(DEFAULT 'http://adm.org'), emp.workdept))) AS CLOB) AS "Result"
    FROM employee emp
    WHERE emp.edlevel = 12
  ```

  This query generates an XML value, whose serialized form looks like the following output (formatted here for convenience; the output XML fragment has no extraneous white space characters, and the output generally appears as one line):

  ```
  EMPNO   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>
  ```

Method invocation

method-invocation:



Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

subject-expression
: An expression with a static result type that is a user-defined structured type.

method-name
: The unqualified name of a method. The static type of *subject-expression* or one of its supertypes must include a method with the specified name.

(*expression***,...)**
: The arguments of *method-name* are specified within parentheses. Empty parentheses can be used to indicate that there are no arguments. The *method-name* and the data types of the specified argument expressions are used to resolve to the specific method, based on the static type of *subject-expression*.

The double-dot operator used for method invocation is a high precedence left to right infix operator. For example, the following two expressions are equivalent:

```
a..b..c + x..y..z
```

and

```
((a..b)..c) + ((x..y)..z)
```

If a method has no parameters other than its subject, it can be invoked with or without parentheses. For example, the following two expressions are equivalent:

```
point1..x
point1..x()
```

Null subjects in method calls are handled as follows:

- If a system-generated mutator method is invoked with a null subject, an error results (SQLSTATE 2202D)
- If any method other than a system-generated mutator is invoked with a null subject, the method is not executed, and its result is null. This rule includes user-defined methods with SELF AS RESULT.

When a database object (a package, view, or trigger, for example) is created, the best fit method that exists for each of its method invocations is found.

Note: Methods of types defined WITH FUNCTION ACCESS can also be invoked using the regular function notation. Function resolution considers all functions, as well as methods with function access as candidate functions. However, functions cannot be invoked using method invocation. Method resolution considers all methods and does not consider functions as candidate methods. Failure to resolve to an appropriate function or method results in an error (SQLSTATE 42884).

Example:

- Use the double-dot operator to invoke a method called AREA. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that the method AREA has been defined previously for the CIRCLE type as `AREA() RETURNS DOUBLE`.

  ```
  SELECT CIRCLE_COL..AREA() FROM RINGS
  ```

Subtype treatment

subtype-treatment:

```
├──TREAT──(──expression──AS──data-type──)────────────────────────────────┤
```

The *subtype-treatment* is used to cast a structured type expression into one of its subtypes. The static type of *expression* must be a user-defined structured type, and that type must be the same type as, or a supertype of, *data-type*. If the type name in *data-type* is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is *data-type*, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not *data-type* or a subtype of *data-type*, an error is returned (SQLSTATE 0D000).

Example:

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as `RGB() RETURNS DOUBLE`.

  ```
  SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
    FROM RINGS
  ```

  At run time, if there are instances of dynamic type CIRCLE, an error is raised (SQLSTATE 0D000). This error can be avoided by using the TYPE predicate in a CASE expression, as follows:

  ```
  SELECT (CASE
    WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
      THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
      ELSE NULL
    END)
    FROM RINGS
  ```

Sequence reference

sequence-reference:

```
├──┬─┤ nextval-expression ├─┬────────────────────────────────────────┤
   └─┤ prevval-expression ├─┘
```

nextval-expression:

```
├──NEXT VALUE FOR──sequence-name──────────────────────────────────────┤
```

prevval-expression:

```
├──PREVIOUS VALUE FOR──sequence-name──────────────────────────────────────┤
```

NEXT VALUE FOR *sequence-name*
A NEXT VALUE expression generates and returns the next value for the sequence specified by *sequence-name*.

PREVIOUS VALUE FOR *sequence-name*
A PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current application process. This value can be referenced repeatedly by using PREVIOUS VALUE expressions that specify the name of the sequence. There may be multiple instances of PREVIOUS VALUE expressions specifying the same sequence name within a single statement; they all return the same value. In a partitioned database environment, a PREVIOUS VALUE expression may not return the most recently generated value.

A PREVIOUS VALUE expression can only be used if a NEXT VALUE expression specifying the same sequence name has already been referenced in the current application process, in either the current or a previous transaction (SQLSTATE 51035).

Notes:

- ***Compatibilities***
  - For compatibility with previous versions of DB2:
    - NEXTVAL and PREVVAL can be specified in place of NEXT VALUE and PREVIOUS VALUE.
- A new value is generated for a sequence when a NEXT VALUE expression specifies the name of that sequence. However, if there are multiple instances of a NEXT VALUE expression specifying the same sequence name within a query, the counter for the sequence is incremented only once for each row of the result, and all instances of NEXT VALUE return the same value for a row of the result.
- The same sequence number can be used as a unique key value in two separate tables by referencing the sequence number with a NEXT VALUE expression for the first row (this generates the sequence value), and a PREVIOUS VALUE expression for the other rows (the instance of PREVIOUS VALUE refers to the sequence value most recently generated in the current session), as shown below:

```
INSERT INTO order(orderno, cutno)
  VALUES (NEXT VALUE FOR order_seq, 123456);

INSERT INTO line_item (orderno, partno, quantity)
  VALUES (PREVIOUS VALUE FOR order_seq, 987654, 1);
```

- NEXT VALUE and PREVIOUS VALUE expressions can be specified in the following places:
  - select-statement or SELECT INTO statement (within the select-clause, provided that the statement does not contain a DISTINCT keyword, a GROUP BY clause, an ORDER BY clause, a UNION keyword, an INTERSECT keyword, or EXCEPT keyword)
  - INSERT statement (within a VALUES clause)
  - INSERT statement (within the select-clause of the fullselect)
  - UPDATE statement (within the SET clause (either a searched or a positioned UPDATE statement), except that NEXT VALUE cannot be specified in the select-clause of the fullselect of an expression in the SET clause)

  - SET Variable statement (except within the select-clause of the fullselect of an expression; a NEXT VALUE expression can be specified in a trigger, but a PREVIOUS VALUE expression cannot)
  - VALUES INTO statement (within the select-clause of the fullselect of an expression)
  - CREATE PROCEDURE statement (within the routine-body of an SQL procedure)
  - CREATE TRIGGER statement within the triggered-action (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
- NEXT VALUE and PREVIOUS VALUE expressions cannot be specified (SQLSTATE 428F9) in the following places:
  - join condition of a full outer join
  - DEFAULT value for a column in a CREATE or ALTER TABLE statement
  - generated column definition in a CREATE OR ALTER TABLE statement
  - summary table definition in a CREATE TABLE or ALTER TABLE statement
  - condition of a CHECK constraint
  - CREATE TRIGGER statement (a NEXT VALUE expression may be specified, but a PREVIOUS VALUE expression cannot)
  - CREATE VIEW statement
  - CREATE METHOD statement
  - CREATE FUNCTION statement
- In addition, a NEXT VALUE expression cannot be specified (SQLSTATE 428F9) in the following places:
  - CASE expression
  - parameter list of an aggregate function
  - subquery in a context other than those explicitly allowed above
  - SELECT statement for which the outer SELECT contains a DISTINCT operator
  - join condition of a join
  - SELECT statement for which the outer SELECT contains a GROUP BY clause
  - SELECT statement for which the outer SELECT is combined with another SELECT statement using the UNION, INTERSECT, or EXCEPT set operator
  - nested table expression
  - parameter list of a table function
  - WHERE clause of the outer-most SELECT statement, or a DELETE or UPDATE statement
  - ORDER BY clause of the outer-most SELECT statement
  - select-clause of the fullselect of an expression, in the SET clause of an UPDATE statement
  - IF, WHILE, DO ... UNTIL, or CASE statement in an SQL routine
- When a value is generated for a sequence, that value is consumed, and the next time that a value is requested, a new value will be generated. This is true even when the statement containing the NEXT VALUE expression fails or is rolled back.

  If an INSERT statement includes a NEXT VALUE expression in the VALUES list for the column, and if an error occurs at some point during the execution of the INSERT (it could be a problem in generating the next sequence value, or a problem with the value for another column), then an insertion failure occurs

(SQLSTATE 23505), and the value generated for the sequence is considered to be consumed. In some cases, reissuing the same INSERT statement might lead to success.

For example, consider an error that is the result of the existence of a unique index for the column for which NEXT VALUE was used and the sequence value generated already exists in the index. It is possible that the next value generated for the sequence is a value that does not exist in the index and so the subsequent INSERT would succeed.

- If in generating a value for a sequence, the maximum value for the sequence is exceeded (or the minimum value for a descending sequence) and cycles are not permitted, then an error occurs (SQLSTATE 23522). In this case, the user could ALTER the sequence to extend the range of acceptable values, or enable cycles for the sequence, or DROP and CREATE a new sequence with a different data type that has a larger range of values.

  For example, a sequence may have been defined with a data type of SMALLINT, and eventually the sequence runs out of assignable values. DROP and re-create the sequence with the new definition to redefine the sequence as INTEGER.
- A reference to a NEXT VALUE expression in the select statement of a cursor refers to a value that is generated for a row of the result table. A sequence value is generated for a NEXT VALUE expression for each row that is fetched from the database. If blocking is done at the client, the values may have been generated at the server prior to the processing of the FETCH statement. This can occur when there is blocking of the rows of the result table. If the client application does not explicitly FETCH all the rows that the database has materialized, then the application will not see the results of all the generated sequence values (for the materialized rows that were not returned).
- A reference to a PREVIOUS VALUE expression in the select statement of a cursor refers to a value that was generated for the specified sequence prior to the opening of the cursor. However, closing the cursor can affect the values returned by PREVIOUS VALUE for the specified sequence in subsequent statements, or even for the same statement in the event that the cursor is reopened. This would be the case when the select statement of the cursor included a reference to NEXT VALUE for the same sequence name.

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;
```

Related reference:

- "Identifiers" on page 58
- "TYPE predicate" on page 222
- "CHAR" on page 288
- "INTEGER" on page 360
- "Fullselect" on page 485
- "CREATE TABLE statement" in the *SQL Reference, Volume 2*
- "Methods" on page 160
- "CREATE FUNCTION (SQL Scalar, Table, or Row) statement" in the *SQL Reference, Volume 2*
- "Casting between data types" on page 96
- "Assignments and comparisons" on page 99
- "Rules for result data types" on page 113
- "Rules for string conversions" on page 117

Predicates

Predicates

A *predicate* specifies a condition that is true, false, or unknown about a given row or group.

The following rules apply to all types of predicates:

- All values specified in a predicate must be compatible.
- An expression used in a basic, quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4 000, a graphic string with a length attribute greater than 2 000, or a LOB string of any size.
- The value of a host variable 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 DATALINK value is limited to the NULL predicate.
- 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.

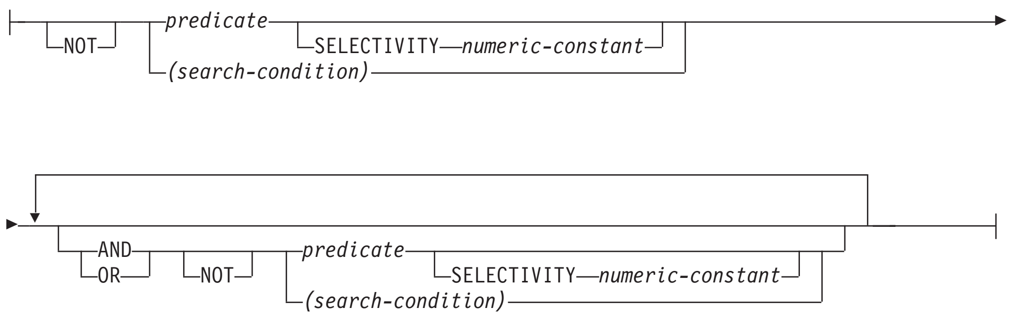
A fullselect is a form of the SELECT statement that, when used in a predicate, is also called a *subquery*.

Related reference:

- "Fullselect" on page 485
- "Rules for string conversions" on page 117

Search conditions

search-condition:



A *search condition* specifies a condition that is “true,” “false,” or “unknown” about a given row.

The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in Table 14, in which P and Q are any predicates:

Table 14. Truth Tables for AND and OR

P	**Q**	**P AND Q**	**P OR Q**
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

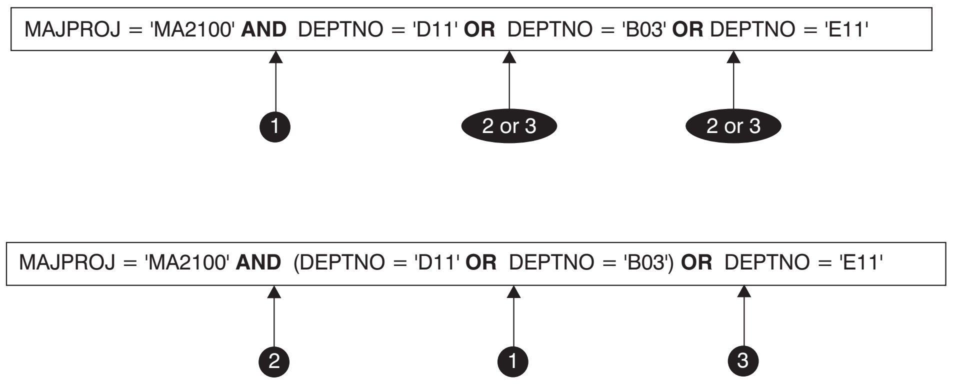

MAJPROJ = 'MA2100' AND DEPTNO = 'D11' OR DEPTNO = 'B03' OR DEPTNO = 'E11'
1
2 or 3
2 or 3
MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03') OR DEPTNO = 'E11'
2
1
3

Figure 12. Search Conditions Evaluation Order

SELECTIVITY *value*

The SELECTIVITY clause is used to indicate to DB2 what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified only when the predicate is a user-defined predicate.

A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function `foo` is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:

```
SELECT *
  FROM STORES
  WHERE foo(parm,parm) = 1 SELECTIVITY 0.004
```

The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.ROUTINES view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).

A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:

- the predicate specification is present in the CREATE FUNCTION statement
- the UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
- there is no negation (NOT operator)

Examples:

In the following query, the `within` UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate.

```
SELECT *
  FROM customers
  WHERE within(location, :sanJose) = 1 SELECTIVITY  0.2
```

However, the presence of `within` in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```
SELECT *
  FROM customers
  WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY  0.3
```

In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance from one store to another is computed by the radius of the city in which the customers live.

```
SELECT *
  FROM customers, stores
  WHERE distance(customers.loc, stores.loc) <
    CityRadius(stores.loc) SELECTIVITY 0.02
```

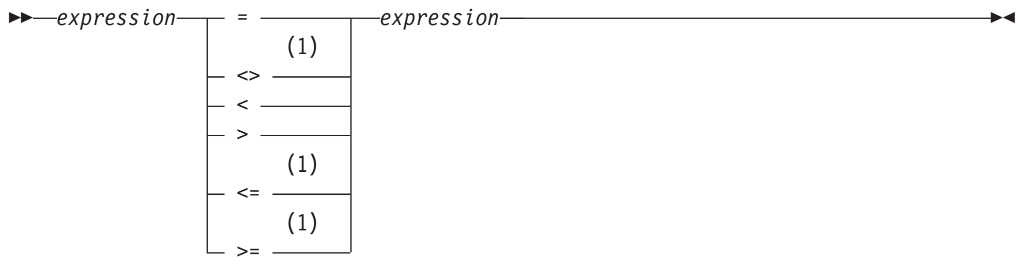
In the above query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

However, since the result produced by CityRadius is used as a range producer function, the above user-defined predicate will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.

Related reference:

- "CREATE FUNCTION (External Scalar) statement" in the *SQL Reference, Volume 2*

Basic predicate



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 that uses these operators, and are not recommended for use when writing new SQL statements.

A *basic predicate* compares two values.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

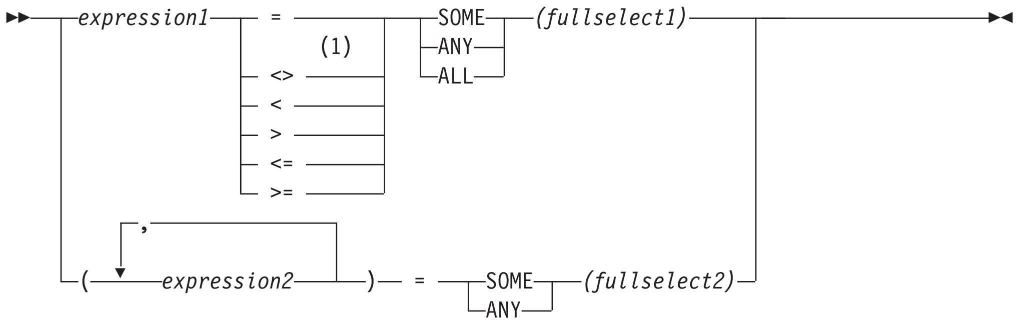
For values *x* and *y*:

Predicate	**Is True If and Only If...**
x **=** *y*	*x* is equal to *y*
x **<>** *y*	*x* is not equal to *y*
x **<** *y*	*x* is less than *y*
x **>** *y*	*x* is greater than *y*
x **>=** *y*	*x* is greater than or equal to *y*
x **<=** *y*	*x* is less than or equal to *y*

Examples:

```
EMPNO='528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

Quantified predicate



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 that uses these operators, and are not recommended for use when writing new SQL statements.

A *quantified predicate* compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

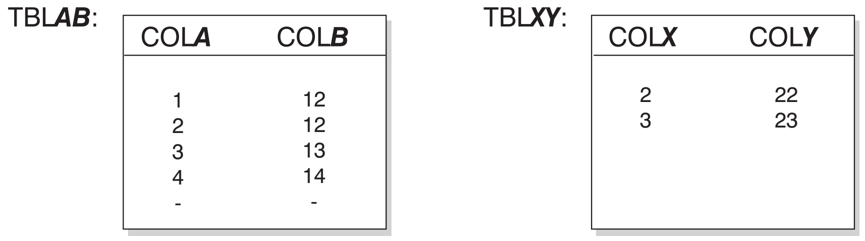
When ALL is specified:

- The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
- The result is false if the specified relationship is false for at least one value returned by the fullselect.
- The result is unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:

- The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
- The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
- The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.


TBLAB:

COLA	COLB
1	12
2	12
3	13
4	14
-	-

TBLXY:

COLX	COLY
2	22
3	23

Figure 13.

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
```

BETWEEN predicate

```
►►─expression─┬─────┬─BETWEEN─expression─AND─expression──────────────────►◄
              └─NOT─┘
```

The BETWEEN predicate compares a value with a range of values.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,
value1 < value2 OR value1 > value3.
```

The first operand (expression) cannot include a function that is variant or has an external action (SQLSTATE 426804).

Given a mixture of datetime values and string representations of datetime values, all values are converted to the data type of the datetime operand.

Examples:

Example 1

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Results in all salaries between $20,000.00 and $40,000.00.

Example 2

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Assuming :HV1 is 5000, results in all salaries below $25,000.00 and above $40,000.00.

EXISTS predicate

```
►►──EXISTS──(fullselect)──────────────────────────────────────────────►◄
```

The EXISTS predicate tests for the existence of certain rows.

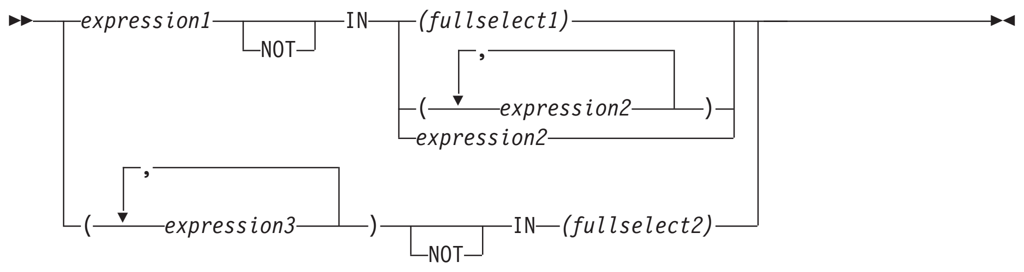
The fullselect may specify any number of columns, and

- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.

Example:

```
   EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```

IN predicate



The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:

  ```
  expression IN expression
  ```

  is equivalent to a basic predicate of the form:

  ```
  expression = expression
  ```

- An IN predicate of the form:

  ```
  expression IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  expression = ANY (fullselect)
  ```

- An IN predicate of the form:

  ```
  expression NOT IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  expression <> ALL (fullselect)
  ```

- An IN predicate of the form:

  ```
  expression IN (expressiona, expressionb, ..., expressionk)
  ```

  is equivalent to:

  ```
  expression = ANY (fullselect)
  ```

  where fullselect in the values-clause form is:

  ```
  VALUES (expressiona), (expressionb), ..., (expressionk)
  ```

- An IN predicate of the form:

  ```
  (expressiona, expressionb,..., expressionk) IN (fullselect)
  ```

  is equivalent to a quantified predicate of the form:

  ```
  (expressiona, expressionb,..., expressionk) = ANY (fullselect)
  ```

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each *expression3* value and its corresponding column of *fullselect2* in the IN predicate must be compatible. The rules for result data types can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary, the code page is determined by applying rules for string conversions to the IN list first, and then to the predicate, using the derived code page for the IN list as the second operand.

Examples:

Example 1: The following evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

Example 2: The following evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

Example 3: Given the following information, this example evaluates to true if the specific value in the row of the COL_1 column matches any of the values in the list:

Table 15. IN Predicate example

Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

the two host variables will be converted to code page 850, based on the rules for string conversions.

Example 4: The following evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

```
YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                   YEAR(CURRENT DATE - 1 YEAR),
                   YEAR(CURRENT DATE - 2 YEARS))
```

Example 5: The following evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```

Related reference:

- "Rules for result data types" on page 113
- "Rules for string conversions" on page 117

LIKE predicate

```
>>──match-expression──┬─────┬──LIKE──pattern-expression──────────────────►
                      └─NOT─┘

►──┬───────────────────────────┬─────────────────────────────────────────►◄
   └─ESCAPE──escape-expression─┘
```

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and the percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for *match-expression*, *pattern-expression*, and *escape-expression* are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

match-expression
: An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

  The expression can be specified by:
  - A constant
  - A special register
  - A host variable (including a locator variable or a file reference variable)
  - A scalar function
  - A large object locator
  - A column name
  - An expression concatenating any of the above

pattern-expression
: An expression that specifies the string that is to be matched.

  The expression can be specified by:
  - A constant
  - A special register
  - A host variable
  - A scalar function whose operands are any of the above
  - An expression concatenating any of the above

  with the following restrictions:
  - No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC, or DBCLOB. In addition it cannot be a BLOB file reference variable.
  - The actual length of *pattern-expression* cannot be more than 32 672 bytes.

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.

You need two underscore characters to match a Unicode supplementary graphic character, because such a character is represented by two UCS-2 characters in a graphic column. However, you only need one underscore character to match a Unicode supplementary character in a character column.

escape-expression

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC, or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- 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).

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.

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')
```

NULL predicate

```
►►──expression──IS──┬─────┬──NULL──────────────────────────────────────────►◄
                    └─NOT─┘
```

The NULL predicate tests for null values.

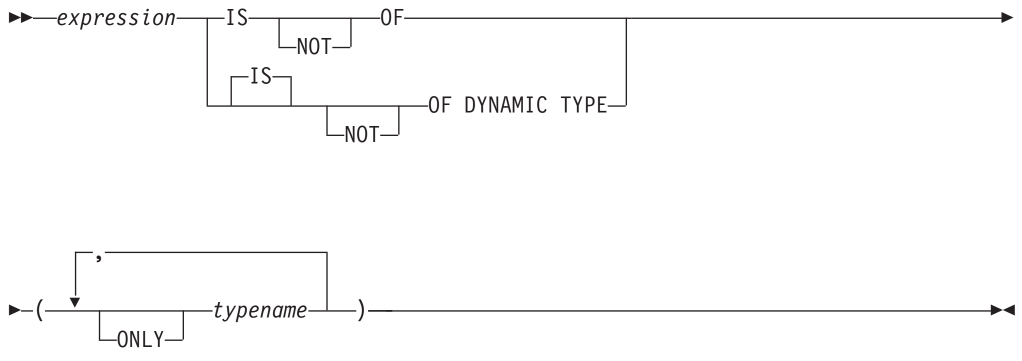
The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

Examples:

```
PHONENO IS NULL

SALARY IS NOT NULL
```

TYPE predicate



A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.

The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This may differ from the target type of an expression involving the reference which is called the static type of the expression.

If the value of *expression* is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the *expression* is a subtype of one of the structured types specified by *typename*, otherwise the result is false. If ONLY precedes any *typename* the proper subtypes of that type are not considered.

If *typename* is not qualified, it is resolved using the SQL path. Each *typename* must identify a user-defined type that is in the type hierarchy of the static type of *expression* (SQLSTATE 428DU).

The DEREF function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of *expression* is the target type of the reference.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

Examples:

A table hierarchy exists with root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table, ACTIVITIES, includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. The following is a type predicate that evaluates to true when a row corresponding to WHO_RESPONSIBLE is a manager:

```
DEREF (WHO_RESPONSIBLE) IS OF (MGR)
```

If a table contains a column EMPLOYEE of type EMP, EMPLOYEE may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```
EMPL IS OF (MGR)
```

returns true when EMPL is not null and is actually a manager.

Related reference:
- "DEREF" on page 316

Chapter 3. Functions

Functions overview

A *function* is an operation that is denoted by a function name followed by a pair of parentheses enclosing the specification of arguments (there may be no arguments).

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

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

Functions are classified as aggregate (column) functions, scalar functions, row functions, or table functions.

- The argument of an *column function* is a collection of like values. A column function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The arguments of a *scalar function* are individual scalar values, which can be of different types and have different meanings. A scalar function returns a single value (possibly null), and can be specified in an SQL statement wherever an expression can be used.
- The argument of a *row function* is a structured type. A row function returns a row of built-in data types and can only be specified as a transform function for a structured type.
- The arguments of a *table function* are individual scalar values, which can be of different types and have different meanings. A table function returns a table to the SQL statement, and can be specified only within the FROM clause of a SELECT statement.

The function name, combined with the schema, gives the fully qualified name of a function. The combination of schema, function name, and input parameters make up a *function signature*.

In some cases, the input parameter type is specified as a specific built-in data type, and in other cases, it is specified through a general variable like *any-numeric-type*. If a particular data type is specified, an exact match will only occur with the specified data type. If a general variable is used, each of the data types associated with that variable results in an exact match.

Additional functions may be available, because user-defined functions can be created in different schemas, using one of the function signatures as a source. You can also create external functions in your applications.

Related concepts:

- "Aggregate functions" on page 257

Related reference:

- "Functions" on page 152
- "Subselect" on page 450
- "CREATE FUNCTION statement" in the *SQL Reference, Volume 2*

Supported functions and SQL administrative routines

Table 16 summarizes information about the supported functions and SQL administrative routines. The function name, combined with the schema, gives the fully qualified name of a function. The “Input parameters” column shows the expected data type for each argument during function invocation. Many of the functions include variations of the input parameters, allowing either different data types or different numbers of arguments to be used. The combination of schema, function name and input parameters makes up a function signature. The “Returns” column shows the possible data types of values returned by the function.

For lists of the supported built-in functions classified by type, see the following tables:
- Aggregate functions (Table 17 on page 250)
- Cast scalar functions (Table 18 on page 251)
- Datalink scalar functions (Table 19 on page 251)
- Datetime scalar functions (Table 20 on page 252)
- Partitioning scalar functions (Table 21 on page 253)
- Numeric scalar functions (Table 22 on page 253)
- String scalar functions (Table 23 on page 254)
- Miscellaneous scalar functions (Table 24 on page 256)

Table 16. Supported functions

Function name	Schema	Description	
	Input parameters		**Returns**
“ABS or ABSVAL” on page 277	SYSIBM	This scalar function returns the absolute value of the argument.	
	Any expression that returns a built-in numeric data type.		Same data type and length as the argument.
“ABS or ABSVAL” on page 277	SYSFUN	This scalar function returns the absolute value of the argument.	
	SMALLINT		SMALLINT
	INTEGER		INTEGER
	BIGINT		BIGINT
	DOUBLE		DOUBLE
“ACOS” on page 278	SYSIBM	This scalar function returns the arccosine of the argument as an angle expressed in radians.	
	DOUBLE		DOUBLE
ALTOBJ	SYSPROC	This procedure (an SQL administrative routine) parses an input CREATE TABLE statement serving as the target data definition language (DDL) for an existing table that is to be altered.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
AM_BASE_RPT_RECOMS	SYSPROC	This table function (an SQL administrative routine) returns recommendations for activity reports used by the activity monitor.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
AM_BASE_RPTS	SYSPROC	This table function (an SQL administrative routine) returns activity reports used by the activity monitor.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
AM_DROP_TASK	SYSPROC	This procedure (an SQL administrative routine) deletes a monitoring task.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
	Input parameters	**Returns**
AM_GET_LOCK_CHN_TB	SYSPROC	This procedure (an SQL administrative routine) returns application lock chain data in tabular format.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
AM_GET_LOCK_CHNS	SYSPROC	This procedure (an SQL administrative routine) displays lock chains for a specified application using a formatted string.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
AM_GET_LOCK_RPT	SYSPROC	This procedure (an SQL administrative routine) displays lock details for an application.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
AM_GET_RPT	SYSPROC	This procedure (an SQL administrative routine) displays activity monitor data for a report.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
AM_SAVE_TASK	SYSPROC	This procedure (an SQL administrative routine) creates or modifies a monitoring task.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
APP	DB2EAS	This procedure (an SQL administrative routine) starts or stops a specified application on the application server for DB2.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
APPLICATION_ID	SYSFUN	This scalar function (an SQL administrative routine) returns the application ID of the current connection.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"ASCII" on page 279	SYSFUN	This scalar function returns the ASCII code value of the leftmost character of the argument as an integer.
	CHAR	INTEGER
	VARCHAR(4000)	INTEGER
	CLOB(1M)	INTEGER
"ASIN" on page 280	SYSIBM	This scalar function returns the arcsine of the argument as an angle expressed in radians.
	DOUBLE	DOUBLE
"ATAN" on page 281	SYSIBM	This scalar function returns the arctangent of the argument as an angle expressed in radians.
	DOUBLE	DOUBLE
"ATANH" on page 283	SYSIBM	This scalar function returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.
	DOUBLE	DOUBLE
"ATAN2" on page 282	SYSIBM	This scalar function returns the arctangent of *x* and *y* coordinates — specified by the first and second arguments — respectively, as an angle expressed in radians.
	DOUBLE, DOUBLE	DOUBLE
"AVG" on page 258	SYSIBM	This aggregate function returns the average of a set of numbers.
	numeric-type[4]	*numeric-type*[1]

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
	Input parameters	**Returns**
"BIGINT" on page 284	SYSIBM	This scalar function returns a 64-bit integer representation of a number or a character string in the form of an integer constant.
	numeric-type	BIGINT
	VARCHAR	BIGINT
"BLOB" on page 286	SYSIBM	This scalar function casts from source type to BLOB, with optional length.
	string-type	BLOB
	string-type, INTEGER	BLOB
"CEILING or CEIL" on page 287	SYSIBM	This scalar function returns the smallest integer greater than or equal to the argument.
	SMALLINT	SMALLINT
	INTEGER	INTEGER
	BIGINT	BIGINT
	DOUBLE	DOUBLE
"CHAR" on page 288	SYSIBM	This scalar function returns a string representation of the source type.
	character-type	CHAR
	character-type, INTEGER	CHAR(*integer*)
	datetime-type	CHAR
	datetime-type, *keyword*[2]	CHAR
	SMALLINT	CHAR(6)
	INTEGER	CHAR(11)
	BIGINT	CHAR(20)
	DECIMAL	CHAR(2+*precision*)
	DECIMAL, VARCHAR	CHAR(2+*precision*)
"CHAR" on page 288	SYSFUN	This scalar function returns a character string representation of a floating-point number.
	DOUBLE	CHAR(24)
"CHR" on page 292	SYSFUN	This scalar function returns the character that has the ASCII code value specified by the argument. The value of the argument should be between 0 and 255; otherwise, the return value is null.
	INTEGER	CHAR(1)
"CLOB" on page 293	SYSIBM	This scalar function casts from source type to CLOB, with optional length.
	character-type	CLOB
	character-type, INTEGER	CLOB
"COALESCE" on page 294 [3]	SYSIBM	This scalar function returns the first non-null argument in the set of arguments.
	any-type, *any-union-compatible-type*, ...	*any-type*
"CONCAT" on page 295	SYSIBM	This scalar function returns the concatenation of two string arguments.
	string-type, *compatible-string-type*	*max-string-type*
"CORRELATION" on page 260	SYSIBM	This aggregate function returns the coefficient of correlation of a set of number pairs.
	numeric-type, *numeric-type*	DOUBLE
"COS" on page 296	SYSIBM	This scalar function returns the cosine of the argument, where the argument is an angle expressed in radians.
	DOUBLE	DOUBLE
"COSH" on page 297	SYSIBM	This scalar function returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.
	DOUBLE	DOUBLE

Table 16. Supported functions (continued)

Function name	Schema	Description	Returns
	Input parameters		Returns
"COT" on page 298	SYSIBM	This scalar function returns the cotangent of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
"COUNT" on page 261	SYSIBM	This aggregate function returns the number of rows or values in a set of rows or values.	
	any-builtin-type[4]		INTEGER
"COUNT_BIG" on page 262	SYSIBM	This aggregate function returns the number of rows or values in a set of rows or values. The result can be greater than the maximum value of INTEGER.	
	any-builtin-type[4]		DECIMAL(31,0)
"COVARIANCE" on page 264	SYSIBM	This aggregate function returns the covariance of a set of number pairs.	
	numeric-type, *numeric-type*		DOUBLE
"DATE" on page 299	SYSIBM	This scalar function returns a date from a single input value.	
	DATE		DATE
	TIMESTAMP		DATE
	DOUBLE		DATE
	VARCHAR		DATE
"DAY" on page 300	SYSIBM	This scalar function returns the day part of a value.	
	VARCHAR		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
"DAYNAME" on page 301	SYSFUN	This scalar function returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument, based on what the locale was when db2start was issued.	
	VARCHAR(26)		VARCHAR(100)
	DATE		VARCHAR(100)
	TIMESTAMP		VARCHAR(100)
"DAYOFWEEK" on page 302	SYSFUN	This scalar function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
"DAYOFWEEK_ISO" on page 303	SYSFUN	This scalar function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
"DAYOFYEAR" on page 304	SYSFUN	This scalar function returns the day of the year in the argument as an integer value in the range 1-366.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
"DAYS" on page 305	SYSIBM	This scalar function returns an integer representation of a date.	
	VARCHAR		INTEGER
	TIMESTAMP		INTEGER
	DATE		INTEGER

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
DB_PARTITIONS	SYSPROC	This table function (an SQL administrative routine) returns the contents of the db2nodes.cfg file in table form.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"DBCLOB" on page 306	SYSIBM	This scalar function casts from source type to DBCLOB, with optional length.
	graphic-type	DBCLOB
	graphic-type, INTEGER	DBCLOB
"DBPARTITIONNUM" on page 307 [3]	SYSIBM	This scalar function returns the database partition number of the row. The argument is a column name within a table.
	any-type	INTEGER
"DECIMAL" on page 309	SYSIBM	This scalar function returns the decimal representation of a number, with optional precision and scale.
	numeric-type	DECIMAL
	numeric-type, INTEGER	DECIMAL
	numeric-type INTEGER, INTEGER	DECIMAL
"DECIMAL" on page 309	SYSIBM	This scalar function returns the decimal representation of a character string, with optional precision, scale, and decimal character.
	VARCHAR	DECIMAL
	VARCHAR, INTEGER	DECIMAL
	VARCHAR, INTEGER, INTEGER	DECIMAL
	VARCHAR, INTEGER, INTEGER, VARCHAR	DECIMAL
"DECRYPT_BIN and DECRYPT_CHAR" on page 313	SYSIBM	This scalar function returns a value that is the result of decrypting encrypted data using a password string.
	VARCHAR FOR BIT DATA	VARCHAR FOR BIT DATA
	VARCHAR FOR BIT DATA, VARCHAR	VARCHAR FOR BIT DATA
"DECRYPT_BIN and DECRYPT_CHAR" on page 313	SYSIBM	This scalar function returns a value that is the result of decrypting encrypted data using a password string.
	VARCHAR FOR BIT DATA	VARCHAR
	VARCHAR FOR BIT DATA, VARCHAR	VARCHAR
"DEGREES" on page 315	SYSFUN	This scalar function returns the number of degrees converted from the argument expressed in radians.
	DOUBLE	DOUBLE
"DEREF" on page 316	SYSIBM	This scalar function returns an instance of the target type of the reference type argument.
	REF(*any-structured-type*) *with defined scope*	*any-structured-type (same as input target type)*
"DIFFERENCE" on page 317	SYSFUN	This scalar function returns the difference between the sounds of the words in the two argument strings, as determined by the SOUNDEX function. A value of 4 means the strings sound the same.
	VARCHAR(4000), VARCHAR(4000)	INTEGER
"DIGITS" on page 318	SYSIBM	This scalar function returns the character string representation of a number.
	DECIMAL	CHAR
"DLCOMMENT" on page 319	SYSIBM	This scalar function returns the comment attribute of a DATALINK value.
	DATALINK	VARCHAR(254)
"DLLINKTYPE" on page 320	SYSIBM	This scalar function returns the link type attribute of a DATALINK value.
	DATALINK	VARCHAR(4)

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
	Input parameters	**Returns**
"DLNEWCOPY" on page 321	SYSIBM	This scalar function returns a DATALINK value that has an attribute indicating that the referenced file has changed.
	VARCHAR(254)	DATALINK
"DLPREVIOUSCOPY" on page 323	SYSIBM	This scalar function returns a DATALINK value that has an attribute indicating that the previous version of the file should be restored.
	VARCHAR(254)	DATALINK
"DLREPLACECONTENT" on page 325	SYSIBM	This scalar function returns a DATALINK value. When the function is on the right hand side of a SET clause in an UPDATE statement, or is in a VALUES clause in an INSERT statement, the assignment of the returned value results in replacing the content of a file by another file and then creating a link to it.
	VARCHAR(200)	DATALINK
	VARCHAR(200), VARCHAR(200)	DATALINK
"DLURLCOMPLETE" on page 327	SYSIBM	This scalar function returns the complete URL (including access token) from a DATALINK value.
	DATALINK	VARCHAR
"DLURLCOMPLETEONLY" on page 328	SYSIBM	This scalar function returns the complete URL (without the access token) from a DATALINK value with a link type of URL.
	DATALINK	VARCHAR(254)
"DLURLCOMPLETEWRITE" on page 329	SYSIBM	This scalar function returns the complete URL value necessary to modify a file specified from a DATALINK value with a link type of URL.
	DATALINK	VARCHAR(254)
"DLURLPATH" on page 330	SYSIBM	This scalar function returns the path and file name (including access token) of a DATALINK value.
	DATALINK	VARCHAR
"DLURLPATHONLY" on page 331	SYSIBM	This scalar function returns the path and file name (without any access token) of a DATALINK value.
	DATALINK	VARCHAR
"DLURLPATHWRITE" on page 332	SYSIBM	This scalar function returns the path and file name necessary to modify a file within a given server from a DATALINK value with a linktype of URL.
	DATALINK	VARCHAR(254)
"DLURLSCHEME" on page 333	SYSIBM	This scalar function returns the scheme from the URL attribute of a DATALINK value.
	DATALINK	VARCHAR
"DLURLSERVER" on page 334	SYSIBM	This scalar function returns the server from the URL attribute of a DATALINK value.
	DATALINK	VARCHAR
"DLVALUE" on page 335	SYSIBM	This scalar function builds a DATALINK value from a data-location argument, link type argument, and optional comment-string argument.
	VARCHAR	DATALINK
	VARCHAR, VARCHAR	DATALINK
	VARCHAR, VARCHAR, VARCHAR	DATALINK
"DOUBLE" on page 336	SYSIBM	This scalar function returns the floating-point representation of a number.
	numeric-type	DOUBLE
"DOUBLE" on page 336	SYSFUN	This scalar function returns the floating-point number corresponding to the character string representation of a number. Leading and trailing blanks in *argument* are ignored.
	VARCHAR	DOUBLE

Table 16. Supported functions (continued)

Function name	Schema	Description	
	Input parameters		Returns
"ENCRYPT" on page 338	SYSIBM	This scalar function returns a value that is the result of encrypting a data string expression.	
	VARCHAR		VARCHAR FOR BIT DATA
	VARCHAR, VARCHAR		VARCHAR FOR BIT DATA
	VARCHAR, VARCHAR, VARCHAR		VARCHAR FOR BIT DATA
"EVENT_MON_STATE" on page 340	SYSIBM	This scalar function returns the operational state of particular event monitor.	
	VARCHAR		INTEGER
EXEC_DB2_SCRIPT	SYSPROC	This procedure (an SQL administrative routine) executes DB2 commands at a DB2 instance.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
"EXP" on page 341	SYSFUN	This scalar function returns the exponential function of the argument.	
	DOUBLE		DOUBLE
"FLOAT" on page 342	SYSIBM	This scalar function is the same as DOUBLE.	
"FLOOR" on page 343	SYSIBM	This scalar function returns the largest integer that is less than or equal to the argument.	
	SMALLINT		SMALLINT
	INTEGER		INTEGER
	BIGINT		BIGINT
	DOUBLE		DOUBLE
"GENERATE_UNIQUE" on page 345	SYSIBM	This scalar function returns a bit data character string that is unique compared to any other execution of the same function.	
	no argument		CHAR(13) FOR BIT DATA
GET_DB_CONFIG	SYSFUN	This table function (an SQL administrative routine) returns database configuration information.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
GET_DBM_CONFIG	SYSFUN	This table function (an SQL administrative routine) returns database manager configuration information.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
GET_DBSIZE_INFO	SYSTOOLS	This procedure (an SQL administrative routine) calculates the database size and maximum capacity.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
GET_ROUTINE_OPTS	SYSPROC	This scalar function (an SQL administrative routine) returns a character string value of the options that are to be used for the creation of SQL procedures in the current session.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
GET_ROUTINE_SAR	SYSFUN	This procedure (an SQL administrative routine) returns the information necessary to install an identical routine on another database server running at least at the same level and operating system.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
"GETHINT" on page 344	SYSIBM	This scalar function returns the password hint if one is found.	
	VARCHAR or CLOB		VARCHAR

Table 16. Supported functions (continued)

	Schema	Description	
Function name	**Input parameters**		**Returns**
“GRAPHIC” on page 347	SYSIBM	This scalar function casts from source type to GRAPHIC, with optional length.	
	graphic-type		GRAPHIC
	graphic-type, INTEGER		GRAPHIC
“GROUPING” on page 265	SYSIBM	This aggregate function is used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.	
	any-type		SMALLINT
“HASHEDVALUE” on page 349 [3]	SYSIBM	This scalar function returns the partitioning map index (0 to 4095) of the row. The argument is a column name within a table.	
	any-type		INTEGER
HEALTH_DB_HIC	SYSPROC	This table function (an SQL administrative routine) returns collection health indicator information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_DB_HIC_HIS	SYSPROC	This table function (an SQL administrative routine) returns collection health indicator history information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_CONT_HI	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator information for containers from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_CONT_HI_HIS	SYSPROC	This table function (an SQL administrative routine returns a table with health indicator history information for containers from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_CONT_INFO	SYSPROC	This table function (an SQL administrative routine) returns a table with container information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_DB_HI	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_DB_HI_HIS	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator history information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_DB_INFO	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a health snapshot of a database.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
HEALTH_DBM_HI	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator information from a health snapshot of the DB2 database manager.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
HEALTH_DBM_HI_HIS	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator history information from a health snapshot of the DB2 database manager.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
HEALTH_DBM_INFO	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a health snapshot of the DB2 database manager.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
HEALTH_HI_REC	SYSPROC	This procedure (an SQL administrative routine) retrieves a set of recommendations that address a health indicator in alert state on a particular DB2 object.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
HEALTH_TBS_HI	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator information for table spaces from a health snapshot of a database.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
HEALTH_TBS_HI_HIS	SYSPROC	This table function (an SQL administrative routine) returns a table with health indicator history information for table spaces from a health snapshot of a database.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
HEALTH_TBS_INFO	SYSPROC	This table function (an SQL administrative routine) returns a table with table space information from a health snapshot of a database.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"HEX" on page 351	SYSIBM	This scalar function returns the hexadecimal representation of a value.
	any-builtin-type	VARCHAR
"HOUR" on page 353	SYSIBM	This scalar function returns the hour part of a value.
	VARCHAR	INTEGER
	TIME	INTEGER
	TIMESTAMP	INTEGER
	DECIMAL	INTEGER
"IDENTITY_VAL_LOCAL" on page 354	SYSIBM	This scalar function returns the most recently assigned value for an identity column.
		DECIMAL
"INSERT" on page 359	SYSFUN	This scalar function returns a string, where *argument3* bytes have been deleted from *argument1* (beginning at *argument2*), and *argument4* has been inserted into *argument1* (beginning at *argument2*).
	VARCHAR(4000), INTEGER, INTEGER, VARCHAR(4000)	VARCHAR(4000)
	CLOB(1M), INTEGER, INTEGER, CLOB(1M)	CLOB(1M)
	BLOB(1M), INTEGER, INTEGER, BLOB(1M)	BLOB(1M)
INSTALLAPP	DB2EAS	This procedure (an SQL administrative routine) installs a specified application on the application server for DB2.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"INTEGER" on page 360	SYSIBM	This scalar function returns the integer representation of a number.
	numeric-type	INTEGER
	VARCHAR	INTEGER

Table 16. Supported functions (continued)

Function name	Schema	Description	
	Input parameters		Returns
"JULIAN_DAY" on page 362	SYSFUN	This scalar function returns an integer value representing the number of days from January 1, 4712 B.C. (the start of the Julian date calendar) to the date value specified in the *argument*.	
	VARCHAR(26)		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
"LCASE (SYSFUN schema)" on page 364	SYSFUN	This scalar function returns a string in which all the characters have been converted to lowercase characters. LCASE will only handle characters in the invariant set. Therefore, LCASE(UCASE(string)) will not necessarily return the same result as LCASE(string).	
	VARCHAR(4000)		VARCHAR(4000)
	CLOB(1M)		CLOB(1M)
"LCASE or LOWER" on page 363	SYSIBM	This scalar function returns a string in which all the characters have been converted to lowercase characters.	
	CHAR		CHAR
	VARCHAR		VARCHAR
"LEFT" on page 365	SYSFUN	This scalar function returns a string consisting of the leftmost *argument2* bytes in *argument1*.	
	VARCHAR(4000), INTEGER		VARCHAR(4000)
	CLOB(1M), INTEGER		CLOB(1M)
	BLOB(1M), INTEGER		BLOB(1M)
"LENGTH" on page 366	SYSIBM	This scalar function returns the length of the operand in bytes (except for double-byte string types, which return the length in characters).	
	any-builtin-type		INTEGER
"LN" on page 367	SYSFUN	This scalar function returns the natural logarithm of the argument (same as LOG).	
	DOUBLE		DOUBLE
"LOCATE" on page 368	SYSFUN	This scalar function returns the starting position of the first occurrence of *argument1* within *argument2*. If the optional third argument is specified, it indicates the character position in *argument2* at which the search is to begin. If *argument1* is not found within *argument2*, the value 0 is returned.	
	VARCHAR(4000), VARCHAR(4000)		INTEGER
	VARCHAR(4000), VARCHAR(4000), INTEGER		INTEGER
	CLOB(1M), CLOB(1M)		INTEGER
	CLOB(1M), CLOB(1M), INTEGER		INTEGER
	BLOB(1M), BLOB(1M)		INTEGER
	BLOB(1M), BLOB(1M), INTEGER		INTEGER
"LOG" on page 369	SYSFUN	This scalar function returns the natural logarithm of the argument (same as LN).	
	DOUBLE		DOUBLE
"LOG10" on page 370	SYSFUN	This scalar function returns the base 10 logarithm of the argument.	
	DOUBLE		DOUBLE
"LONG_VARCHAR" on page 371	SYSIBM	This scalar function returns a long string.	
	character-type		LONG VARCHAR
"LONG_VARGRAPHIC" on page 372	SYSIBM	This scalar function casts from source type to LONG VARGRAPHIC.	
	graphic-type		LONG VARGRAPHIC

Table 16. Supported functions (continued)

Function name	Schema	Description	
	Input parameters		Returns
“LTRIM (SYSFUN schema)” on page 374	SYSFUN	This scalar function returns the characters of the argument with leading blanks removed.	
	VARCHAR(4000)		VARCHAR(4000)
	CLOB(1M)		CLOB(1M)
“LTRIM” on page 373	SYSIBM	This scalar function returns the characters of the argument with leading blanks removed.	
	CHAR		VARCHAR
	VARCHAR		VARCHAR
	GRAPHIC		VARGRAPHIC
	VARGRAPHIC		VARGRAPHIC
“MAX” on page 267	SYSIBM	This aggregate function returns the maximum value in a set of values.	
	any-builtin-type 5		*same as input type*
“MICROSECOND” on page 375	SYSIBM	This scalar function returns the microsecond (time-unit) part of a value.	
	VARCHAR		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
“MIDNIGHT_SECONDS” on page 376	SYSFUN	This scalar function returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the *argument*.	
	VARCHAR(26)		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
“MIN” on page 269	SYSIBM	This aggregate function returns the minimum value in a set of values.	
	any-builtin-type 5		*same as input type*
“MINUTE” on page 377	SYSIBM	This scalar function returns the minute part of a value.	
	VARCHAR		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
“MOD” on page 378	SYSFUN	This scalar function returns the remainder (modulus) of *argument1* divided by *argument2*. The result is negative only if *argument1* is negative.	
	SMALLINT, SMALLINT		SMALLINT
	INTEGER, INTEGER		INTEGER
	BIGINT, BIGINT		BIGINT
“MONTH” on page 379	SYSIBM	This scalar function returns the month part of a value.	
	VARCHAR		INTEGER
	DATE		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
“MONTHNAME” on page 380	SYSFUN	This scalar function returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument that is a date or a timestamp, based on what the locale was when the database was started.	
	VARCHAR(26)		VARCHAR(100)
	DATE		VARCHAR(100)
	TIMESTAMP		VARCHAR(100)

Table 16. Supported functions (continued)

	Schema	Description	
Function name	**Input parameters**		**Returns**
MQPUBLISH	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) publishes data to an MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQREAD	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) returns a message from an MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQREADALL	DB2MQ, DB2MQ1C	This table function (an SQL administrative routine) returns a table with messages and message metadata from an MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQREADALLCLOB	DB2MQ	This table function (an SQL administrative routine) returns a table containing messages and message metadata from a specified MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQREADCLOB	DB2MQ	This scalar function (an SQL administrative routine) returns a message from a specified MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQRECEIVE	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) returns a message from an MQSeries location and removes the message from the associated queue.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQRECEIVEALL	DB2MQ, DB2MQ1C	This table function (an SQL administrative routine) returns a table containing the messages and message metadata from an MQSeries location and removes the messages from the associated queue.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQRECEIVEALLCLOB	DB2MQ	This table function (an SQL administrative routine) returns a table containing messages and message metadata from a specified MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQRECEIVECLOB	DB2MQ	This scalar function (an SQL administrative routine) returns a message from a specified MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQSEND	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) sends data to an MQSeries location.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQSUBSCRIBE	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) subscribes to MQSeries messages published on a specific topic.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
MQUNSUBSCRIBE	DB2MQ, DB2MQ1C	This scalar function (an SQL administrative routine) unsubscribes from MQSeries messages published on a specific topic.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
	Input parameters	**Returns**
"MULTIPLY_ALT" on page 381	SYSIBM	This scalar function returns the product of two arguments as a decimal value. This function is useful when the sum of the argument precisions is greater than 31.
	exact-numeric-type, *exact-numeric-type*	DECIMAL
"NULLIF" on page 383 [3]	SYSIBM	This scalar function returns NULL if the arguments are equal, or returns the first argument if they are not equal.
	any-type [5], *any-comparable-type*[5]	*any-type*
"POSSTR" on page 384	SYSIBM	This scalar function returns the position at which one string is contained in another.
	string-type, *compatible-string-type*	INTEGER
"POWER" on page 386	SYSFUN	This scalar function returns the value of *argument1* to the power of *argument2*.
	INTEGER, INTEGER	INTEGER
	BIGINT, BIGINT	BIGINT
	DOUBLE, INTEGER	DOUBLE
	DOUBLE, DOUBLE	DOUBLE
PUT_ROUTINE_SAR	SYSFUN	This procedure (an SQL administrative routine) passes the information necessary to create and define an SQL routine at the database server.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"QUARTER" on page 387	SYSFUN	This scalar function returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.
	VARCHAR(26)	INTEGER
	DATE	INTEGER
	TIMESTAMP	INTEGER
"RADIANS" on page 388	SYSFUN	This scalar function returns the number of radians converted from the argument, which is expressed in degrees.
	DOUBLE	DOUBLE
"RAISE_ERROR" on page 389 [3]	SYSIBM	This scalar function 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.
	VARCHAR, VARCHAR	*any-type* [6]
"RAND" on page 390	SYSFUN	This scalar function returns a random floating point value between 0 and 1, using the argument as an optional seed value.
	no argument required	DOUBLE
	INTEGER	DOUBLE
"REAL" on page 391	SYSIBM	This scalar function returns the single-precision floating-point representation of a number.
	numeric-type	REAL
REBIND_ROUTINE_PACKAGE	SYSPROC	This procedure (an SQL administrative routine) rebinds the package associated with an SQL procedure.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"REC2XML" on page 392	SYSIBM	This scalar function returns a string formatted with XML tags, containing column names and column data.
	DECIMAL, VARCHAR, VARCHAR, *any-type*[7]	VARCHAR
"Regression functions" on page 270	SYSIBM	The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.
	numeric-type, *numeric-type*	DOUBLE

Table 16. Supported functions (continued)

Function name	Schema	Description	Returns
	Input parameters		
"Regression functions" on page 270	SYSIBM	The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.	
	numeric-type, *numeric-type*		INTEGER
"Regression functions" on page 270	SYSIBM	The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_R2 aggregate function returns the coefficient of determination for the regression.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_SLOPE aggregate function returns the slope of the line.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.	
	numeric-type, *numeric-type*		DOUBLE
"Regression functions" on page 270	SYSIBM	The REGR_SYY aggregate function returns quantities used to compute diagnostic statistics.	
	numeric-type, *numeric-type*		DOUBLE
| REORGCHK_IX_STATS	SYSPROC	This procedure (an SQL administrative routine) checks index statistics to determine whether or not there is a need for reorganization.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
| REORGCHK_TB_STATS	SYSPROC	This procedure (an SQL administrative routine) checks table statistics to determine whether or not there is a need for reorganization.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
"REPEAT" on page 396	SYSFUN	This scalar function returns a character string composed of *argument1* repeated *argument2* times.	
	VARCHAR(4000), INTEGER		VARCHAR(4000)
	CLOB(1M), INTEGER		CLOB(1M)
	BLOB(1M), INTEGER		BLOB(1M)
"REPLACE" on page 397	SYSFUN	This scalar function replaces all occurrences of *argument2* in *argument1* with *argument3*.	
	VARCHAR(4000), VARCHAR(4000), VARCHAR(4000)		VARCHAR(4000)
	CLOB(1M), CLOB(1M), CLOB(1M)		CLOB(1M)
	BLOB(1M), BLOB(1M), BLOB(1M)		BLOB(1M)
"RIGHT" on page 398	SYSFUN	This scalar function returns a string consisting of the rightmost *argument2* bytes in *argument1*.	
	VARCHAR(4000), INTEGER		VARCHAR(4000)
	CLOB(1M), INTEGER		CLOB(1M)
	BLOB(1M), INTEGER		BLOB(1M)

Table 16. Supported functions (continued)

Function name	Schema	Description	Returns
	Input parameters		**Returns**
"ROUND" on page 399	SYSIBM	This scalar function returns the first argument rounded to *argument2* places right of the decimal point. If *argument2* is negative, *argument1* is rounded to the absolute value of *argument2* places to the left of the decimal point.	
	INTEGER, INTEGER		INTEGER
	BIGINT, INTEGER		BIGINT
	DOUBLE, INTEGER		DOUBLE
"RTRIM (SYSFUN schema)" on page 402	SYSFUN	This scalar function returns the characters of the argument with trailing blanks removed.	
	VARCHAR(4000)		VARCHAR(4000)
	CLOB(1M)		CLOB(1M)
"RTRIM" on page 401	SYSIBM	This scalar function returns the characters of the argument with trailing blanks removed.	
	CHAR		VARCHAR
	VARCHAR		VARCHAR
	GRAPHIC		VARGRAPHIC
	VARGRAPHIC		VARGRAPHIC
"SECOND" on page 403	SYSIBM	This scalar function returns the second (time unit) part of a value.	
	VARCHAR		INTEGER
	TIME		INTEGER
	TIMESTAMP		INTEGER
	DECIMAL		INTEGER
| SERVER	DB2EAS	This procedure (an SQL administrative routine) starts or stops the application server for DB2.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
| SET_ROUTINE_OPTS	SYSPROC	This procedure (an SQL administrative routine) sets the options that are to be used for the creation of SQL procedures in the current session.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
"SIGN" on page 404	SYSFUN	This scalar function returns an indicator of the sign of the argument. If the argument is less than zero, -1 is returned. If the argument equals zero, 0 is returned. If the argument is greater than zero, 1 is returned.	
	SMALLINT		SMALLINT
	INTEGER		INTEGER
	BIGINT		BIGINT
	DOUBLE		DOUBLE
"SIN" on page 405	SYSIBM	This scalar function returns the sine of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
"SINH" on page 406	SYSIBM	This scalar function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.	
	DOUBLE		DOUBLE
"SMALLINT" on page 407	SYSIBM	This scalar function returns the small integer representation of a number.	
	numeric-type		SMALLINT
	VARCHAR		SMALLINT

Table 16. Supported functions (continued)

Function name	Schema	Description
	Input parameters	Returns
SNAPSHOT_AGENT	SYSPROC	This table function (an SQL administrative routine) returns a table with information about agents from an application snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_APPL	SYSPROC	This table function (an SQL administrative routine) returns a table of general information from an application snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_APPL_INFO	SYSPROC	This table function (an SQL administrative routine) returns a table of general information from an application snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_BP	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a buffer pool snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_CONTAINER	SYSPROC	This table function (an SQL administrative routine) returns a table with container configuration information from a table space snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_DATABASE	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a database snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_DBM	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a snapshot of the DB2 database manager.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_DYN_SQL	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a dynamic SQL snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_FCM	SYSPROC	This table function (an SQL administrative routine) returns a table with database manager level information regarding the fast communication manager (FCM).
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_FCMNODE	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a snapshot of the fast communication manager in the database manager.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_FILEW	SYSPROC	This procedure (an SQL administrative routine) writes system snapshot data to a file located in the `tmp` subdirectory of the instance directory.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SNAPSHOT_LOCK	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a lock snapshot.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description	Returns
SNAPSHOT_LOCKWAIT	SYSPROC	This table function (an SQL administrative routine) returns a table with lock waits information from an application snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_QUIESCERS	SYSPROC	This table function (an SQL administrative routine) returns a table with information about quiescers from a table space snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_RANGES	SYSPROC	This table function (an SQL administrative routine) returns a table with information from a range snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_STATEMENT	SYSPROC	This table function (an SQL administrative routine) returns a table with information about statements from an application snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_SUBSECT	SYSPROC	This table function (an SQL administrative routine) returns a table with information about subsections of access plans from an application snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_SWITCHES	SYSPROC	This table function (an SQL administrative routine) returns a table with information about the database snapshot switch state.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_TABLE	SYSPROC	This table function (an SQL administrative routine) returns a table with activity information from a table snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_TBREORG	SYSPROC	This table function (an SQL administrative routine) returns a table with table reorganization information.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_TBS	SYSPROC	This table function (an SQL administrative routine) returns a table with activity information from a table space snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
SNAPSHOT_TBS_CFG	SYSPROC	This table function (an SQL administrative routine) returns a table with configuration information from a table space snapshot.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		
"SOUNDEX" on page 408	SYSFUN	This scalar function returns a 4-character code representing the sound of the words in the argument. This result can be compared with the sound of other strings.	
	VARCHAR(4000)		CHAR(4)
"SPACE" on page 409	SYSFUN	This scalar function returns a character string consisting of *argument1* blanks.	
	INTEGER		VARCHAR(4000)
SQLCACHE_SNAPSHOT	SYSFUN	This table function (an SQL administrative routine) returns a table with a snapshot of the DB2 dynamic SQL statement cache.	
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.		

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
"SQRT" on page 410	SYSFUN	This scalar function returns the square root of the argument.
	DOUBLE	DOUBLE
"STDDEV" on page 273	SYSIBM	This aggregate function returns the standard deviation of a set of numbers.
	DOUBLE	DOUBLE
"SUBSTR" on page 411	SYSIBM	This scalar function returns a substring of string *argument1*, starting at *argument2*. The substring is *argument3* characters long. If *argument3* is not specified, the remainder of the string is assumed.
	string-type, INTEGER	*string-type*
	string-type, INTEGER, INTEGER	*string-type*
"SUM" on page 274	SYSIBM	This aggregate function returns the sum of a set of numbers.
	numeric-type[4]	*max-numeric-type*[1]
SYSINSTALLOBJECTS	SYSPROC	This procedure (an SQL administrative routine) creates or drops the database objects that are required for a specific tool.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
SYSINSTALLROUTINES	SYSPROC	This procedure (an SQL administrative routine) creates new procedures and functions in the SYSPROC schema if the **db2updv8** utility was not run after database migration.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
"TABLE_NAME" on page 414	SYSIBM	This scalar function 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.
	VARCHAR	VARCHAR(128)
	VARCHAR, VARCHAR	VARCHAR(128)
"TABLE_SCHEMA" on page 415	SYSIBM	This scalar function 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.
	VARCHAR	VARCHAR(128)
	VARCHAR, VARCHAR	VARCHAR(128)
"TAN" on page 417	SYSIBM	This scalar function returns the tangent of the argument, where the argument is an angle expressed in radians.
	DOUBLE	DOUBLE
"TANH" on page 418	SYSIBM	This scalar function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.
	DOUBLE	DOUBLE
"TIME" on page 419	SYSIBM	This scalar function returns a time from a value.
	TIME	TIME
	TIMESTAMP	TIME
	VARCHAR	TIME
"TIMESTAMP" on page 420	SYSIBM	This scalar function returns a timestamp from a value or a pair of values.
	TIMESTAMP	TIMESTAMP
	VARCHAR	TIMESTAMP
	VARCHAR, VARCHAR	TIMESTAMP
	VARCHAR, TIME	TIMESTAMP
	DATE, VARCHAR	TIMESTAMP
	DATE, TIME	TIMESTAMP

Table 16. Supported functions (continued)

Function name	Schema	Description	Returns
	Input parameters		**Returns**
"TIMESTAMP_FORMAT" on page 421	SYSIBM	This scalar function returns a timestamp from a character string (*argument1*) that has been interpreted using a format template (*argument2*).	
	VARCHAR, VARCHAR		TIMESTAMP
"TIMESTAMP_ISO" on page 423	SYSFUN	This scalar function returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of CURRENT DATE for the date elements, and zero for the fractional time element.	
	DATE		TIMESTAMP
	TIME		TIMESTAMP
	TIMESTAMP		TIMESTAMP
	VARCHAR(26)		TIMESTAMP
"TIMESTAMPDIFF" on page 424	SYSFUN	This scalar function returns an estimated number of intervals of type *argument1*, based on the difference between two timestamps. The second argument is the result of subtracting two timestamp types and converting the result to CHAR. Valid interval types are: **1** Fractions of a second **2** Seconds **4** Minutes **8** Hours **16** Days **32** Weeks **64** Months **128** Quarters **256** Years	
	INTEGER, CHAR(22)		INTEGER
"TO_CHAR" on page 426	SYSIBM	This scalar function returns a character representation of a timestamp.	
	Same as VARCHAR_FORMAT.		Same as VARCHAR_FORMAT.
"TO_DATE" on page 427	SYSIBM	This scalar function returns a timestamp from a character string.	
	Same as TIMESTAMP_FORMAT.		Same as TIMESTAMP_FORMAT.
"TRANSLATE" on page 428	SYSIBM	This scalar function returns a string in which one or more characters may have been translated into other characters.	
	CHAR		CHAR
	VARCHAR		VARCHAR
	CHAR, VARCHAR, VARCHAR		CHAR
	VARCHAR, VARCHAR, VARCHAR		VARCHAR
	CHAR, VARCHAR, VARCHAR, VARCHAR		CHAR
	VARCHAR, VARCHAR, VARCHAR, VARCHAR		VARCHAR
	GRAPHIC, VARGRAPHIC, VARGRAPHIC		GRAPHIC
	VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		VARGRAPHIC
	GRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		GRAPHIC
	VARGRAPHIC, VARGRAPHIC, VARGRAPHIC, VARGRAPHIC		VARGRAPHIC

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
Function name	**Input parameters**	**Returns**
“TRUNCATE or TRUNC” on page 430	SYSIBM	This scalar function returns *argument1* truncated to *argument2* places right of the decimal point. If *argument2* is negative, *argument1* is truncated to the absolute value of *argument2* places to the left of the decimal point.
	INTEGER, INTEGER	INTEGER
	BIGINT, INTEGER	BIGINT
	DOUBLE, INTEGER	DOUBLE
“TYPE_ID” on page 431 [3]	SYSIBM	This scalar function returns the internal data type identifier of the dynamic data type of the argument. Note that the result of this function is not portable across databases.
	any-structured-type	INTEGER
“TYPE_NAME” on page 432 [3]	SYSIBM	This scalar function returns the unqualified name of the dynamic data type of the argument.
	any-structured-type	VARCHAR(18)
“TYPE_SCHEMA” on page 433 [3]	SYSIBM	This scalar function returns the schema name of the dynamic type of the argument.
	any-structured-type	VARCHAR(128)
“UCASE or UPPER” on page 434	SYSFUN	This scalar function returns a string in which all the characters have been converted to uppercase characters.
	VARCHAR	VARCHAR
“UCASE or UPPER” on page 434	SYSIBM	This scalar function returns a string in which all the characters have been converted to uppercase characters.
	CHAR	CHAR
	VARCHAR	VARCHAR
UNINSTALLAPP	DB2EAS	This procedure (an SQL administrative routine) uninstalls a specified application on the application server for DB2.
	A complete description of this SQL administrative routine can be found in the DB2 Information Center.	
“VALUE” on page 435 [3]	SYSIBM	This scalar function is the same as COALESCE.
“VARCHAR” on page 436	SYSIBM	This scalar function returns a VARCHAR representation of the first argument. If a second argument is present, it specifies the length of the result.
	character-type	VARCHAR
	character-type, INTEGER	VARCHAR
	datetime-type	VARCHAR
“VARCHAR_FORMAT” on page 438	SYSIBM	This scalar function returns a character representation of a timestamp (*argument1*), formatted according to a template (*argument2*).
	TIMESTAMP, VARCHAR	VARCHAR
	VARCHAR, VARCHAR	VARCHAR
“VARGRAPHIC” on page 440	SYSIBM	This scalar function returns a VARGRAPHIC representation of the first argument. If a second argument is present, it specifies the length of the result.
	graphic-type	VARGRAPHIC
	graphic-type, INTEGER	VARGRAPHIC
	VARCHAR	VARGRAPHIC
“VARIANCE” on page 275	SYSIBM	This aggregate function returns the variance of a set of numbers.
	DOUBLE	DOUBLE

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description / Returns
	Schema	**Description**
	Input parameters	**Returns**
"WEEK" on page 442	SYSFUN	This scalar function returns the week of the year in the argument as an integer value in the range of 1-54.
	VARCHAR(26)	INTEGER
	DATE	INTEGER
	TIMESTAMP	INTEGER
"WEEK_ISO" on page 443	SYSFUN	This scalar function returns the week of the year in the argument as an integer value in the range of 1-53. The first day of a week is Monday. Week 1 is the first week of the year to contain a Thursday.
	VARCHAR(26)	INTEGER
	DATE	INTEGER
	TIMESTAMP	INTEGER
"YEAR" on page 444	SYSIBM	This scalar function returns the year part of a value.
	VARCHAR	INTEGER
	DATE	INTEGER
	TIMESTAMP	INTEGER
	DECIMAL	INTEGER
"+"	SYSIBM	Adds two numeric operands.
	numeric-type, *numeric-type*	*max-numeric-type*
"+"	SYSIBM	Unary plus operator.
	numeric-type	*numeric-type*
"+"	SYSIBM	Datetime plus operator.
	DATE, DECIMAL(8,0)	DATE
	TIME, DECIMAL(6,0)	TIME
	TIMESTAMP, DECIMAL(20,6)	TIMESTAMP
	DECIMAL(8,0), DATE	DATE
	DECIMAL(6,0), TIME	TIME
	DECIMAL(20,6), TIMESTAMP	TIMESTAMP
	datetime-type, DOUBLE, *labeled-duration-code*	*datetime-type*
"–"	SYSIBM	Subtracts two numeric operands.
	numeric-type, *numeric-type*	*max-numeric-type*
"–"	SYSIBM	Unary minus operator.
	numeric-type	*numeric-type* 1

Table 16. Supported functions (continued)

Function name	Schema / Input parameters	Description	Returns
	Schema	**Description**	
Function name	**Input parameters**		**Returns**
"−"	SYSIBM	Datetime minus operator.	
	DATE, DATE		DECIMAL(8,0)
	TIME, TIME		DECIMAL(6,0)
	TIMESTAMP, TIMESTAMP		DECIMAL(20,6)
	DATE, VARCHAR		DECIMAL(8,0)
	TIME, VARCHAR		DECIMAL(6,0)
	TIMESTAMP, VARCHAR		DECIMAL(20,6)
	VARCHAR, DATE		DECIMAL(8,0)
	VARCHAR, TIME		DECIMAL(6,0)
	VARCHAR, TIMESTAMP		DECIMAL(20,6)
	DATE, DECIMAL(8,0)		DATE
	TIME, DECIMAL(6,0)		TIME
	TIMESTAMP, DECIMAL(20,6)		TIMESTAMP
	datetime-type, DOUBLE, *labeled-duration-code*		*datetime-type*
"*"	SYSIBM	Multiplies two numeric operands.	
	numeric-type, *numeric-type*		*max-numeric-type*
"/"	SYSIBM	Divides two numeric operands.	
	numeric-type, *numeric-type*		*max-numeric-type*
"‖"	SYSIBM	Same as CONCAT.	

Notes

- References to string data types that are not qualified by a length should be assumed to support the maximum length for the data type.
- References to a DECIMAL data type without precision and scale should be assumed to allow any supported precision and scale.

Table 16. Supported functions (continued)

Function name	Schema	Description	
	Input parameters		Returns

Key to Table

any-builtin-type	Any data type that is not a distinct type.
any-type	Any type defined to the database.
any-structured-type	Any user-defined structured type defined to the database.
any-comparable-type	Any type that is comparable with other argument types as defined in "Assignments and comparisons" on page 99 .
any-union-compatible-type	Any type that is compatible with other argument types as defined in "Rules for result data types" on page 113 .
character-type	Any of the character string types: CHAR, VARCHAR, LONG VARCHAR, CLOB.
compatible-string-type	A string type that comes from the same grouping as the other argument (for example, if one argument is a *character-type* the other must also be a *character-type*).
datetime-type	Any of the datetime types: DATE, TIME, TIMESTAMP.
exact-numeric-type	Any of the exact numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL.
graphic-type	Any of the double byte character string types: GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB.
labeled-duration-code	As a type this is a SMALLINT. If the function is invoked using the infix form of the plus or minus operator, labeled-durations as defined in "Expressions" can be used. For a source function that does not use the plus or minus operator character as the name, the following values must be used for the labeled-duration-code argument when invoking the function. **1** YEAR or YEARS **2** MONTH or MONTHS **3** DAY or DAYS **4** HOUR or HOURS **5** MINUTE or MINUTES **6** SECOND or SECONDS **7** MICROSECOND or MICROSECONDS
LOB-type	Any of the large object types: BLOB, CLOB, DBCLOB.
max-numeric-type	The maximum numeric type of the arguments where maximum is defined as the rightmost *numeric-type*.
max-string-type	The maximum string type of the arguments where maximum is defined as the rightmost *character-type* or *graphic-type*. If arguments are BLOB, the *max-string-type* is BLOB.
numeric-type	Any of the numeric types: SMALLINT, INTEGER, BIGINT, DECIMAL, REAL, DOUBLE.
string-type	Any type from *character-type*, *graphic-type* or BLOB.

Table 16. Supported functions (continued)

Function name	Schema	Description	
	Input parameters		Returns

Table Footnotes

[1] When the input parameter is SMALLINT, the result type is INTEGER. When the input parameter is REAL, the result type is DOUBLE.

[2] Keywords allowed are ISO, USA, EUR, JIS, and LOCAL. This function signature is not supported as a sourced function.

[3] This function cannot be used as a source function.

[4] The keyword ALL or DISTINCT may be used before the first parameter. If DISTINCT is specified, the use of user-defined structured types, long string types or a DATALINK type is not supported.

[5] The use of user-defined structured types, long string types or a DATALINK type is not supported.

[6] The type returned by RAISE_ERROR depends upon the context of its use. RAISE_ERROR, if not cast to a particular type, will return a type appropriate to its invocation within a CASE expression.

[7] The use of *graphic-type*, *LOB-type*, long string types, and DATALINK types is not supported.

Table 17. Aggregate functions

Function	Description
"AVG" on page 258	Returns the average of a set of numbers.
"CORRELATION" on page 260	Returns the coefficient of correlation of a set of number pairs.
"COUNT" on page 261	Returns the number of rows or values in a set of rows or values.
"COUNT_BIG" on page 262	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 264	Returns the covariance of a set of number pairs.
"GROUPING" on page 265	Used with grouping-sets and super-groups to indicate sub-total rows generated by a grouping set. The value returned is 0 or 1. A value of 1 means that the value of the argument in the returned row is a null value, and the row was generated for a grouping set. This generated row provides a sub-total for a grouping set.
"MAX" on page 267	Returns the maximum value in a set of values.
"MIN" on page 269	Returns the minimum value in a set of values.
"Regression functions" on page 270	The REGR_AVGX aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 270	The REGR_AVGY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 270	The REGR_COUNT aggregate function returns the number of non-null number pairs used to fit the regression line.
"Regression functions" on page 270	The REGR_INTERCEPT or REGR_ICPT aggregate function returns the y-intercept of the regression line.
"Regression functions" on page 270	The REGR_R2 aggregate function returns the coefficient of determination for the regression.
"Regression functions" on page 270	The REGR_SLOPE aggregate function returns the slope of the line.
"Regression functions" on page 270	The REGR_SXX aggregate function returns quantities used to compute diagnostic statistics.

Table 17. Aggregate functions (continued)

Function	Description
"Regression functions" on page 270	The REGR_SXY aggregate function returns quantities used to compute diagnostic statistics.
"Regression functions" on page 270	The REGR_SYY aggregate function returns quantities used to compute diagnostic statistics.
"STDDEV" on page 273	Returns the standard deviation of a set of numbers.
"SUM" on page 274	Returns the sum of a set of numbers.
"VARIANCE" on page 275	Returns the variance of a set of numbers.

Table 18. Cast scalar functions

Function	Description
"BIGINT" on page 284	Returns a 64-bit integer representation of a number or a character string in the form of an integer constant.
"BLOB" on page 286	Returns a BLOB representation of a string of any type.
"CHAR" on page 288	Returns a CHARACTER representation of a value.
"CLOB" on page 293	Returns a CLOB representation of a value.
"DATE" on page 299	Returns a DATE from a value.
"DBCLOB" on page 306	Returns a DBCLOB representation of a string.
"DECIMAL" on page 309	Returns a DECIMAL representation of a number.
"DOUBLE" on page 336	Returns a DOUBLE PRECISION representation of a number.
"FLOAT" on page 342	Returns a FLOAT representation of a number.
"GRAPHIC" on page 347	Returns a GRAPHIC representation of a string.
"INTEGER" on page 360	Returns an INTEGER representation of a number.
"LONG_VARCHAR" on page 371	Returns a LONG VARCHAR representation of a value.
"LONG_VARGRAPHIC" on page 372	Returns a LONG VARGRAPHIC representation of a value.
"REAL" on page 391	Returns a REAL representation of a number.
"SMALLINT" on page 407	Returns a SMALLINT representation of a number.
"TIME" on page 419	Returns a TIME from a value.
"TIMESTAMP" on page 420	Returns a TIMESTAMP from a value or a pair of values.
"VARCHAR" on page 436	Returns a VARCHAR representation of a value.
"VARGRAPHIC" on page 440	Returns a VARGRAPHIC representation of a value.

Table 19. Datalink scalar functions

Function	Description
"DLCOMMENT" on page 319	Returns the comment attribute of a DATALINK value.
"DLLINKTYPE" on page 320	Returns the link type attribute of a DATALINK value.
"DLNEWCOPY" on page 321	Returns a DATALINK value that has an attribute indicating that the referenced file has changed.
"DLPREVIOUSCOPY" on page 323	Returns a DATALINK value that has an attribute indicating that the previous version of the file should be restored.

Table 19. Datalink scalar functions (continued)

Function	Description
"DLREPLACECONTENT" on page 325	Returns a DATALINK value. When the function is on the right hand side of a SET clause in an UPDATE statement, or is in a VALUES clause in an INSERT statement, the assignment of the returned value results in replacing the content of a file by another file and then creating a link to it.
"DLURLCOMPLETE" on page 327	Returns the complete URL (including access token) from a DATALINK value.
"DLURLCOMPLETEONLY" on page 328	Returns the complete URL (without the access token) from a DATALINK value with a link type of URL.
"DLURLCOMPLETEWRITE" on page 329	Returns the complete URL value necessary to modify a file specified from a DATALINK value with a link type of URL.
"DLURLPATH" on page 330	Returns the path and file name (including access token) of a DATALINK value.
"DLURLPATHONLY" on page 331	Returns the path and file name (without any access token) of a DATALINK value.
"DLURLPATHWRITE" on page 332	Returns the path and file name necessary to modify a file within a given server from a DATALINK value with a linktype of URL.
"DLURLSCHEME" on page 333	Returns the scheme from the URL attribute of a DATALINK value.
"DLURLSERVER" on page 334	Returns the server from the URL attribute of a DATALINK value.
"DLVALUE" on page 335	Builds a DATALINK value from a data-location argument, link type argument, and optional comment-string argument.

Table 20. Datetime scalar functions

Function	Description
"DAY" on page 300	Returns the day part of a value.
"DAYNAME" on page 301	Returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument, based on what the locale was when db2start was issued.
"DAYOFWEEK" on page 302	Returns the day of the week from a value, where 1 is Sunday and 7 is Saturday.
"DAYOFWEEK_ISO" on page 303	Returns the day of the week from a value, where 1 is Monday and 7 is Sunday.
"DAYOFYEAR" on page 304	Returns the day of the year from a value.
"DAYS" on page 305	Returns an integer representation of a date.
"HOUR" on page 353	Returns the hour part of a value.
"JULIAN_DAY" on page 362	Returns an integer value representing the number of days from January 1, 4712 B.C. to the date specified in the argument.
"MICROSECOND" on page 375	Returns the microsecond part of a value.
"MIDNIGHT_SECONDS" on page 376	Returns an integer value representing the number of seconds between midnight and a specified time value.

Table 20. Datetime scalar functions (continued)

Function	Description
"MINUTE" on page 377	Returns the minute part of a value.
"MONTH" on page 379	Returns the month part of a value.
"MONTHNAME" on page 380	Returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument that is a date or a timestamp, based on what the locale was when the database was started.
"QUARTER" on page 387	Returns an integer that represents the quarter of the year in which a date resides.
"SECOND" on page 403	Returns the seconds part of a value.
"TIMESTAMP_FORMAT" on page 421	Returns a timestamp from a character string (*argument1*) that has been interpreted using a format template (*argument2*).
"TIMESTAMP_ISO" on page 423	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 424	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 426	Returns a CHARACTER representation of a timestamp.
"TO_DATE" on page 427	Returns a timestamp from a character string.
"VARCHAR_FORMAT" on page 438	Returns a CHARACTER representation of a timestamp (*argument1*), formatted according to a template (*argument2*).
"WEEK" on page 442	Returns the week of the year from a value, where the week starts with Sunday.
"WEEK_ISO" on page 443	Returns the week of the year from a value, where the week starts with Monday.
"YEAR" on page 444	Returns the year part of a value.

Table 21. Partitioning scalar functions

Function	Description
"DBPARTITIONNUM" on page 307	Returns the database partition number of the row. The argument is a column name within a table.
"HASHEDVALUE" on page 349	Returns the partitioning map index (0 to 4095) of the row. The argument is a column name within a table.

Table 22. Numeric scalar functions

Function	Description
"ABS or ABSVAL" on page 277	Returns the absolute value of a number.
"ACOS" on page 278	Returns the arc cosine of a number, in radians.
"ASIN" on page 280	Returns the arc sine of a number, in radians.
"ATAN" on page 281	Returns the arc tangent of a number, in radians.

Table 22. Numeric scalar functions (continued)

Function	Description
"ATANH" on page 283	Returns the hyperbolic arc tangent of a number, in radians.
"ATAN2" on page 282	Returns the arc tangent of x and y coordinates as an angle expressed in radians.
"CEILING or CEIL" on page 287	Returns the smallest integer value that is greater than or equal to a number.
"COS" on page 296	Returns the cosine of a number.
"COSH" on page 297	Returns the hyperbolic cosine of a number.
"COT" on page 298	Returns the cotangent of the argument, where the argument is an angle expressed in radians.
"DEGREES" on page 315	Returns the number of degrees of an angle.
"DIGITS" on page 318	Returns a character-string representation of the absolute value of a number.
"EXP" on page 341	Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument.
"FLOOR" on page 343	Returns the largest integer value that is less than or equal to a number.
"LN" on page 367	Returns the natural logarithm of a number.
"LOG" on page 369	Returns the natural logarithm of a number (same as LN).
"LOG10" on page 370	Returns the common logarithm (base 10) of a number.
"MOD" on page 378	Returns the remainder of the first argument divided by the second argument.
"MULTIPLY_ALT" on page 381	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.
"POWER" on page 386	Returns the result of raising the first argument to the power of the second argument.
"RADIANS" on page 388	Returns the number of radians for an argument that is expressed in degrees.
"RAND" on page 390	Returns a random number.
"ROUND" on page 399	Returns a numeric value that has been rounded to the specified number of decimal places.
"SIGN" on page 404	Returns the sign of a number.
"SIN" on page 405	Returns the sine of a number.
"SINH" on page 406	Returns the hyperbolic sine of a number.
"SQRT" on page 410	Returns the square root of a number.
"TAN" on page 417	Returns the tangent of a number.
"TANH" on page 418	Returns the hyperbolic tangent of a number.
"TRUNCATE or TRUNC" on page 430	Returns a number value that has been truncated at a specified number of decimal places.

Table 23. String scalar functions

Function	Description
"ASCII" on page 279	Returns the ASCII code value of the leftmost character of the argument as an integer.

Table 23. String scalar functions (continued)

Function	Description
"CHR" on page 292	Returns the character that has the ASCII code value specified by the argument.
"CONCAT" on page 295	Returns a string that is the concatenation of two strings.
"DECRYPT_BIN and DECRYPT_CHAR" on page 313	Returns a value that is the result of decrypting encrypted data using a password string.
"DECRYPT_BIN and DECRYPT_CHAR" on page 313	Returns a value that is the result of decrypting encrypted data using a password string.
"DIFFERENCE" on page 317	Returns the difference between the sounds of the words in two argument strings, as determined by the SOUNDEX function. A value of 4 means the strings sound the same.
"ENCRYPT" on page 338	Returns a value that is the result of encrypting a data string expression.
"GENERATE_UNIQUE" on page 345	Returns a bit data character string that is unique compared to any other execution of the same function.
"GETHINT" on page 344	Returns the password hint if one is found.
"INSERT" on page 359	Returns a string, where *argument3* bytes have been deleted from *argument1* (beginning at *argument2*), and *argument4* has been inserted into *argument1* (beginning at *argument2*).
"LCASE or LOWER" on page 363	Returns a string in which all the characters have been converted to lowercase characters.
"LEFT" on page 365	Returns the leftmost characters from a string.
"LOCATE" on page 368	Returns the starting position of one string within another string.
"LTRIM" on page 373	Removes blanks from the beginning of a string expression.
"POSSTR" on page 384	Returns the starting position of one string within another string.
"REPEAT" on page 396	Returns a character string composed of *argument1* repeated *argument2* times.
"REPLACE" on page 397	Replaces all occurrences of *argument2* in *argument1* with *argument3*.
"RIGHT" on page 398	Returns the rightmost characters from a string.
"RTRIM" on page 401	Removes blanks from the end of a string expression.
"SOUNDEX" on page 408	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 409	Returns a character string that consists of a specified number of blanks.
"SUBSTR" on page 411	Returns a substring of a string.
"TRANSLATE" on page 428	Returns a string in which one or more characters in a string are converted to other characters.
"UCASE or UPPER" on page 434	Returns a string in which all the characters have been converted to uppercase characters.

Table 24. Miscellaneous scalar functions

Function	Description
“COALESCE” on page 294	Returns the first argument that is not null.
“DEREF” on page 316	Returns an instance of the target type of the reference type argument.
“EVENT_MON_STATE” on page 340	Returns the operational state of particular event monitor.
“HEX” on page 351	Returns a hexadecimal representation of a value.
“IDENTITY_VAL_LOCAL” on page 354	Returns the most recently assigned value for an identity column.
“LENGTH” on page 366	Returns the length of a value.
“NULLIF” on page 383	Returns a null value if the arguments are equal; otherwise, it returns the value of the first argument.
“RAISE_ERROR” on page 389	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 392	Returns a string formatted with XML tags, containing column names and column data.
“TABLE_NAME” on page 414	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 415	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 431	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 432	Returns the unqualified name of the dynamic data type of the argument.
“TYPE_SCHEMA” on page 433	Returns the schema name of the dynamic data type of the argument.
“VALUE” on page 435	Returns the first argument that is not null.

Aggregate functions

The argument of a column function is a set of values derived from an expression. The expression can include columns, but cannot include a *scalar-fullselect* or another column function (SQLSTATE 42607). The scope of the set is a group or an intermediate result table.

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, then the column functions are not applied; the result of the query is the empty set; the SQLCODE is set to +100; and the SQLSTATE is set to ’02000’.

If a GROUP BY clause is *not* specified in a query, and the intermediate result of the FROM, WHERE, and HAVING clauses is the empty set, then the column functions are applied to the empty set.

For example, the result of the following SELECT statement is the number of distinct values of JOBCODE for employees in department D01:

```
SELECT COUNT(DISTINCT JOBCODE)
  FROM CORPDATA.EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

The keyword DISTINCT is not considered an argument of the function, but rather a specification of an operation that is performed before the function is applied. If DISTINCT is specified, duplicate values are eliminated. If ALL is implicitly or explicitly specified, duplicate values are not eliminated.

Expressions can be used in column functions. For example:

```
SELECT MAX(BONUS + 1000)
  INTO :TOP_SALESREP_BONUS
  FROM EMPLOYEE
  WHERE COMM > 5000
```

Column functions can be qualified with a schema name (for example, SYSIBM.COUNT(*)).

Related reference:
- “SQL queries” on page 449

AVG

```
          ┌─ALL──────┐
►►──AVG──(─┼──────────┼──expression──)──────────────────────────────►◄
          └─DISTINCT─┘
```

The schema is SYSIBM.

The AVG function returns the average of a set of numbers.

The argument values must be numbers (built-in types only) and their sum must be within the range of the data type of the result, except for a decimal result data type. For decimal results, their sum must be within the range supported by a decimal data type having a precision of 31 and a scale identical to the scale of the argument values. The result can be null.

The data type of the result is the same as the data type of the argument values, except that:

- The result is a large integer if the argument values are small integers.
- The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal with precision *p* and scale *s*, the precision of the result is 31 and the scale is 31-*p*+*s*.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

If the type of the result is integer, the fractional part of the average is lost.

Examples:

- Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) 'D11'.

  ```
  SELECT AVG(PRSTAFF)
    INTO :AVERAGE
    FROM PROJECT
    WHERE DEPTNO = 'D11'
  ```

  Results in AVERAGE being set to 4.25 (that is 17/4) when using the sample table.

- Using the PROJECT table, set the host variable ANY_CALC (decimal(5,2)) to the average of each unique staffing level value (PRSTAFF) of projects in department (DEPTNO) 'D11'.

  ```
  SELECT AVG(DISTINCT PRSTAFF)
    INTO :ANY_CALC
    FROM PROJECT
    WHERE DEPTNO = 'D11'
  ```

Results in ANY_CALC being set to 4.66 (that is 14/3) when using the sample table.

CORRELATION

```
►►──┬─CORRELATION─┬──(──expression1──,──expression2──)─────────────────────►◄
    └─CORR────────┘
```

The schema is SYSIBM.

The CORRELATION function returns the coefficient of correlation of a set of number pairs.

The argument values must be numbers.

The data type of the result is double-precision floating point. The result can be null. When not null, the result is between −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.

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

```
              ┌─ALL──────┐
►►──COUNT──(──┬┴──────────┴──expression──┬──)─────────────────────────►◄
              │ └─DISTINCT─┘              │
              └─*─────────────────────────┘
```

The schema is SYSIBM.

The COUNT function returns the number of rows or values in a set of rows or values.

The data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

The result of the function is a large integer. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set. A row that includes only NULL values is included in the count.

The argument of COUNT(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT(*expression*) or COUNT(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:
- Using the EMPLOYEE table, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is 'F'.
  ```
  SELECT COUNT(*)
    INTO :FEMALE
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```
  Results in FEMALE being set to 13 when using the sample table.
- Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.
  ```
  SELECT COUNT(DISTINCT WORKDEPT)
    INTO :FEMALE_IN_DEPT
    FROM EMPLOYEE
    WHERE SEX = 'F'
  ```
  Results in FEMALE_IN_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

COUNT_BIG

```
                       ┌─ALL──────┐
►►──COUNT_BIG──(──┬──┼──────────┼──expression──┬──)──────────────────►◄
                  │  └─DISTINCT─┘              │
                  └─*──────────────────────────┘
```

The schema is SYSIBM.

The COUNT_BIG function returns the number of rows or values in a set of rows or values. It is similar to COUNT except that the result can be greater than the maximum value of integer.

The resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

The result of the function is a decimal with precision 31 and scale 0. The result cannot be null.

The argument of COUNT_BIG(*) is a set of rows. The result is the number of rows in the set. A row that includes only NULL values is included in the count.

The argument of COUNT_BIG(DISTINCT *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null and duplicate values. The result is the number of different non-null values in the set.

The argument of COUNT_BIG(*expression*) or COUNT_BIG(ALL *expression*) is a set of values. The function is applied to the set of values derived from the argument values by the elimination of null values. The result is the number of non-null values in the set, including duplicates.

Examples:

- Refer to COUNT examples and substitute COUNT_BIG for occurrences of COUNT. The results are the same except for the data type of the result.
- Some applications may require the use of COUNT but need to support values larger than the largest integer. This can be achieved by use of sourced user-defined functions and setting the SQL path. The following series of statements shows how to create a sourced function to support COUNT(*) based on COUNT_BIG and returning a decimal value with a precision of 15. The SQL path is set such that the sourced function based on COUNT_BIG is used in subsequent statements such as the query shown.

  ```
  CREATE FUNCTION RICK.COUNT() RETURNS DECIMAL(15,0)
    SOURCE SYSIBM.COUNT_BIG();
  SET CURRENT FUNCTION PATH RICK, SYSTEM PATH;
  SELECT COUNT(*) FROM EMPLOYEE;
  ```

  Note how the sourced function is defined with no parameters to support COUNT(*). This only works if you name the function COUNT and do not qualify the function with the schema name when it is used. To get the same effect as COUNT(*) with a name other than COUNT, invoke the function with no parameters. Thus, if RICK.COUNT had been defined as RICK.MYCOUNT instead, the query would have to be written as follows:

  ```
  SELECT MYCOUNT() FROM EMPLOYEE;
  ```

If the count is taken on a specific column, the sourced function must specify the type of the column. The following statements created a sourced function that will take any CHAR column as a argument and use COUNT_BIG to perform the counting.

```
CREATE FUNCTION RICK.COUNT(CHAR()) RETURNS DOUBLE
  SOURCE SYSIBM.COUNT_BIG(CHAR());
SELECT COUNT(DISTINCT WORKDEPT) FROM EMPLOYEE;
```

COVARIANCE

```
►►──┬─COVARIANCE─┬──(──expression1──,──expression2──)──────────────────►◄
    └─COVAR──────┘
```

The schema is SYSIBM.

The COVARIANCE function returns the (population) covariance of a set of number pairs.

The argument values must be numbers.

The data type of the result is double-precision floating point. The result can be null.

The function is applied to the set of (*expression1*,*expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the covariance of the value pairs in the set. The result is equivalent to the following:

1. Let avgexp1 be the result of AVG(*expression1*) and let avgexp2 be the result of AVG(*expression2*).
2. The result of COVARIANCE(*expression1*, *expression2*) is AVG((*expression1* - avgexp1) * (*expression2* - avgexp2)

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable COVARNCE (double-precision floating point) to the covariance between salary and bonus for those employees in department (WORKDEPT) ’A00’.

  ```
  SELECT COVARIANCE(SALARY, BONUS)
    INTO :COVARNCE
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00'
  ```

  COVARNCE is set to approximately 1.68888888888889E+006 when using the sample table.

GROUPING

```
►►──GROUPING──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

Used in conjunction with grouping-sets and super-groups, the GROUPING function returns a value that indicates whether or not a row returned in a GROUP BY answer set is a row generated by a grouping set that excludes the column represented by *expression*.

The argument can be of any type, but must be an item of a GROUP BY clause.

The result of the function is a small integer. It is set to one of the following values:

1 The value of *expression* in the returned row is a null value, and the row was generated by the super-group. This generated row can be used to provide sub-total values for the GROUP BY expression.

0 The value is other than the above.

Example:

The following query:

```
   SELECT SALES_DATE, SALES_PERSON,
       SUM(SALES) AS UNITS_SOLD,
       GROUPING(SALES_DATE) AS DATE_GROUP,
       GROUPING(SALES_PERSON) AS SALES_GROUP
     FROM SALES
     GROUP BY CUBE (SALES_DATE, SALES_PERSON)
     ORDER BY SALES_DATE, SALES_PERSON
```

results in:

```
SALES_DATE SALES_PERSON    UNITS_SOLD  DATE_GROUP  SALES_GROUP
---------- --------------- ----------- ----------- -----------
12/31/1995 GOUNOT                    1           0           0
12/31/1995 LEE                       6           0           0
12/31/1995 LUCCHESSI                 1           0           0
12/31/1995 -                         8           0           1
03/29/1996 GOUNOT                   11           0           0
03/29/1996 LEE                      12           0           0
03/29/1996 LUCCHESSI                 4           0           0
03/29/1996 -                        27           0           1
03/30/1996 GOUNOT                   21           0           0
03/30/1996 LEE                      21           0           0
03/30/1996 LUCCHESSI                 4           0           0
03/30/1996 -                        46           0           1
03/31/1996 GOUNOT                    3           0           0
03/31/1996 LEE                      27           0           0
03/31/1996 LUCCHESSI                 1           0           0
03/31/1996 -                        31           0           1
04/01/1996 GOUNOT                   14           0           0
04/01/1996 LEE                      25           0           0
04/01/1996 LUCCHESSI                 4           0           0
04/01/1996 -                        43           0           1
-          GOUNOT                   50           1           0
-          LEE                      91           1           0
-          LUCCHESSI                14           1           0
-          -                       155           1           1
```

An application can recognize a SALES_DATE sub-total row by the fact that the value of DATE_GROUP is 0 and the value of SALES_GROUP is 1. A SALES_PERSON sub-total row can be recognized by the fact that the value of DATE_GROUP is 1 and the value of SALES_GROUP is 0. A grand total row can be recognized by the value 1 for both DATE_GROUP and SALES_GROUP.

Related reference:

- "Subselect" on page 450

MAX

```
           ┌─ALL──────┐
►►──MAX──(─┼──────────┼─expression──)──────────────────────────────────►◄
           └─DISTINCT─┘
```

The schema is SYSIBM.

The MAX function returns the maximum value in a set of values.

The argument values can be of any built-in type other than a long string or DATALINK.

The resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

The data type, length and code page of the result are the same as the data type, length and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable MAX_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY/12) value.

  ```
  SELECT MAX(SALARY) / 12
    INTO :MAX_SALARY
    FROM EMPLOYEE
  ```

  Results in MAX_SALARY being set to 4395.83 when using the sample table.
- Using the PROJECT table, set the host variable LAST_PROJ(char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

  ```
  SELECT MAX(PROJNAME)
    INTO :LAST_PROJ
    FROM PROJECT
  ```

  Results in LAST_PROJ being set to 'WELD LINE PLANNING' when using the sample table.
- Similar to the previous example, set the host variable LAST_PROJ (char(40)) to the project name that comes last in the collating sequence when a project name is concatenated with the host variable PROJSUPP. PROJSUPP is '_Support'; it has a char(8) data type.

  ```
  SELECT MAX(PROJNAME CONCAT PROJSUPP)
    INTO :LAST_PROJ
    FROM PROJECT
  ```

Results in LAST_PROJ being set to 'WELD LINE PLANNING_SUPPORT' when using the sample table.

MIN

```
           ┌─ALL──────┐
►►──MIN──(─┼──────────┼─expression──)──────────────────────────────────►◄
           └─DISTINCT─┘
```

The MIN function returns the minimum value in a set of values.

The argument values can be of any built-in type other than a long string or DATALINK.

The resulting data type of *expression* cannot be a LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

The data type, length, and code page of the result are the same as the data type, length, and code page of the argument values. The result is considered to be a derived value and can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values.

If this function is applied to an empty set, the result of the function is a null value. Otherwise, the result is the minimum value in the set.

The specification of DISTINCT has no effect on the result and therefore is not recommended. It is included for compatibility with other relational systems.

Examples:

- Using the EMPLOYEE table, set the host variable COMM_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

  ```
  SELECT MAX(COMM) - MIN(COMM)
    INTO :COMM_SPREAD
    FROM EMPLOYEE
    WHERE WORKDEPT = 'D11'
  ```

  Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.
- Using the PROJECT table, set the host variable (FIRST_FINISHED (char(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

  ```
  SELECT MIN(PRENDATE)
    INTO :FIRST_FINISHED
    FROM PROJECT
  ```

  Results in FIRST_FINISHED being set to '1982-09-15' when using the sample table.

Regression functions

```
>>-+-REGR_AVGX---------+--(--expression1--,--expression2--)----><
   +-REGR_AVGY---------+
   +-REGR_COUNT--------+
   +-+-REGR_INTERCEPT-+-+
   | '-REGR_ICPT------' |
   +-REGR_R2-----------+
   +-REGR_SLOPE--------+
   +-REGR_SXX----------+
   +-REGR_SXY----------+
   '-REGR_SYY----------'
```

The schema is SYSIBM.

The regression functions support the fitting of an ordinary-least-squares regression line of the form y = a * x + b to a set of number pairs. The first element of each pair (*expression1*) is interpreted as a value of the dependent variable (that is, a ″y value″). The second element of each pair (*expression2*) is interpreted as a value of the independent variable (that is, an ″x value″).

The REGR_COUNT function returns the number of non-null number pairs used to fit the regression line (see below).

The REGR_INTERCEPT (or REGR_ICPT) function returns the y-intercept of the regression line (″b″ in the above equation).

The REGR_R2 function returns the coefficient of determination (″R-squared″ or ″goodness-of-fit″) for the regression.

The REGR_SLOPE function returns the slope of the line (″a″ in the above equation).

The REGR_AVGX, REGR_AVGY, REGR_SXX, REGR_SXY, and REGR_SYY functions return quantities that can be used to compute various diagnostic statistics needed for the evaluation of the quality and statistical validity of the regression model (see below).

The argument values must be numbers.

The data type of the result of REGR_COUNT is integer. For the remaining functions, the data type of the result is double precision floating point. The result can be null. When not null, the result of REGR_R2 is between 0 and 1, and the result of both REGR_SXX and REGR_SYY is non-negative.

Each function is applied to the set of (*expression1*, *expression2*) pairs derived from the argument values by the elimination of all pairs for which either *expression1* or *expression2* is null.

If the set is not empty and VARIANCE(*expression2*) is positive, REGR_COUNT returns the number of non-null pairs in the set, and the remaining functions return results that are defined as follows:

```
REGR_SLOPE(expression1,expression2) =
COVARIANCE(expression1,expression2)/VARIANCE(expression2)
```

```
REGR_INTERCEPT(expression1, expression2) =
AVG(expression1) - REGR_SLOPE(expression1, expression2) * AVG(expression2)
```

```
REGR_R2(expression1, expression2) =
POWER(CORRELATION(expression1, expression2), 2) if VARIANCE(expression1)>0
REGR_R2(expression1, expression2) = 1 if VARIANCE(expression1)=0
```

```
REGR_AVGX(expression1, expression2) = AVG(expression2)
```

```
REGR_AVGY(expression1, expression2) = AVG(expression1)
```

```
REGR_SXX(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression2)
```

```
REGR_SYY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * VARIANCE(expression1)
```

```
REGR_SXY(expression1, expression2) =
REGR_COUNT(expression1, expression2) * COVARIANCE(expression1, expression2)
```

If the set is not empty and VARIANCE(*expression2*) is equal to zero, then the regression line either has infinite slope or is undefined. In this case, the functions REGR_SLOPE, REGR_INTERCEPT, and REGR_R2 each return a null value, and the remaining functions return values as defined above. If the set is empty, REGR_COUNT returns zero and the remaining functions return a null value.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

The regression functions are all computed simultaneously during a single pass through the data. In general, it is more efficient to use the regression functions to compute the statistics needed for a regression analysis than to perform the equivalent computations using ordinary column functions such as AVERAGE, VARIANCE, COVARIANCE, and so forth.

The usual diagnostic statistics that accompany a linear-regression analysis can be computed in terms of the above functions. For example:

Adjusted R2
: 1 - ((1 - REGR_R2) * ((REGR_COUNT - 1) / (REGR_COUNT - 2)))

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

```
           ┌─ALL──────┐
►►──STDDEV──(─┼──────────┼──expression──)────────────────────────────────►◄
           └─DISTINCT─┘
```

The schema is SYSIBM.

The STDDEV function returns the standard deviation of a set of numbers.

The argument values must be numbers.

The data type of the result is double-precision floating point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the standard deviation of the values in the set.

The order in which the values are aggregated is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable DEV (double-precision floating point) to the standard deviation of the salaries for those employees in department (WORKDEPT) 'A00'.

  ```
  SELECT STDDEV(SALARY)
    INTO :DEV
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00'
  ```

  Results in DEV being set to approximately 9938.00 when using the sample table.

SUM

```
          ┌─ALL──────┐
►►──SUM──(─┼──────────┼─expression──)──────────────────────────────────►◄
          └─DISTINCT─┘
```

The schema is SYSIBM.

The SUM function returns the sum of a set of numbers.

The argument values must be numbers (built-in types only) and their sum must be within the range of the data type of the result.

The data type of the result is the same as the data type of the argument values except that:
- The result is a large integer if the argument values are small integers.
- The result is double-precision floating point if the argument values are single-precision floating point.

If the data type of the argument values is decimal, the precision of the result is 31 and the scale is the same as the scale of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are also eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the sum of the values in the set.

Example:
- Using the EMPLOYEE table, set the host variable JOB_BONUS (decimal(9,2)) to the total bonus (BONUS) paid to clerks (JOB='CLERK').

  ```
  SELECT SUM(BONUS)
    INTO :JOB_BONUS
    FROM EMPLOYEE
    WHERE JOB = 'CLERK'
  ```

  Results in JOB_BONUS being set to 2800 when using the sample table.

VARIANCE

```
                        ┌─ALL──────┐
►►──┬─VARIANCE─┬──(─┼──────────┼─expression─)──────────────────────────►◄
    └─VAR──────┘    └─DISTINCT─┘
```

The schema is SYSIBM.

The VARIANCE function returns the variance of a set of numbers.

The argument values must be numbers.

The data type of the result is double-precision floating point. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, redundant duplicate values are eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the variance of the values in the set.

The order in which the values are added is undefined, but every intermediate result must be within the range of the result data type.

Example:

- Using the EMPLOYEE table, set the host variable VARNCE (double-precision floating point) to the variance of the salaries for those employees in department (WORKDEPT) ’A00’.

  ```
  SELECT VARIANCE(SALARY)
    INTO :VARNCE
    FROM EMPLOYEE
    WHERE WORKDEPT = 'A00'
  ```

  Results in VARNCE being set to approximately 98763888.88 when using the sample table.

Scalar functions

A scalar function can be used wherever an expression can be used. However, the restrictions that apply to the use of expressions and column functions also apply when an expression or column function is used within a scalar function. For example, the argument of a scalar function can be a column function only if a column function is allowed in the context in which the scalar function is used.

The restrictions on the use of column functions do not apply to scalar functions, because a scalar function is applied to a single value rather than to a set of values.

The result of the following SELECT statement has as many rows as there are employees in department D01:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D01'
```

Scalar functions can be qualified with a schema name (for example, SYSIBM.CHAR(123)).

In a Unicode database, all scalar functions that accept a character or graphic string will accept any string types for which conversion is supported.

ABS or ABSVAL

```
►►──┬─ABS────┬──(──expression──)──────────────────────────────────►◄
    └─ABSVAL─┘
```

The schema is SYSIBM.

This function was first available in FixPak 2 of Version 7.1. The SYSFUN version of the ABS (or ABSVAL) function continues to be available.

Returns the absolute value of the argument. The argument can be any built-in numeric data type.

The result has the same data type and length attribute as the argument. The result can be null; if the argument is null, the result is the null value. If the argument is the maximum negative value for SMALLINT, INTEGER or BIGINT, the result is an overflow error.

Example:

```
ABS(-51234)
```

returns an INTEGER with a value of 51234.

ACOS

```
►►—ACOS—(—expression—)——————————————————————————————————————————►◄
```

The schema is SYSIBM. (The SYSFUN version of the ACOS function continues to be available.)

Returns the arccosine of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

Example:

Assume that the host variable ACOSINE is a DECIMAL(10,9) host variable with a value of 0.070737202.

```
   SELECT ACOS(:ACOSINE)
     FROM SYSIBM.SYSDUMMY1
```

This statement returns the approximate value 1.49.

ASCII

```
►►──ASCII──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the ASCII code value of the leftmost character of the argument as an integer.

The argument can be of any built-in character string type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed. For a VARCHAR, the maximum length is 4 000 bytes, and for a CLOB, the maximum length is 1 048 576 bytes. LONG VARCHAR is converted to CLOB for processing by the function.

The result of the function is always INTEGER.

The result can be null; if the argument is null, the result is the null value.

ASIN

```
►►──ASIN──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the ASIN function continues to be available.)

Returns the arcsine on the argument as an angle expressed in radians.

The argument can be of any built-in numeric type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATAN

```
►►──ATAN──(──expression──)─────────────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the ATAN function continues to be available.)

Returns the arctangent of the argument as an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATAN2

```
►►──ATAN2──(──expression──,──expression──)─────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the ATAN2 function continues to be available.)

Returns the arctangent of x and y coordinates as an angle expressed in radians. The x and y coordinates are specified by the first and second arguments, respectively.

The first and the second arguments can be of any built-in numeric data type. Both are converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

ATANH

```
►►──ATANH──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

Returns the hyperbolic arctangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

BIGINT

```
►►──BIGINT──(──┬─numeric-expression───┬──)──────────────────────────────►◄
               ├─character-expression─┤
               └─datetime-expression──┘
```

The schema is SYSIBM.

The BIGINT function returns a 64-bit integer representation of a number, character string, date, time, or timestamp in the form of an integer constant. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

numeric-expression
: An expression that returns a value of any built-in numeric data type.

  If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a big integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

character-expression
: An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.

  If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a big integer column or variable.

datetime-expression
: An expression that is of one of the following data types:

  - DATE. The result is a BIGINT value representing the date as *yyyymmdd*.
  - TIME. The result is a BIGINT value representing the time as *hhmmss*.
  - TIMESTAMP. The result is a BIGINT value representing the timestamp as *yyyymmddhhmmss*. The microseconds portion of the timestamp value is not included in the result.

The result of the function is a big integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples:

- From ORDERS_HISTORY table, count the number of orders and return the result as a big integer value.

  ```
  SELECT BIGINT (COUNT_BIG(*))
    FROM ORDERS_HISTORY
  ```

- Using the EMPLOYEE table, select the EMPNO column in big integer form for further processing in the application.

  ```
  SELECT BIGINT (EMPNO) FROM EMPLOYEE
  ```

- Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-22-14.07.21.136421'.

  ```
  BIGINT(RECEIVED)
  ```

results in the value `19 881 222 140 721`.

- Assume that the column STARTTIME (time) has an internal value equivalent to '12:03:04'.

  ```
  BIGINT(STARTTIME)
  ```

  results in the value `120 304`.

BLOB

```
►►──BLOB──(──string-expression──┬─────────────┬──)──────────────────►◄
                                └─,──integer──┘
```

The schema is SYSIBM.

The BLOB function returns a BLOB representation of a string of any type.

string-expression
: A *string-expression* whose value can be a character string, graphic string, or a binary string.

integer
: An integer value specifying the length attribute of the resulting BLOB data type. If *integer* is not specified, the length attribute of the result is the same as the length of the input, except where the input is graphic. In this case, the length attribute of the result is twice the length of the input.

The result of the function is a BLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples

- Given a table with a BLOB column named TOPOGRAPHIC_MAP and a VARCHAR column named MAP_NAME, locate any maps that contain the string 'Pellow Island' and return a single binary string with the map name concatenated in front of the actual map.

```
SELECT BLOB(MAP_NAME  || ':  ') || TOPOGRAPHIC_MAP
  FROM ONTARIO_SERIES_4
  WHERE TOPOGRAPHIC_MAP LIKE BLOB('%Pellow Island%')
```

CEILING or CEIL

```
►►──┬─CEILING─┬──(──expression──)──────────────────────────────────────►◄
    └─CEIL────┘
```

The schema is SYSIBM. (The SYSFUN version of the CEILING or CEIL function continues to be available.)

Returns the smallest integer value greater than or equal to the argument.

The argument can be of any built-in numeric type. The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

CHAR

Character to Character:

```
►►──CHAR──(──character-expression──┬─────────────┬──)──────────────────►◄
                                    └─,──integer──┘
```

Datetime to Character:

```
►►──CHAR──(──datetime-expression──┬──────────────────┬──)──────────────►◄
                                   └─,──┬─ISO───┬────┘
                                        ├─USA───┤
                                        ├─EUR───┤
                                        ├─JIS───┤
                                        └─LOCAL─┘
```

Integer to Character:

```
►►──CHAR──(──integer-expression──)─────────────────────────────────────►◄
```

Decimal to Character:

```
►►──CHAR──(──decimal-expression──┬───────────────────────┬──)──────────►◄
                                  └─,──decimal-character──┘
```

Floating-point to Character:

```
►►──CHAR──(──floating-point-expression──┬───────────────────────┬──)───►◄
                                         └─,──decimal-character──┘
```

The schema is SYSIBM. The SYSFUN.CHAR(*floating-point-expression*) signature continues to be available. In this case, the decimal character is locale sensitive, and therefore returns either a period or a comma, depending on the locale of the database server.

The CHAR function returns a fixed-length character string representation of:
- A character string, if the first argument is any type of character string
- A datetime value, if the first argument is a date, time, or timestamp
- An integer number, if the first argument is a SMALLINT, INTEGER, or BIGINT
- A decimal number, if the first argument is a decimal number
- A double-precision floating-point number, if the first argument is a DOUBLE or REAL

The first argument must be of a built-in data type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Note: The CAST expression can also be used to return a string expression.

The result of the function is a fixed-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Character to Character

character-expression
An expression that returns a value that is of the CHAR, VARCHAR, LONG VARCHAR, or CLOB data type.

integer
The length attribute of the resulting fixed-length character string. The value must be between 0 and 254.

If the length of the character expression is less than the length attribute of the result, the result is padded with blanks up to the length of the result. If the length of the character expression is greater than the length attribute of the result, the result is truncated. A warning is returned (SQLSTATE 01004), unless the truncated characters were all blanks, and the character expression was not a long string (LONG VARCHAR or CLOB).

Datetime to Character

datetime-expression
An expression that is of one of the following three data types:

date The result is the character string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

time The result is the character string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

timestamp
The result is the character string representation of the timestamp. The length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the string is the code page of the database at the application server.

Integer to Character

integer-expression
An expression that returns a value that is of an integer data type (either SMALLINT, INTEGER, or BIGINT).

The result is the character string representation of the argument in the form of an SQL integer constant. The result consists of *n* characters, which represent the significant digits in the argument, and is preceded by a minus sign if the argument is negative. The result is left justified.

- If the first argument is a small integer, the length of the result is 6.
- If the first argument is a large integer, the length of the result is 11.
- If the first argument is a big integer, the length of the result is 20.

If the number of 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 string is the code page of the database at the application server.

Decimal to Character

decimal-expression
: An expression that returns a value that is a decimal data type. If a different precision and scale are required, the DECIMAL scalar function can be used first to make the change.

decimal-character
: Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (−), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is the fixed-length character string representation of the argument. The result includes a decimal character and *p* digits, where *p* is the precision of the *decimal-expression*, with a preceding minus sign if the argument is negative. The length of the result is 2 + *p*, where *p* is the precision of the *decimal-expression*. This means that a positive value will always include one trailing blank.

The code page of the string is the code page of the database at the application server.

Floating-point to Character

floating-point-expression
: An expression that returns a value that is a floating-point data type (DOUBLE or REAL).

decimal-character
: Specifies the single-byte character constant that is used to delimit the decimal digits in the result character string. The character constant cannot be a digit, the plus sign (+), the minus sign (−), or a blank (SQLSTATE 42815). The default is the period (.) character.

The result is the fixed-length character string representation of the argument in the form of a floating-point constant. The length of the result is 24. If the argument is negative, the first character of the result is a minus sign; otherwise, the first character is a digit. If the argument value is zero, the result is 0E0; otherwise, the result includes the smallest number of characters that can represent the value of the argument, such that the mantissa consists of a single digit other than zero followed by the *decimal-character* and a sequence of digits. If the number of characters in the result is less than 24, the result is padded on the right with blanks.

The code page of the string is the code page of the database at the application server.

Examples:

- Assume that the PRSTDATE column has an internal value equivalent to 1988-12-25. The following function returns the value '12/25/1988'.

```
CHAR(PRSTDATE, USA)
```

- Assume that the STARTING column has an internal value equivalent to 17:12:30, and that the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours). The following function returns the value '5:12 PM'.

  ```
  CHAR(STARTING, USA)
  ```

  The following function returns the value '10:12 PM'.

  ```
  CHAR(STARTING + :HOUR_DUR, USA)
  ```
- Assume that the RECEIVED column (TIMESTAMP) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns. The following function returns the value '1988-12-25-17.12.30.000000'.

  ```
  CHAR(RECEIVED)
  ```
- The LASTNAME column is defined as VARCHAR(15). The following function returns the values in this column as fixed-length character strings that are 10 characters long. LASTNAME values that are more than 10 characters long (excluding trailing blanks) are truncated and a warning is returned.

  ```
  SELECT CHAR(LASTNAME,10) FROM EMPLOYEE
  ```
- The EDLEVEL column is defined as SMALLINT. The following function returns the values in this column as fixed-length character strings. An EDLEVEL value of 18 is returned as the CHAR(6) value '18 ' ('18' followed by four blanks).

  ```
  SELECT CHAR(EDLEVEL) FROM EMPLOYEE
  ```
- The SALARY column is defined as DECIMAL with a precision of 9 and a scale of 2. The current value (18357.50) is to be displayed with a comma as the decimal character (18357,50). The following function returns the value '00018357,50'.

  ```
  CHAR(SALARY, ',')
  ```
- Values in the SALARY column are to be subtracted from 20000.25 and displayed with the default decimal character. The following function returns the value '-0001642.75'.

  ```
  CHAR(20000.25 - SALARY)
  ```
- Assume that the host variable SEASONS_TICKETS is defined as INTEGER and has a value of 10000. The following function returns the value '10000.00 '.

  ```
  CHAR(DECIMAL(:SEASONS_TICKETS,7,2))
  ```
- Assume that the host variable DOUBLE_NUM is defined as DOUBLE and has a value of -987.654321E-35. The following function returns the value '-9.87654321E-33 '. Because the result data type is CHAR(24), there are nine trailing blanks in the result.

  ```
  CHAR(:DOUBLE_NUM)
  ```

Related reference:

- "Expressions" on page 168
- "VARCHAR" on page 436

CHR

```
►►──CHR──(──expression──)────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the character that has the ASCII code value specified by the argument.

The argument can be either INTEGER or SMALLINT. The value of the argument should be between 0 and 255; otherwise, the return value is null.

The result of the function is CHAR(1). The result can be null; if the argument is null, the result is the null value.

CLOB

```
►►──CLOB──(──character-string-expression──┬──────────────┬──)──────────────────►◄
                                          └─,──integer───┘
```

The schema is SYSIBM.

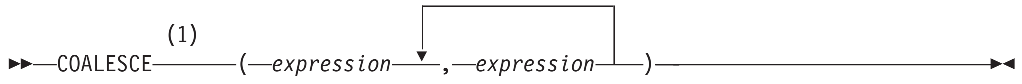
The CLOB function returns a CLOB representation of a character string type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

character-string-expression
: An *expression* that returns a value that is a character string.

integer
: An integer value specifying the length attribute of the resulting CLOB data type. The value must be between 0 and 2 147 483 647. If *integer* is not specified, the length of the result is the same as the length of the first argument.

The result of the function is a CLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

COALESCE



Notes:

1 VALUE is a synonym for COALESCE.

The schema is SYSIBM.

COALESCE returns the first argument that is not null.

The arguments are evaluated in the order in which they are specified, and the result of the function is the first argument that is not null. The result can be null only if all the arguments can be null, and the result is null only if all the arguments are null. The selected argument is converted, if necessary, to the attributes of the result.

The arguments must be compatible. They can be of either a built-in or user-defined data type. (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.)

Examples:

- When selecting all the values from all the rows in the DEPARTMENT table, if the department manager (MGRNO) is missing (that is, null), then return a value of ’ABSENT’.

  ```
  SELECT DEPTNO, DEPTNAME, COALESCE(MGRNO, 'ABSENT'), ADMRDEPT
    FROM DEPARTMENT
  ```
- When selecting the employee number (EMPNO) and salary (SALARY) from all the rows in the EMPLOYEE table, if the salary is missing (that is, null), then return a value of zero.

  ```
  SELECT EMPNO, COALESCE(SALARY, 0)
    FROM EMPLOYEE
  ```

Related reference:

- “Rules for result data types” on page 113

CONCAT

```
           (1)
►►──CONCAT──────(──expression1──,──expression2──)──────────────────────►◄
```

Notes:

1 || may be used as a synonym for CONCAT.

The schema is SYSIBM.

Returns the concatenation of two string arguments. The two arguments must be compatible types.

The result of the function is a string. Its length is the sum of the lengths of the two arguments. If either argument can be null, the result can be null; if the argument is null, the result is the null value.

Related reference:

- "Expressions" on page 168

COS

```
►►──COS──(──expression──)─────────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the COS function continues to be available.)

Returns the cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

COSH

```
►►──COSH──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

COT

```
▶▶─COT─(─expression─)──────────────────────────────────────────▶◀
```

The schema is SYSIBM. (The SYSFUN version of the COT function continues to be available.)

Returns the cotangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

DATE

```
>>—DATE—(—expression—)——————————————————————————————————————><
```

The schema is SYSIBM.

The DATE function returns a date from a value.

The argument must be a date, timestamp, a positive number less than or equal to 3 652 059, a valid string representation of a date or timestamp, or a string of length 7 that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.

Only Unicode databases support an argument that is a graphic string representation of a date or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

If the argument is a string of length 7, it must represent a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366, denoting a day of that year.

The result of the function is a date. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the date part of the value.
- If the argument is a number:
  - The result is the date that is *n*-1 days after January 1, 0001, where *n* is the integral part of the number.
- If the argument is a string with a length of 7:
  - The result is the date represented by the string.

Examples:

Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-25-17.12.30.000000'.
- This example results in an internal representation of '1988-12-25'.
  ```
  DATE(RECEIVED)
  ```
- This example results in an internal representation of '1988-12-25'.
  ```
  DATE('1988-12-25')
  ```
- This example results in an internal representation of '1988-12-25'.
  ```
  DATE('25.12.1988')
  ```
- This example results in an internal representation of '0001-02-04'.
  ```
  DATE(35)
  ```

DAY

```
►►──DAY──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

The DAY function returns the day part of a value.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:
  - The result is the day part of the value, which is an integer between −99 and 99. A nonzero result has the same sign as the argument.

Examples:
- Using the PROJECT table, set the host variable END_DAY (smallint) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

  ```
  SELECT DAY(PRENDATE)
    INTO :END_DAY
    FROM PROJECT
    WHERE PROJNAME = 'WELD LINE PLANNING'
  ```

  Results in END_DAY being set to 15 when using the sample table.
- Assume that the column DATE1 (date) has an internal value equivalent to 2000-03-15 and the column DATE2 (date) has an internal value equivalent to 1999-12-31.

  ```
  DAY(DATE1 - DATE2)
  ```

  Results in the value 15.

DAYNAME

```
▶▶──DAYNAME──(──expression──)──────────────────────────────────────────▶◀
```

The schema is SYSFUN.

Returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

DAYOFWEEK

```
►►──DAYOFWEEK──(──expression──)──────────────────────────────────────►◄
```

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYOFWEEK_ISO

```
►►──DAYOFWEEK_ISO──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYOFYEAR

```
►►──DAYOFYEAR──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the day of the year in the argument as an integer value in the range 1-366.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

DAYS

```
►►──DAYS──(──expression──)────────────────────────────────────────►◄
```

The schema is SYSIBM.

The DAYS function returns an integer representation of a date.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Examples:

- Using the PROJECT table, set the host variable EDUCATION_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

  ```
  SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
    INTO :EDUCATION_DAYS
    FROM PROJECT
    WHERE PROJNO = 'IF2000'
  ```

  Results in EDUCATION_DAYS being set to 396.
- Using the PROJECT table, set the host variable TOTAL_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

  ```
  SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
    INTO :TOTAL_DAYS
    FROM PROJECT
    WHERE DEPTNO = 'E21'
  ```

  Results in TOTAL_DAYS being set to 1584 when using the sample table.

DBCLOB

```
►►—DBCLOB—(—graphic-expression——————————)——————————————————►◄
                                └─,—integer─┘
```

The schema is SYSIBM.

The DBCLOB function returns a DBCLOB representation of a graphic string type.

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is either:

- Left as is, if the supplied argument is a character string
- Converted to the blank character (X'0020'), if the supplied argument is a graphic string

Do not rely on these behaviors, because they might change in a future release.

The result of the function is a DBCLOB. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

graphic-expression
: An expression that returns a value that is a graphic string.

integer
: An integer value specifying the length attribute of the resulting DBCLOB data type. The value must be between 0 and 1 073 741 823. If *integer* is not specified, the length of the result is the same as the length of the first argument.

DBPARTITIONNUM

```
►►──DBPARTITIONNUM──(──column-name──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The DBPARTITIONNUM function returns the partition number of the row. For example, if used in a SELECT clause, it returns the partition number for each row of the table that was used to form the result of the SELECT statement.

The partition number returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partition number given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partition number of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column in a table. The column can have any data type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.) If *column-name* references a column in a view, the expression in the view for the column must reference a column of the underlying base table, and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the partition number is returned by the DBPARTITIONNUM function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER and is never null. Since row-level information is returned, the results are the same, regardless of which column is specified for the table. If there is no `db2nodes.cfg` file, the result is 0.

The DBPARTITIONNUM function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with versions earlier than Version 8, the keyword NODENUMBER can be substituted for DBPARTITIONNUM.

Examples:

- Count the number of rows where the row for an EMPLOYEE is on a different partition from the employee's department description in DEPARTMENT.

  ```
  SELECT COUNT(*) FROM DEPARTMENT D, EMPLOYEE E
    WHERE D.DEPTNO=E.WORKDEPT
    AND DBPARTITIONNUM(E.LASTNAME) <> DBPARTITIONNUM(D.DEPTNO)
  ```

- Join the EMPLOYEE and DEPARTMENT tables where the rows of the two tables are on the same partition.

  ```
  SELECT * FROM DEPARTMENT D, EMPLOYEE E
    WHERE DBPARTITIONNUM(E.LASTNAME) = DBPARTITIONNUM(D.DEPTNO)
  ```

- Log the employee number and the projected partition number of the new row into a table called EMPINSERTLOG1 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG1
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AW NEWTABLE
FOR EACH ROW
INSERT INTO EMPINSERTLOG1
VALUES(NEWTABLE.EMPNO, DBPARTITIONNUM
(NEWTABLE.EMPNO))
```

Related reference:

- "CREATE VIEW statement" in the *SQL Reference, Volume 2*

DECIMAL

Numeric to Decimal:

```
►►──┬─DECIMAL─┬──(──numeric-expression──────────────────────────────►
    └─DEC─────┘

►──┬────────────────────────────────────────────┬──)───────────────►◄
   └─,──precision-integer──┬──────────────────┬─┘
                           └─,──scale-integer─┘
```

Character to Decimal:

```
►►──┬─DECIMAL─┬──(──character-expression─────────────────────────────►
    └─DEC─────┘

►──┬─────────────────────────────────────────────────────────────────┬──)───►◄
   └─,──precision-integer──┬───────────────────────────────────────┬─┘
                           └─,──scale-integer──┬─────────────────────┬─┘
                                               └─,──decimal-character─┘
```

Datetime to Decimal:

```
►►──┬─DECIMAL─┬──(──datetime-expression──────────────────────────────►
    └─DEC─────┘

►──┬────────────────────────────────────────────┬──)───────────────►◄
   └─,──precision-integer──┬──────────────────┬─┘
                           └─,──scale-integer─┘
```

The schema is SYSIBM.

The DECIMAL function returns a decimal representation of:
- A number
- A character string representation of a decimal number
- A character string representation of an integer number
- A character string representation of a floating-point number
- A datetime value if the argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a decimal number with precision *p* and scale *s*, where *p* and *s* are the second and third arguments, respectively. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Numeric to Decimal

numeric-expression
: An expression that returns a value of any numeric data type.

precision-integer
: An integer constant with a value in the range of 1 to 31.

The default for *precision-integer* depends on the data type of *numeric-expression*:
- 15 for floating-point and decimal
- 19 for big integer
- 11 for large integer
- 5 for small integer.

scale-integer
An integer constant in the range of 0 to the *precision-integer* value. The default is zero.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with precision *p* and scale *s*, where *p* and *s* are the second and third arguments, respectively. An error occurs if the number of significant decimal digits required to represent the whole part of the number is greater than *p*–*s*.

Character to Decimal

character-expression
An *expression* that returns a value that is a character string with a length not greater than the maximum length of a character constant (4 000 bytes). It cannot have a CLOB or LONG VARCHAR data type. Leading and trailing blanks are eliminated from the string. The resulting substring must conform to the rules for forming an SQL integer or decimal constant (SQLSTATE 22018).

The *character-expression* is converted to the database code page if required to match the code page of the constant *decimal-character*.

precision-integer
An integer constant with a value in the range 1 to 31 that specifies the precision of the result. If not specified, the default is 15.

scale-integer
An integer constant with a value in the range 0 to *precision-integer* that specifies the scale of the result. If not specified, the default is 0.

decimal-character
Specifies the single-byte character constant used to delimit the decimal digits in *character-expression* from the whole part of the number. The character cannot be a digit, plus (+), minus (–), or blank, and it can appear at most once in *character-expression* (SQLSTATE 42815).

The result is a decimal number with precision *p* and scale *s*, where *p* and *s* are the second and third arguments, respectively. Digits are truncated from the end of the decimal number if the number of digits to the right of the decimal character is greater than the scale. An error occurs if the number of significant digits to the left of the decimal character (the whole part of the number) in *character-expression* is greater than *p*–*s* (SQLSTATE 22003). The default decimal character is not valid in the substring if a different value for the *decimal-character* argument is specified (SQLSTATE 22018).

Datetime to Decimal

datetime-expression
An expression that is of one of the following data types:
- DATE. The result is a DECIMAL(8,0) value representing the date as *yyyymmdd*.

- TIME. The result is a DECIMAL(6,0) value representing the time as *hhmmss*.
- TIMESTAMP. The result is a DECIMAL(20,6) value representing the timestamp as *yyyymmddhhmmss.nnnnnn*.

This function allows the user to specify a precision, or a precision and a scale. However, a scale cannot be specified without specifying a precision. The default value for (precision,scale) is (8,0) for DATE, (6,0) for TIME, and (20,6) for TIMESTAMP.

The result is a decimal number with precision *p* and scale *s*, where *p* and *s* are the second and third arguments, respectively. Digits are truncated from the end if the number of digits to the right of the decimal character is greater than the scale. An error occurs if the number of significant digits to the left of the decimal character (the whole part of the number) in *datetime-expression* is greater than *p–s* (SQLSTATE 22003).

Examples:

- Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
  FROM EMPLOYEE
```

- Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be ″cast″ as decimal(8,0).

```
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM PROJECT
```

- Assume that updates to the SALARY column are input through a window as a character string using comma as a decimal character (for example, the user inputs 21400,50). Once validated by the application, it is assigned to the host variable newsalary which is defined as CHAR(10).

```
UPDATE STAFF
  SET SALARY = DECIMAL(:newsalary, 9, 2, ',')
  WHERE ID = :empid;
```

  The value of newsalary becomes 21400.50.

- Add the default decimal character (.) to a value.

```
DECIMAL('21400,50', 9, 2, '.')
```

  This fails because a period (.) is specified as the decimal character, but a comma (,) appears in the first argument as a delimiter.

- Assume that the column STARTING (time) has an internal value equivalent to ’12:10:00’.

```
DECIMAL(STARTING)
```

  results in the value `121 000`.

- Assume that the column RECEIVED (timestamp) has an internal value equivalent to ’1988-12-22-14.07.21.136421’.

```
DECIMAL(RECEIVED)
```

  results in the value `19 881 222 140 721.136421`.

- The following table shows the decimal result and resulting precision and scale for various datetime input values.

DECIMAL(arguments)	Precision and Scale	Result
DECIMAL(2000-03-21)	(8,0)	20000321
DECIMAL(2000-03-21, 10)	(10,0)	20000321
DECIMAL(2000-03-21, 12, 2)	(12,2)	20000321.00
DECIMAL(12:02:21)	(6,0)	120221
DECIMAL(12:02:21, 10)	(10,0)	120221
DECIMAL(12:02:21, 10, 2)	(10,2)	120221.00
DECIMAL(2000-03-21-12.02.21.123456)	(20, 6)	20000321120221.123456
DECIMAL(2000-03-21-12.02.21.123456, 23)	(23, 6)	20000321120221.123456
DECIMAL(2000-03-21-12.02.21.123456, 23, 4)	(23, 4)	20000321120221.1234

DECRYPT_BIN and DECRYPT_CHAR

```
►►──┬─DECRYPT_BIN──┬──(──encrypted-data──┬─────────────────────────────────┬──)────►◄
    └─DECRYPT_CHAR─┘                     └─,──password-string-expression─┘
```

The schema is SYSIBM.

The DECRYPT_BIN and DECRYPT_CHAR functions both return a value that is the result of decrypting *encrypted-data*. The password used for decryption is either the *password-string-expression* value or the ENCRYPTION PASSWORD value assigned by the SET ENCRYPTION PASSWORD statement. The DECRYPT_BIN and DECRYPT_CHAR functions can only decrypt values that are encrypted using the ENCRYPT function (SQLSTATE 428FE).

encrypted-data
: An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT 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 or decryption will result in an error (SQLSTATE 428FD). If the value of the password argument is null or not provided, the data will be encrypted using the ENCRYPTION PASSWORD value, which must have been set for the session (SQLSTATE 51039).

The result of the DECRYPT_BIN function is VARCHAR FOR BIT DATA. The result of the DECRYPT_CHAR function is VARCHAR. If the *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 the *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 the *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 that uses a code page different from the code page in which the data was encrypted, then expansion may 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: This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = 'Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832');
SELECT DECRYPT_CHAR(SSN)
  FROM EMP;
```

This returns the value ’289-46-8832’.

Example 2: This example explicitly passes the encryption password.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832','Ben123','');
SELECT DECRYPT(SSN,'Ben123')
  FROM EMP;
```

This example returns the value '289-46-8832'.

Related reference:

- "SET ENCRYPTION PASSWORD statement" in the *SQL Reference, Volume 2*
- "ENCRYPT" on page 338
- "GETHINT" on page 344

DEGREES

```
►►──DEGREES──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the number of degrees converted from the argument expressed in radians.

The argument can be of any built-in numeric type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

DEREF

```
▶▶──DEREF──(──expression──)──────────────────────────────────────────▶◀
```

The DEREF function returns an instance of the target type of the argument.

The argument can be any value with a reference data type that has a defined scope (SQLSTATE 428DT).

The static data type of the result is the target type of the argument. The dynamic data type of the result is a subtype of the target type of the argument. The result can be null. The result is the null value if *expression* is a null value or if *expression* is a reference that has no matching OID in the target table.

The result is an instance of the subtype of the target type of the reference. The result is determined by finding the row of the target table or target view of the reference that has an object identifier that matches the reference value. The type of this row determines the dynamic type of the result. Since the type of the result can be based on a row of a subtable or subview of the target table or target view, the authorization ID of the statement must have SELECT privilege on the target table and all of its subtables or the target view and all of its subviews (SQLSTATE 42501).

Examples:

Assume that EMPLOYEE is a table of type EMP, and that its object identifier column is named EMPID. Then the following query returns an object of type EMP (or one of its subtypes), for each row of the EMPLOYEE table (and its subtables). This query requires SELECT privilege on EMPLOYEE and all its subtables.

```
SELECT DEREF(EMPID) FROM EMPLOYEE
```

Related reference:

- "TYPE_NAME" on page 432

DIFFERENCE

```
►►──DIFFERENCE──(──expression──,──expression──)──────────────────────────────►◄
```

The schema is SYSFUN.

Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings. A value of 4 is the best possible sound match.

The arguments can be character strings that are either CHAR or VARCHAR up to 4 000 bytes. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:

```
VALUES (DIFFERENCE('CONSTRAINT','CONSTANT'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONSTANT')),
(DIFFERENCE('CONSTRAINT','CONTRITE'),SOUNDEX('CONSTRAINT'),
SOUNDEX('CONTRITE'))
```

This example returns the following.

```
 1          2    3
----------- ---- ----
4 C523 C523
2 C523 C536
```

In the first row, the words have the same result from SOUNDEX while in the second row the words have only some similarity.

Related reference:

- "SOUNDEX" on page 408

DIGITS

```
►►──DIGITS──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

The DIGITS function returns a character-string representation of a number.

The argument must be an expression that returns a value of type SMALLINT, INTEGER, BIGINT or DECIMAL.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result of the function is a fixed-length character string representing the absolute value of the argument without regard to its scale. The result does not include a sign or a decimal character. Instead, it consists exclusively of digits, including, if necessary, leading zeros to fill out the string. The length of the string is:

- 5 if the argument is a small integer
- 10 if the argument is a large integer
- 19 if the argument is a big integer
- *p* if the argument is a decimal number with a precision of *p*.

Examples:

- Assume that a table called TABLEX contains an INTEGER column called INTCOL containing 10-digit numbers. List all distinct four digit combinations of the first four digits contained in column INTCOL.

  ```
  SELECT DISTINCT SUBSTR(DIGITS(INTCOL),1,4)
    FROM TABLEX
  ```
- Assume that COLUMNX has the DECIMAL(6,2) data type, and that one of its values is -6.28. Then, for this value:

  ```
  DIGITS(COLUMNX)
  ```

  returns the value '000628'.

  The result is a string of length six (the precision of the column) with leading zeros padding the string out to this length. Neither sign nor decimal point appear in the result.

DLCOMMENT

```
►►──DLCOMMENT──(──datalink-expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The DLCOMMENT function returns the comment value, if it exists, from a DATALINK value.

The argument must be an expression that results in a value with data type of DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

- Prepare a statement to select the date, the description, and the comment (from the link in the ARTICLES column) from the HOCKEY_GOALS table. The rows to be selected are those for goals scored by either of the Richard brothers (Maurice or Henri).

  ```
  stmtvar = "SELECT DATE_OF_GOAL, DESCRIPTION, DLCOMMENT(ARTICLES)
               FROM HOCKEY_GOALS
               WHERE BY_PLAYER = 'Maurice Richard'
               OR BY_PLAYER = 'Henri Richard' ";
  EXEC SQL PREPARE HOCKEY_STMT FROM :stmtvar;
  ```

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','A comment')
  ```

  then the following function operating on that value:

  ```
  DLCOMMENT(COLA)
  ```

  will return the value:

  ```
  A comment
  ```

DLLINKTYPE

```
►►──DLLINKTYPE──(──datalink-expression──)─────────────────────────►◄
```

The schema is SYSIBM.

The DLLINKTYPE function returns the linktype value from a DATALINK value.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(4). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  then the following function operating on that value:

  ```
  DLLINKTYPE(COLA)
  ```

  will return the value:

  ```
  URL
  ```

DLNEWCOPY

```
►►──DLNEWCOPY──(──data-location──,──has-token──)──────────────────────►◄
```

The schema is SYSIBM.

The DLNEWCOPY function returns a DATALINK value which has an attribute indicating that the referenced file has changed. If such a value is assigned to a DATALINK column as a result of an SQL UPDATE statement, DB2 is notified that an update to the linked file has completed. If the DATALINK column is defined with RECOVERY YES, the new version of the linked file is archived asynchronously. If such a value is assigned to a DATALINK column as a result of an SQL INSERT statement, an error (SQLSTATE 428D1) is returned.

data-location
: A VARCHAR(200) expression that specifies a varying-length character string containing a complete URL value. The value may have been obtained earlier by a SELECT statement through the DLURLCOMPLETEWRITE function.

has-token
: An INTEGER value that indicates whether the data location contains a write token.

  0 The data location does not contain a write token.

  1 The data location contains a write token.

  An error occurs if the value is neither 0 nor 1 (SQLSTATE 42815), or the token embedded in the data location is not valid (SQLSTATE 428D1).

The result of the function is a DATALINK value without the write token. Neither *data-location* nor *has-token* can be null.

For a DATALINK column defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE, the write token must be in the data location to complete the SQL UPDATE statement (SQLSTATE 428D1). On the other hand, for WRITE PERMISSION ADMIN NOT REQUIRING TOKEN FOR UPDATE, the write token is not required, but is allowed in the data location.

For a DATALINK column defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE, the write token must be the same as the one used to open the specified file, if it was opened (SQLSTATE 428D1).

For any WRITE PERMISSION ADMIN column, even if the write token has expired, the token is still considered valid as long as the same token is used to open the specified file for write access.

In a case where no file update has taken place, or the DATALINK file is linked with other options, such as WRITE PERMISSION BLOCKED/FS or NO LINK CONTROL, this function will behave like DLVALUE.

Examples:

- Given a DATALINK value that was inserted into column COLA (defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE) in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

Use the scalar function DLURLCOMPLETEWRITE to fetch the value:

```
SELECT DLURLCOMPLETEWRITE(COLA)
  FROM TBLA
  WHERE ...
```

It returns:

```
HTTP://DLFS.ALMADEN.IBM.COM/x/y/****************;a.b
```

where **************** represents the write token.

Use the above value to locate and update the content of the file. Issue the following SQL UPDATE statement to indicate that the file has been successfully changed:

```
UPDATE TBLA
  SET COLA = DLNEWCOPY('http://dlfs.almaden.ibm.com/x/y/********
    ********;a.b', 1)
  WHERE ...
```

where **************** represents the same write token used to modify the file referenced by the URL value. Note that if COLA is defined with WRITE PERMISSION ADMIN NOT REQUIRING TOKEN FOR UPDATE, the write token is not required in the above example.

- The value of the second argument (*has-token*) can be substituted by the following CASE statement. Assume the URL value is contained in a variable named *url_file*. Issue the following SQL UPDATE statement to indicate that the file has been successfully changed:

```
EXEC SQL UPDATE TBLA
  SET COLA = DLNEWCOPY(:url_file,
    (CASE
      WHEN LENGTH(:url_file) = LENGTH(DLURLCOMPLETEONLY(COLA))
      THEN 0
      ELSE 1
     END))
  WHERE ...
```

DLPREVIOUSCOPY

```
►►──DLPREVIOUSCOPY──(──data-location──,──has-token──)──────────────────────►◄
```

The schema is SYSIBM.

The DLPREVIOUSCOPY function returns a DATALINK value which has an attribute indicating that the previous version of the file should be restored. If such a value is assigned to a DATALINK column as a result of an SQL UPDATE statement, it triggers DB2 to restore the linked file from the previously committed version. If such a value is assigned to a DATALINK column as a result of an SQL INSERT statement, an error (SQLSTATE 428D1) is returned.

data-location
A VARCHAR(200) expression that specifies a varying-length character string containing a complete URL value. The value may have been obtained earlier by a SELECT statement through the DLURLCOMPLETEWRITE function.

has-token
An INTEGER value that indicates whether the data location contains a write token.

0 The data location does not contain a write token.

1 The data location contains a write token.

An error occurs if the value is neither 0 nor 1 (SQLSTATE 42815), or the token embedded in the data location is not valid (SQLSTATE 428D1).

The result of the function is a DATALINK value without the write token. Neither *data-location* nor *has-token* can be null.

For a DATALINK column defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE, the write token must be in the data location to complete the SQL UPDATE statement (SQLSTATE 428D1). On the other hand, for WRITE PERMISSION ADMIN NOT REQUIRING TOKEN FOR UPDATE, the write token is not required, but is allowed in the data location.

For a DATALINK column defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE, the write token must be the same as the one used to open the specified file, if it was opened (SQLSTATE 428D1).

For any WRITE PERMISSION ADMIN column, even if the write token has expired, the token is still considered valid as long as the same token is used to open the specified file for write access.

Examples:

- Given a DATALINK value that was inserted into column COLA (defined with WRITE PERMISSION ADMIN REQUIRING TOKEN FOR UPDATE and RECOVERY YES) in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  Use the scalar function DLURLCOMPLETEWRITE to fetch the value:

  ```
  SELECT DLURLCOMPLETEWRITE(COLA)
    FROM TBLA
    WHERE ...
  ```

It returns:

```
HTTP://DLFS.ALMADEN.IBM.COM/x/y/****************;a.b
```

where **************** represents the write token.

Use the above value to locate and update the content of the file. Issue the following SQL UPDATE statement to back out the file changes and restore to the previous committed version:

```
UPDATE TBLA
  SET COLA = DLPREVIOUSCOPY('http://dlfs.almaden.ibm.com/x/y/********
    ********;a.b', 1)
  WHERE ...
```

where **************** represents the same write token used to modify the file referenced by the URL value. Note that if COLA is defined with WRITE PERMISSION ADMIN NOT REQUIRING TOKEN FOR UPDATE, the write token is not required in the above example.

- The value of the second argument (*has-token*) can be substituted by the following CASE statement. Assume the URL value is contained in a variable named *url_file*. Issue the following SQL UPDATE statement to back out the file changes and restore to the previous committed version:

```
EXEC SQL UPDATE TBLA
  SET COLA = DLPREVIOUSCOPY(:url_file,
    (CASE
      WHEN LENGTH(:url_file) = LENGTH(DLURLCOMPLETEONLY(COLA))
      THEN 0
      ELSE 1
     END))
  WHERE ...
```

DLREPLACECONTENT

```
►►──DLREPLACECONTENT──(──data-location-target──,──data-location-source──┬────────────────────┬──)──────►◄
                                                                         └─,──comment-string──┘
```

The schema is SYSIBM.

The DLREPLACECONTENT function returns a DATALINK value. When the function is on the right hand side of a SET clause in an UPDATE statement, or is in a VALUES clause in an INSERT statement, the assignment of the returned value results in replacing the content of a file by another file and then creating a link to it. The actual file replacement process is done during commit processing of the current transaction.

data-location-target
: A VARCHAR(200) expression that specifies a varying-length character string containing a complete URL value.

data-location-source
: A VARCHAR expression that specifies the data location of a file in URL format. As a result of an assignment in an UPDATE or an INSERT statement, this file is renamed to the name of the file that is pointed to by *data-location-target*; the ownership and permission attributes of the target file are retained.

  There is a restriction that *data-location-source* can only be one of the following:
  - A zero-length value
  - A NULL value
  - The value of *data-location-target* plus a suffix string. The suffix string can be up to 20 characters in length. The characters of the suffix string must belong to the URL character set. Moreover, the string cannot contain a "\" character under the UNC scheme, or the "/" character under other valid schemes (SQLSTATE 428D1).

comment-string
: An optional VARCHAR value that contains a comment or additional location information.

The result of the function is a DATALINK value. If any argument can be null, the result can be null; if *data-location-target* is null, the result is the null value.

If *data-location-source* is null, a zero-length string, or exactly the same as *data-location-target*, the effect of DLREPLACECONTENT is the same as DLVALUE.

Example:
- Replace the content of a linked file by another file. Given a DATALINK value that was inserted into column PICT_FILE in table TBLA using the following INSERT statement:

  ```
  EXEC SQL INSERT INTO TBLA (PICT_ID, PICT_FILE)
    VALUES(1000, DLVALUE('HTTP://HOSTA.COM/dlfs/image-data/pict1.gif'));
  ```

  Replace the content of this file with another file by issuing the following SQL UPDATE statement:

```
EXEC SQL UPDATE TBLA
  SET PICT_FILE =
    DLREPLACECONTENT('HTTP://HOSTA.COM/dlfs/image-data/pict1.gif',
                     'HTTP://HOSTA.COM/dlfs/image-data/pict1.gif.new')
  WHERE PICT_ID = 1000;
```

DLURLCOMPLETE

```
►►──DLURLCOMPLETE──(──datalink-expression──)─────────────────────────────►◄
```

The DLURLCOMPLETE function returns the data location attribute from a DATALINK value with a link type of URL. When *datalink-expression* is a DATALINK column defined with the attribute READ PERMISSION DB, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  the following function operating on that value:

  ```
  DLURLCOMPLETE(COLA)
  ```

  returns:

  ```
  HTTP://DLFS.ALMADEN.IBM.COM/x/y/****************;a.b
  ```

  where **************** represents the access token.

DLURLCOMPLETEONLY

```
►►──DLURLCOMPLETEONLY──(──datalink-expression──)──────────────────────►◄
```

The schema is SYSIBM.

The DLURLCOMPLETEONLY function returns the data location attribute from a DATALINK value with a link type of URL. The value returned *never* includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes a comment, the result is a zero length string.

Example:

- Given a DATALINK value that was inserted into a DATALINK column COLA (defined with READ PERMISSION DB) in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  the following function operating on that value:

  ```
  DLURLCOMPLETEONLY(COLA)
  ```

  returns:

  ```
  HTTP://DLFS.ALMADEN.IBM.COM/x/y/a.b
  ```

DLURLCOMPLETEWRITE

```
►►──DLURLCOMPLETEWRITE──(──datalink-expression──)─────────────────────────►◄
```

The schema is SYSIBM.

The DLURLCOMPLETEWRITE function returns the complete URL value from a DATALINK value with a link type of URL. If the DATALINK value produced from *datalink-expression* comes from a DATALINK column defined with WRITE PERMISSION ADMIN, a write token is included in the return value. The returned value can be used to locate and update the linked file.

If the DATALINK column is defined with another WRITE PERMISSION option (not ADMIN) or NO LINK CONTROL, DLURLCOMPLETEWRITE returns just the URL value without a write token. If the file reference is derived from a DATALINK column defined with WRITE PERMISSION FS, a token is not required to write to the file, because write permission is controlled by the file system; if the file reference is derived from a DATALINK column defined with WRITE PERMISSION BLOCKED, the file cannot be written to at all.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes a comment, the result is a zero length string.

Example:

- Given a DATALINK value that was inserted into a DATALINK column COLA (defined with WRITE PERMISSION ADMIN) in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  the following function operating on that value:

  ```
  DLURLCOMPLETEWRITE(COLA)
  ```

  returns:

  ```
  HTTP://DLFS.ALMADEN.IBM.COM/x/y/****************;a.b
  ```

  where **************** represents the write token. If COLA is not defined with WRITE PERMISSION ADMIN, the write token will not be present.

DLURLPATH

```
►►──DLURLPATH──(──datalink-expression──)──────────────────────────────►◄
```

The schema is SYSIBM.

The DLURLPATH function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. When *datalink-expression* is a DATALINK column defined with the attribute READ PERMISSION DB, the value includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  then the following function operating on that value:

  ```
  DLURLPATH(COLA)
  ```

  will return the value:

  ```
  /x/y/****************;a.b
  ```

  (where *************** represents the access token)

DLURLPATHONLY

```
►►──DLURLPATHONLY──(──datalink-expression──)──────────────────────────►◄
```

The schema is SYSIBM.

The DLURLPATHONLY function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. The value returned NEVER includes a file access token.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  then the following function operating on that value:

  ```
  DLURLPATHONLY(COLA)
  ```

  will return the value:

  ```
  /x/y/a.b
  ```

DLURLPATHWRITE

```
►►──DLURLPATHWRITE──(──datalink-expression──)─────────────────────────────►◄
```

The schema is SYSIBM.

The DLURLPATHWRITE function returns the path and file name necessary to access a file within a given server from a DATALINK value with a linktype of URL. The value returned includes a write token if the DATALINK value produced from *datalink_expression* comes from a DATALINK column defined with WRITE PERMISSION ADMIN.

If the DATALINK column is defined with other WRITE PERMISSION options (not ADMIN) or NO LINK CONTROL, DLURLPATHWRITE returns the path and file name without a write token. If the file reference is derived from a DATALINK column defined with WRITE PERMISSION FS, a token is not required to write to the file, because write permission is controlled by the file system; if the file reference is derived from a DATALINK column defined with WRITE PERMISSION BLOCKED, the file cannot be written to at all.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes a comment, the result is a zero length string.

Example:

- Given a DATALINK value that was inserted into a DATALINK column COLA (defined with WRITE PERMISSION ADMIN) in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  the following function operating on that value:

  ```
  DLURLPATHWRITE(COLA)
  ```

  returns:

  ```
  /x/y/****************;a.b
  ```

  where **************** represents the write token. If COLA is not defined with WRITE PERMISSION ADMIN, the write token will not be present.

DLURLSCHEME

```
►►──DLURLSCHEME──(──datalink-expression──)─────────────────────────────────►◄
```

The schema is SYSIBM.

The DLURLSCHEME function returns the scheme from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(20). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  then the following function operating on that value:

  ```
  DLURLSCHEME(COLA)
  ```

  will return the value:

  ```
  HTTP
  ```

DLURLSERVER

```
►►──DLURLSERVER──(──datalink-expression──)─────────────────────────────►◄
```

The schema is SYSIBM.

The DLURLSERVER function returns the file server from a DATALINK value with a linktype of URL. The value will always be in upper case.

The argument must be an expression that results in a value with data type DATALINK.

The result of the function is VARCHAR(254). If the argument can be null, the result can be null; if the argument is null, the result is the null value.

If the DATALINK value only includes the comment the result returned is a zero length string.

Example:

- Given a DATALINK value that was inserted into column COLA of a row in table TBLA using the scalar function:

  ```
  DLVALUE('http://dlfs.almaden.ibm.com/x/y/a.b','URL','a comment')
  ```

  then the following function operating on that value:

  ```
  DLURLSERVER(COLA)
  ```

  will return the value:

  ```
  DLFS.ALMADEN.IBM.COM
  ```

DLVALUE

```
►►──DLVALUE──(──data-location──┬──────────────────────────────────────────────┬──────►
                               └─,──linktype-string──┬──────────────────────┬─┘
                                                     └─,──comment-string────┘

►──)──────────────────────────────────────────────────────────────────────────────►◄
```

The schema is SYSIBM.

The DLVALUE function returns a DATALINK 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. When the function is on the right hand side of a SET clause in an UPDATE statement or is in a VALUES clause in an INSERT statement, it usually also creates a link to a file. However, if only a comment is specified (in which case the data-location is a zero-length string), the DATALINK value is created with empty linkage attributes so there is no file link.

data-location
: If the link type is URL, then this is an expression that yields a varying length character string containing a complete URL value.

linktype-string
: An optional VARCHAR expression that specifies the link type of the DATALINK value. The only valid value is 'URL' (SQLSTATE 428D1).

comment-string
: An optional VARCHAR(254) value that provides a comment or additional location information. The length of *data-location* plus *comment-string* must not exceed 200 bytes.

The result of the function is a DATALINK value. If any argument of the DLVALUE function can be null, the result can be null; If the *data-location* is null, the result is the null value.

When defining a DATALINK value using this function, consider the maximum length of the target of the value. For example, if a column is defined as DATALINK(200), then the maximum length of the *data-location* plus the *comment* is 200 bytes.

Example:

- Insert a row into the table. The URL values for the first two links are contained in the variables named url_article and url_snapshot. The variable named url_snapshot_comment contains a comment to accompany the snapshot link. There is, as yet, no link for the movie, only a comment in the variable named url_movie_comment.

```
EXEC SQL
  INSERT INTO HOCKEY_GOALS
    VALUES('Maurice Richard',
           'Montreal Canadien',
           '?',
           'Boston Bruins,
           '1952-04-24',
           'Winning goal in game 7 of Stanley Cup final',
           DLVALUE(:url_article),
           DLVALUE(:url_snapshot, 'URL', :url_snapshot_comment),
           DLVALUE('', 'URL', :url_movie_comment) );
```

DOUBLE

Numeric to Double:

```
►►──┬─DOUBLE───────────┬──(──numeric-expression──)──────────────────────►◄
    ├─FLOAT────────────┤
    └─DOUBLE_PRECISION─┘
```

Character String to Double:

```
►►──DOUBLE──(──string-expression──)─────────────────────────────────────►◄
```

The schema is SYSIBM. However, the schema for DOUBLE(*string-expression*) is SYSFUN.

The DOUBLE function returns a floating-point number corresponding to a:
- number if the argument is a numeric expression
- character string representation of a number if the argument is a string expression.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Numeric to Double

numeric-expression
: The argument is an expression that returns a value of any built-in numeric data type.

  The result of the function is a double-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

  The result is the same number that would occur if the argument were assigned to a double-precision floating-point column or variable.

Character String to Double

string-expression
: The argument can be of type CHAR or VARCHAR in the form of a numeric constant. Leading and trailing blanks in argument are ignored.

  The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

  The result is the same number that would occur if the string was considered a constant and assigned to a double-precision floating-point column or variable.

Example:

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. To eliminate the possibility of out-of-range results, DOUBLE is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, DOUBLE(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

ENCRYPT

```
►►──ENCRYPT──────────────────────────────────────────────────────────────────────►

►─(──data-string-expression──┬─────────────────────────────────────────────────────┬──)──────►◄
                             └─,──password-string-expression──┬─────────────────────┬─┘
                                                              └─,──hint-string-expression─┘
```

The schema is SYSIBM.

The ENCRYPT function returns a value that is the result of encrypting *data-string-expression*. The password used for encryption is either the *password-string-expression* value or the ENCRYPTION PASSWORD value (as assigned using the SET ENCRYPTION PASSWORD statement). In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

data-string-expression
: An expression that returns a CHAR or VARCHAR value 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 VARCHAR value with at least 6 bytes and no more than 127 bytes (SQLSTATE 428FC). The value represents the password used to encrypt the *data-string-expression*. If the value of the password argument is null or not provided, the data will be encrypted using the ENCRYPTION PASSWORD value, which must have been set for the session (SQLSTATE 51039).

hint-string-expression
: An expression that returns a CHAR or VARCHAR value up to 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.

The length attribute of the result is:

- When the optional hint parameter is specified, the length attribute of the non-encrypted data + 8 bytes + the number of bytes to the next 8 byte boundary + 32 bytes for the hint length.
- With no hint parameter, the length attribute of the non-encrypted data + 8 bytes + the number of bytes to 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.

Notice 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 used is RC2 block cipher with padding, the 128-bit secret key is derived from the password using a MD2 message digest.
- ***Encryption Passwords and Data:*** It is the user's responsibility to perform password management. Once the data is encrypted only the password used to encrypt it can be used to decrypt it (SQLSTATE 428FD). Be careful when using CHAR variables to set password values as they may be padded with blanks. The encrypted result may contain null terminator and other non-printable characters.
- ***Table Column Definition:*** When defining columns and types to contain encrypted data, always calculate the length attribute as follows.

  For encrypted data with no hint:

  Maximum length of the non-encrypted data + 8 bytes + the number of bytes to the next 8 byte boundary = encrypted data column length.

  For encrypted data with an embedded hint:

  Maximum length of the non-encrypted data + 8 bytes + the number of bytes to the next 8 byte boundary + 32 bytes for the hint length = encrypted data column length.

  Any assignment or cast to a length shorter than the suggested data length may result in failed decryption in the future and lost data. Blanks are valid encrypted data values that may be truncated when stored in a column that is too short.

  Some sample column length calculations:

```
Maximum length of non-encrypted data          6 bytes
8 bytes                                       8 bytes
Number of bytes to the next 8 byte boundary   2 bytes
                                            ---------
Encrypted data column length                 16 bytes

Maximum length of non-encrypted data         32 bytes
8 bytes                                       8 bytes
Number of bytes to the next 8 byte boundary   8 bytes
                                            ---------
Encrypted data column length                 48 bytes
```

- ***Administration of encrypted data:*** Encrypted data can only be decrypted on servers that support the decryption functions that correspond to the ENCRYPT function. Hence, replication of columns with encrypted data should only be done to servers that support the DECRYPT_BIN or DECRYPT_CHAR function.

Examples:

Example 1: This example uses the ENCRYPTION PASSWORD value to hold the encryption password.

```
SET ENCRYPTION PASSWORD = 'Ben123';
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832');
```

Example 2: This example explicitly passes the encryption password.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832','Ben123');
```

Example 3: The hint 'Ocean' is stored to help the user remember the encryption password of 'Pacific'.

```
INSERT INTO EMP(SSN) VALUES ENCRYPT('289-46-8832','Pacific','Ocean');
```

Related reference:

- "DECRYPT_BIN and DECRYPT_CHAR" on page 313
- "GETHINT" on page 344

EVENT_MON_STATE

```
►►—EVENT_MON_STATE—(—string-expression—)——————————————————————►◄
```

The schema is SYSIBM.

The EVENT_MON_STATE function returns the current state of an event monitor.

The argument is a string expression with a resulting type of CHAR or VARCHAR and a value that is the name of an event monitor. If the named event monitor does not exist in the SYSCAT.EVENTMONITORS catalog table, SQLSTATE 42704 will be returned. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result is an integer with one of the following values:

0	The event monitor is inactive.
1	The event monitor is active.

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

The following example selects all of the defined event monitors, and indicates whether each is active or inactive:

```
SELECT EVMONNAME,
  CASE
    WHEN EVENT_MON_STATE(EVMONNAME) = 0 THEN 'Inactive'
    WHEN EVENT_MON_STATE(EVMONNAME) = 1 THEN 'Active'
  END
  FROM SYSCAT.EVENTMONITORS
```

EXP

```
>>──EXP──(──expression──)──────────────────────────────────────────><
```

The schema is SYSFUN.

Returns the exponential function of the argument.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

FLOAT

```
►►──FLOAT──(──numeric-expression──)──────────────────────────────────►◄
```

The schema is SYSIBM.

The FLOAT function returns a floating-point representation of a number. FLOAT is a synonym for DOUBLE.

Related reference:

- "DOUBLE" on page 336

FLOOR

```
►►──FLOOR──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the FLOOR function continues to be available.)

Returns the largest integer value less than or equal to the argument.

The result of the function has the same data type and length attribute as the argument except that the scale is 0 if the argument is DECIMAL. For example, an argument with a data type of DECIMAL(5,5) returns DECIMAL(5,0).

The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

GETHINT

```
►►──GETHINT──(──encrypted-data──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The GETHINT function will return the password hint if one is found in the *encrypted-data*. A password hint is a phrase that will help data owners remember passwords; for example, ’Ocean’ as a hint to remember ’Pacific’. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

encrypted-data
: An expression that returns a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA value that is a complete, encrypted data string. The data string must have been encrypted using the ENCRYPT function (SQLSTATE 428FE).

The result of the function is VARCHAR(32). The result can be null; if the hint parameter was not added to the *encrypted-data* by the ENCRYPT function or the first argument is null, the result is the null value.

Example:

In this example the hint ’Ocean’ is stored to help the user remember the encryption password ’Pacific’.

```
INSERT INTO EMP (SSN) VALUES ENCRYPT('289-46-8832', 'Pacific','Ocean');
SELECT GETHINT(SSN)
  FROM EMP;
```

The value returned is ’Ocean’.

Related reference:

- “DECRYPT_BIN and DECRYPT_CHAR” on page 313
- “ENCRYPT” on page 338

GENERATE_UNIQUE

```
►►──GENERATE_UNIQUE──(──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

The GENERATE_UNIQUE function returns a bit data character string 13 bytes long (CHAR(13) FOR BIT DATA) that is unique compared to any other execution of the same function. (The system clock is used to generate the internal Universal Time, Coordinated (UTC) timestamp along with the partition number on which the function executes. Adjustments that move the actual system clock backward could result in duplicate values.) The function is defined as not-deterministic.

There are no arguments to this function (the empty parentheses must be specified).

The result of the function is a unique value that includes the internal form of the Universal Time, Coordinated (UTC) and the partition number where the function was processed. The result cannot be null.

The result of this function can be used to provide unique values in a table. Each successive value will be greater than the previous value, providing a sequence that can be used within a table. The value includes the partition number where the function executed so that a table partitioned across multiple partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

This function differs from using the special register CURRENT TIMESTAMP in that a unique value is generated for each row of a multiple row insert statement or an insert statement with a fullselect.

The timestamp value that is part of the result of this function can be determined using the TIMESTAMP scalar function with the result of GENERATE_UNIQUE as an argument.

Examples:

- Create a table that includes a column that is unique for each row. Populate this column using the GENERATE_UNIQUE function. Notice that the UNIQUE_ID column has "FOR BIT DATA" specified to identify the column as a bit data character string.

  ```
  CREATE TABLE EMP_UPDATE
    (UNIQUE_ID CHAR(13) FOR BIT DATA,
    EMPNO CHAR(6),
    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.

GRAPHIC

Graphic to Graphic:

```
►►──GRAPHIC──(──graphic-expression──┬────────────┬──)──────────────►◄
                                    └─,──integer─┘
```

Character to Graphic:

```
►►──GRAPHIC──(──character-expression──)────────────────────────────►◄
```

Datetime to Graphic:

```
►►──GRAPHIC──(──datetime-expression──┬────────────────┬──)─────────►◄
                                     └─,──┬─ISO───┬───┘
                                          ├─USA───┤
                                          ├─EUR───┤
                                          ├─JIS───┤
                                          └─LOCAL─┘
```

The schema is SYSIBM.

The GRAPHIC function returns a fixed-length graphic string representation of:
- A graphic string, if the first argument is any type of graphic string
- A datetime value (Unicode database only), if the first argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X’0020’). Do not rely on this behavior, because it might change in a future release.

The result of the function is a fixed-length graphic string (GRAPHIC data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Graphic to Graphic

graphic-expression
An expression that returns a value that is a graphic string.

integer
An integer value specifying the length attribute of the resulting GRAPHIC data type. The value must be between 1 and 127. If a value is not specified, the length attribute of the result is the same as the length attribute of the first argument.

Character to Graphic

character-expression
An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length is 16 336 bytes.

The length attribute of the result is equal to the length attribute of the argument.

Datetime to Graphic

datetime-expression
An expression that is of one of the following three data types:

date The result is the graphic string representation of the date in the format specified by the second argument. The length of the result is 10. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

time The result is the graphic string representation of the time in the format specified by the second argument. The length of the result is 8. An error is returned if the second argument is specified and is not a valid value (SQLSTATE 42703).

timestamp
The result is the graphic string representation of the timestamp. The length of the result is 26. The second argument is not applicable and must not be specified (SQLSTATE 42815).

The code page of the string is the code page of the database at the application server.

Related reference:
- “VARGRAPHIC” on page 440

HASHEDVALUE

```
►►──HASHEDVALUE──(──column-name──)────────────────────────────────────►◄
```

The schema is SYSIBM.

The HASHEDVALUE function returns the partitioning map index of the row obtained by applying the partitioning function on the partitioning key value of the row. For example, if used in a SELECT clause, it returns the partitioning map index for each row of the table that was used to form the result of the SELECT statement.

The partitioning map index returned on transition variables and tables is derived from the current transition values of the partitioning key columns. For example, in a before insert trigger, the function will return the projected partitioning map index given the current values of the new transition variables. However, the values of the partitioning key columns may be modified by a subsequent before insert trigger. Thus, the final partitioning map index of the row when it is inserted into the database may differ from the projected value.

The argument must be the qualified or unqualified name of a column in a table. The column can have any data type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any data type as an argument, it is not necessary to create additional signatures to support user-defined distinct types.) If *column-name* references a column of a view the expression in the view for the column must reference a column of the underlying base table and the view must be deletable. A nested or common table expression follows the same rules as a view.

The specific row (and table) for which the partitioning map index is returned by the HASHEDVALUE function is determined from the context of the SQL statement that uses the function.

The data type of the result is INTEGER in the range 0 to 4095. For a table with no partitioning key, the result is always 0. A null value is never returned. Since row-level information is returned, the results are the same, regardless of which column is specified for the table.

The HASHEDVALUE function cannot be used on replicated tables, within check constraints, or in the definition of generated columns (SQLSTATE 42881).

For compatibility with versions earlier than Version 8, the function name PARTITION can be substituted for HASHEDVALUE.

Example:

- List the employee numbers (EMPNO) from the EMPLOYEE table for all rows with a partitioning map index of 100.

  ```
  SELECT EMPNO FROM EMPLOYEE
    WHERE HASHEDVALUE(PHONENO) = 100
  ```

- Log the employee number and the projected partitioning map index of the new row into a table called EMPINSERTLOG2 for any insertion of employees by creating a before trigger on the table EMPLOYEE.

```
CREATE TRIGGER EMPINSLOGTRIG2
  BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AW NEWTABLE
  FOR EACH ROW
  INSERT INTO EMPINSERTLOG2
    VALUES(NEWTABLE.EMPNO, HASHEDVALUE(NEWTABLE.EMPNO))
```

Related reference:

- "CREATE VIEW statement" in the *SQL Reference, Volume 2*

HEX

```
►►──HEX──(──expression──)────────────────────────────────────────────►◄
```

The schema is SYSIBM.

The HEX function returns a hexadecimal representation of a value as a character string.

The argument can be an expression that is a value of any built-in data type with a maximum length of 16 336 bytes.

The result of the function is a character string. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The code page is the database code page.

The result is a string of hexadecimal digits. The first two represent the first byte of the argument, the next two represent the second byte of the argument, and so forth. If the argument is a datetime value or a numeric value the result is the hexadecimal representation of the internal form of the argument. The hexadecimal representation that is returned may be different depending on the application server where the function is executed. Cases where differences would be evident include:

- Character string arguments when the HEX function is performed on an ASCII client with an EBCDIC server or on an EBCDIC client with an ASCII server.
- Numeric arguments (in some cases) when the HEX function is performed where client and server systems have different byte orderings for numeric values.

The type and length of the result vary based on the type and length of character string arguments.

- Character string
  - Fixed length not greater than 127
    - Result is a character string of fixed length twice the defined length of the argument.
  - Fixed length greater than 127
    - Result is a character string of varying length twice the defined length of the argument.
  - Varying length
    - Result is a character string of varying length with maximum length twice the defined maximum length of the argument.
- Graphic string
  - Fixed length not greater than 63
    - Result is a character string of fixed length four times the defined length of the argument.
- Fixed length greater than 63
  - Result is a character string of varying length four times the defined length of the argument.
- Varying length

– Result is a character string of varying length with maximum length four times the defined maximum length of the argument.

Examples:

Assume the use of a DB2 for AIX application server for the following examples.

- Using the DEPARTMENT table set the host variable HEX_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the 'PLANNING' department (DEPTNAME).

  ```
  SELECT HEX(MGRNO)
    INTO :HEX_MGRNO
    FROM DEPARTMENT
    WHERE DEPTNAME = 'PLANNING'
  ```

  HEX_MGRNO will be set to '303030303230' when using the sample table (character value is '000020').
- Suppose COL_1 is a column with a data type of char(1) and a value of 'B'. The hexadecimal representation of the letter 'B' is X'42'. HEX(COL_1) returns a two-character string '42'.
- Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. An eight-character string '0004010C' is the result of applying the HEX function to the internal representation of the decimal value, 40.1.

HOUR

```
▶▶──HOUR──(──expression──)──────────────────────────────────────────▶◀
```

The schema is SYSIBM.

The HOUR function returns the hour part of a value.

The argument must be a time, timestamp, time duration, timestamp duration, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the hour part of the value, which is an integer between 0 and 24.
- If the argument is a time duration or timestamp duration:
  - The result is the hour part of the value, which is an integer between −99 and 99. A nonzero result has the same sign as the argument.

Example:

Using the CL_SCHED sample table, select all the classes that start in the afternoon.

```
SELECT * FROM CL_SCHED
  WHERE HOUR(STARTING) BETWEEN 12 AND 17
```

IDENTITY_VAL_LOCAL

```
►►──IDENTITY_VAL_LOCAL──(──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

The IDENTITY_VAL_LOCAL function is a non-deterministic function that returns the most recently assigned value for an identity column, where the assignment occurred as a result of a single row INSERT statement using a VALUES clause. The function has no input parameters.

The result is a DECIMAL(31,0), regardless of the actual data type of the corresponding identity column.

The value returned by the function is the value assigned to the identity column of the table identified in the most recent single row INSERT statement. 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 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 DB2.

The function returns a null value if a single row INSERT statement with a VALUES clause has not been issued at the current processing level against a table containing an identity column.

The result of the function is not affected by the following:
- A single row INSERT statement with a VALUES clause for a table without an identity column.
- A multiple row INSERT statement with a VALUES clause.
- An INSERT statement with a fullselect.
- A ROLLBACK TO SAVEPOINT statement.

Notes:
- Expressions in the VALUES clause of an INSERT statement are evaluated prior to the assignments for the target columns of the INSERT statement. 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 statement. 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 statement invoked in the same triggered action that had a VALUES clause for a table containing an identity column. (This applies to both FOR EACH ROW and FOR EACH STATEMENT after insert triggers.) If a single row INSERT statement with a VALUES clause for a table containing an identity column was not executed within the same triggered action, prior to the invocation of the IDENTITY_VAL_LOCAL function, then the function returns a null value.
- Since the results of the IDENTITY_VAL_LOCAL function are not deterministic, the result of an invocation of the IDENTITY_VAL_LOCAL 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 DB2. The assigned value could be a value specified in a SET transition variable statement, within the body of a before insert trigger, for a trigger transition variable associated with the identity column.
- The value returned by the function is unpredictable following a failed single row INSERT with a VALUES clause into a table with an identity column. The value may be the value that would have been returned from the function had it been invoked prior to the failed INSERT, or it may be the value that would have been assigned had the INSERT succeeded. The actual value returned depends on the point of failure and is therefore unpredictable.

Examples:

Example 1: Set the variable IVAR to the value assigned to the identity column in the EMPLOYEE table. If this insert is the first into the EMPLOYEE table, then IVAR would have a value of 1.

```
CREATE TABLE EMPLOYEE
  (EMPNO   INTEGER GENERATED ALWAYS AS IDENTITY,
   NAME    CHAR(30),
   SALARY  DECIMAL(5,2),
   DEPTNO  SMALLINT)
```

Example 2: An IDENTITY_VAL_LOCAL function invoked in an INSERT statement returns the value associated with the previous single row INSERT statement, with a VALUES clause for a table with an identity column. Assume for this example that there are two tables, T1 and T2. Both T1 and T2 have an identity column named C1. DB2 generates values in sequence, starting with 1, for the C1 column in table T1, and values in sequence, starting with 10, for the C1 column in table T2.

```
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
```

which gives a result of:

```
C1            C2
-----------   ----------
          1            5
          2            6
```

and now, declaring the function for the variable IVAR:

```
VALUES IDENTITY_VAL_LOCAL() INTO :IVAR
```

At this point, the IDENTITY_VAL_LOCAL function would return a value of 2 in IVAR, because that was the value most recently assigned by DB2. The following INSERT statement inserts a single row into T2, where column C2 gets a value of 2 from the IDENTITY_VAL_LOCAL function.

```
INSERT INTO T2 (C2) VALUES (IDENTITY_VAL_LOCAL())

SELECT * FROM T2
  WHERE C1 = DECIMAL(IDENTITY_VAL_LOCAL(),15,0)
```

returning a result of:

```
C1                  C2
-----------------   ----------
              10.            2
```

Invoking the IDENTITY_VAL_LOCAL function after this insert results in a value of 10, which is the value generated by DB2 for column C1 of T2.

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 statement inserts a row into the EMP_ACT table. This INSERT 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. This is because an INSERT statement has not been issued at this level of the nesting, and as such, if the IDENTITY_VAL_LOCAL function were invoked in the VALUES clause of the INSERT for EMP_ACT, then it would return a null value. This INSERT statement for the EMP_ACT table also results in the generation of a new identity column value for the ACNT_NUM column.

A second triggered INSERT statement inserts a row into the ACCT_LOG table. This 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 statement 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 EMPLOYEE table.

From the invoking application (that is, the level at which the INSERT to EMPLOYEE is issued), set the variable IVAR to the value assigned to the EMPNO column of the EMPLOYEE table by the original INSERT statement.

```
INSERT INTO EMPLOYEE (NAME, SALARY, DEPTNO)
  VALUES ('Rupert', 989.99, 50);
```

The contents of the three tables after processing the original INSERT statement and all of the triggered actions are:

```
   SELECT EMPNO, SUBSTR(NAME,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 assigned to the EMPNO column of the EMPLOYEE table. The following VALUES statement results in setting IVAR to 1000. The insert into the EMP_ACT table (which occurred after the insert into the EMPLOYEE table and at a lower nesting level) has no affect on what is returned by this invocation of the IDENTITY_VAL_LOCAL function.

```
   VALUES IDENTITY_VAL_LOCAL() INTO :IVAR;
```

Related samples:

- “fnuse.out -- HOW TO USE BUILT-IN SQL FUNCTIONS (C)”
- “fnuse.sqc -- How to use built-in SQL functions (C)”
- “fnuse.out -- HOW TO USE FUNCTIONS (C++)”
- “fnuse.sqC -- How to use built-in SQL functions (C++)”

INSERT

```
►►──INSERT──(──expression1──,──expression2──,──expression3──,──expression4──)─────►◄
```

The schema is SYSFUN.

Returns a string where *expression3* bytes have been deleted from *expression1*, beginning at *expression2*, and where *expression4* has been inserted into *expression1*, beginning at *expression2*. If the length of the result string exceeds the maximum for the return type, an error is returned (SQLSTATE 38552).

The first argument is a character string or a 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. The second and third arguments must be a numeric value with a data type of SMALLINT or INTEGER. If the first argument is a character string, then the fourth argument must also be a character string. If the first argument is a binary string, then the fourth argument must be a binary string. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. For the first and fourth arguments, CHAR is converted to VARCHAR and LONG VARCHAR to CLOB(1M), for second and third arguments SMALLINT is converted to INTEGER for processing by the function.

The result is based on the argument types as follows:
- VARCHAR(4000) if both the first and fourth arguments are VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if either the first or fourth argument is CLOB or LONG VARCHAR
- BLOB(1M) if both first and fourth arguments are BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
- Delete one character from the word ’DINING’ and insert ’VID’, both beginning at the third character.

  ```
  VALUES CHAR(INSERT('DINING', 3, 1, 'VID'), 10)
  ```

  This example returns the following:

  ```
  1
  ----------
  DIVIDING
  ```

  As mentioned, the output of the INSERT function is VARCHAR(4000). In this example, the function CHAR has been used to limit the output of INSERT to 10 bytes. The starting location of a particular string can be found using the LOCATE function.

Related reference:
- “LOCATE” on page 368

INTEGER

```
>>──┬─INTEGER─┬──(──┬─numeric-expression───┬──)──────────────────────────><
    └─INT─────┘     ├─character-expression─┤
                    ├─date-expression──────┤
                    └─time-expression──────┘
```

The schema is SYSIBM.

The INTEGER function returns an integer representation of a number, character string, date, or time in the form of an integer constant. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

numeric-expression
: An expression that returns a value of any built-in numeric data type.

  If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a large integer column or variable. If the whole part of the argument is not within the range of integers, an error occurs. The decimal part of the argument is truncated if present.

character-expression
: An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). The character string cannot be a long string.

  If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a large integer column or variable.

date-expression
: An expression that returns a value of the DATE data type. The result is an INTEGER value representing the date as *yyyymmdd*.

time-expression
: An expression that returns a value of the TIME data type. The result is an INTEGER value representing the time as *hhmmss*.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Examples:

- Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and employee number (EMPNO). The list should be in descending order of the calculated value.

  ```
  SELECT INTEGER (SALARY / EDLEVEL), SALARY, EDLEVEL, EMPNO
    FROM EMPLOYEE
    ORDER BY 1 DESC
  ```

- Using the EMPLOYEE table, select the EMPNO column in integer form for further processing in the application.

  ```
  SELECT INTEGER(EMPNO) FROM EMPLOYEE
  ```

- Assume that the column BIRTHDATE (date) has an internal value equivalent to '1964-07-20'.

```
INTEGER(BIRTHDATE)
```

results in the value `19 640 720`.

- Assume that the column STARTTIME (time) has an internal value equivalent to ’12:03:04’.

```
INTEGER(STARTTIME)
```

results in the value `120 304`.

JULIAN_DAY

```
►►──JULIAN_DAY──(──expression──)────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns an integer value representing the number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the date value specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

LCASE or LOWER

```
►►──┬─LCASE─┬──(──string-expression──)──────────────────────────►◄
    └─LOWER─┘
```

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

The LCASE or LOWER function returns a string in which all the SBCS characters have been converted to lowercase characters. That is, the characters A-Z will be translated to the characters a-z, and characters with diacritical marks will be translated to their lowercase equivalents if they exist. For example, in code page 850, É maps to é. Because not all characters are translated, LCASE(UCASE(*string-expression*)) does not necessarily return the same result as LCASE(*string-expression*).

The argument must be an expression whose value is a CHAR or VARCHAR data type.

The result of the function has the same data type and length attribute as the argument. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

Example:

Ensure that the characters in the value of column JOB in the EMPLOYEE table are returned in lowercase characters.

```
SELECT LCASE(JOB)
  FROM EMPLOYEE
  WHERE EMPNO = '000020';
```

The result is the value 'manager'.

Related reference:

- "LCASE (SYSFUN schema)" on page 364

LCASE (SYSFUN schema)

```
►►──LCASE──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns a string in which all the characters A-Z have been converted to the characters a-z (characters with diacritical marks are not converted). Note that LCASE(UCASE(string)) will therefore not necessarily return the same result as LCASE(string).

The argument can be of any built-in character string type. For a VARCHAR, the maximum length is 4 000 bytes, and for a CLOB, the maximum length is 1 048 576 bytes.

The result of the function is:

- VARCHAR(4000) if the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR

The result can be null; if the argument is null, the result is the null value.

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.

LEFT

```
►►──LEFT──(──expression1──,──expression2──)──────────────────────────────►◄
```

The schema is SYSFUN.

Returns a string consisting of the leftmost *expression2* bytes in *expression1*. The *expression1* value is effectively padded on the right with the necessary number of blank characters so that the specified substring of *expression1* always exists.

The first argument is a character string or a 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. The second argument must be of data type INTEGER or SMALLINT.

The result of the function is:
- VARCHAR(4000) if the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR
- BLOB(1M) if the argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

LENGTH

```
►►──LENGTH──(──expression──)────────────────────────────────────────►◄
```

The schema is SYSIBM.

The LENGTH function returns the length of a value.

The argument can be an expression that returns a value of any built-in data type.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the length of the argument. The length does not include the null indicator byte of column arguments that allow null values. The length of strings includes blanks but does not include the length control field of varying-length strings. The length of a varying-length string is the actual length, not the maximum length.

The length of a graphic string is the number of DBCS characters. The length of all other values is the number of bytes used to represent the value:
- 2 for small integer
- 4 for large integer
- (p/2)+1 for decimal numbers with precision p
- The length of the string for binary strings
- The length of the string for character strings
- 4 for single-precision floating-point
- 8 for double-precision floating-point
- 4 for date
- 3 for time
- 10 for timestamp

Examples:
- Assume the host variable ADDRESS is a varying length character string with a value of '895 Don Mills Road'.
  ```
  LENGTH(:ADDRESS)
  ```
  Returns the value 18.
- Assume that START_DATE is a column of type DATE.
  ```
  LENGTH(START_DATE)
  ```
  Returns the value 4.
- This example returns the value 10.
  ```
  LENGTH(CHAR(START_DATE, EUR))
  ```

LN

```
►►──LN──(──expression──)───────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the natural logarithm of the argument (same as LOG).

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

LOCATE

```
►►──LOCATE──(──search-string──,──source-string──┬────────────┬──)──────────────►◄
                                                 └─,──start───┘
```

The schema is SYSFUN.

Returns the starting position of the first occurrence of *search-string* within *source-string*. If the optional *start* is specified, it indicates the character position in *source-string* at which the search is to begin. The LOCATE function would then be equivalent to:

```
POSSTR(SUBSTR(source-string, start), search-string) + start - 1
```

If *search-string* is not found within *source-string*, the value 0 is returned.

If the first argument is a character string, the second argument must be a character string. For a VARCHAR, the maximum length is 4 000 bytes, and for a CLOB, the maximum length is 1 048 576 bytes. If the first argument is a binary string, the second argument must be a binary string with a maximum length of 1 048 576 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 third argument must be INTEGER or SMALLINT.

The result of the function is INTEGER. The result can be null; if any argument is null, the result is the null value.

Example:

- Find the location of the letter ’N’ (first occurrence) in the word ’DINING’.

  ```
  VALUES LOCATE ('N', 'DINING')
  ```

  This example returns the following:

  ```
  1
  -----------
            3
  ```

LOG

```
►►──LOG──(──expression──)──────────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the natural logarithm of the argument (same as LN).

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

LOG10

```
►►──LOG10──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the base 10 logarithm of the argument.

The argument can be of any built-in numeric type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

LONG_VARCHAR

```
►►──LONG_VARCHAR──(──character-string-expression──)──────────────────►◄
```

The schema is SYSIBM.

The LONG_VARCHAR function returns a LONG VARCHAR representation of a character string data type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

character-string-expression
: An *expression* that returns a value that is a character string with a maximum length of 32 700 bytes.

The result of the function is a LONG VARCHAR. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

LONG_VARGRAPHIC

```
►►──LONG_VARGRAPHIC──(──graphic-expression──)──────────────────────────────►◄
```

The schema is SYSIBM.

The LONG_VARGRAPHIC function returns a LONG VARGRAPHIC representation of a double-byte character string.

graphic-expression
: An *expression* that returns a value that is a graphic string with a maximum length of 16 350 double byte characters.

The result of the function is a LONG VARGRAPHIC. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

LTRIM

```
►►──LTRIM──(──string-expression──)──────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

The LTRIM function removes blanks from the beginning of *string-expression*.

The argument can be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.
- If the argument is a graphic string in a DBCS or EUC database, then the leading double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the leading UCS-2 blanks are removed.
- Otherwise, the leading single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of *string-expression* is VARCHAR or CHAR
- VARGRAPHIC if the data type of *string-expression* is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of *string-expression* minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example:

Assume that host variable HELLO is defined as CHAR(9) and has a value of ’ Hello’.

```
   VALUES LTRIM(:HELLO)
```

The result is ’Hello’.

Related reference:
- “LTRIM (SYSFUN schema)” on page 374

LTRIM (SYSFUN schema)

```
►►──LTRIM──(──expression──)──────────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the characters of the argument with leading blanks removed.

The argument can be of any built-in character string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes.

The result of the function is:
- VARCHAR(4000) if the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

The result can be null; if the argument is null, the result is the null value.

MICROSECOND

```
►►──MICROSECOND──(──expression──)────────────────────────────────────────►◄
```

The schema is SYSIBM.

The MICROSECOND function returns the microsecond part of a value.

The argument must be a timestamp, timestamp duration, or a valid character string representation of a timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a timestamp or a valid string representation of a timestamp:
  - The integer ranges from 0 through 999 999.
- If the argument is a duration:
  - The result reflects the microsecond part of the value which is an integer between −999 999 through 999 999. A nonzero result has the same sign as the argument.

Example:
- Assume a table TABLEA contains two columns, TS1 and TS2, of type TIMESTAMP. Select all rows in which the microseconds portion of TS1 is not zero and the seconds portion of TS1 and TS2 are identical.

```
SELECT * FROM TABLEA
  WHERE MICROSECOND(TS1) <> 0
    AND
  SECOND(TS1) = SECOND(TS2)
```

MIDNIGHT_SECONDS

```
►►──MIDNIGHT_SECONDS──(──expression──)─────────────────────────────────►◄
```

The schema is SYSFUN.

Returns an integer value in the range 0 to 86 400, representing the number of seconds between midnight and the time value specified in the argument.

The argument must be a time, timestamp, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Examples:

- Find the number of seconds between midnight and 00:10:10, and midnight and 13:10:10.

  ```
  VALUES (MIDNIGHT_SECONDS('00:10:10'), MIDNIGHT_SECONDS('13:10:10'))
  ```

  This example returns the following:

  ```
  1           2
  ----------- -----------
          610       47410
  ```

  Since a minute is 60 seconds, there are 610 seconds between midnight and the specified time. The same follows for the second example. There are 3600 seconds in an hour, and 60 seconds in a minute, resulting in 47410 seconds between the specified time and midnight.
- Find the number of seconds between midnight and 24:00:00, and midnight and 00:00:00.

  ```
  VALUES (MIDNIGHT_SECONDS('24:00:00'), MIDNIGHT_SECONDS('00:00:00'))
  ```

  This example returns the following:

  ```
  1           2
  ----------- -----------
        86400           0
  ```

  Note that these two values represent the same point in time, but return different MIDNIGHT_SECONDS values.

MINUTE

```
►►──MINUTE──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The MINUTE function returns the minute part of a value.

The argument must be a time, timestamp, time duration, timestamp duration, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the minute part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the minute part of the value, which is an integer between −99 and 99. A nonzero result has the same sign as the argument.

Example:
- Using the CL_SCHED sample table, select all classes with a duration less than 50 minutes.

```
SELECT * FROM CL_SCHED
  WHERE HOUR(ENDING - STARTING) = 0
    AND
  MINUTE(ENDING - STARTING) < 50
```

MOD

```
►►──MOD──(──expression──,──expression──)──────────────────────────────►◄
```

The schema is SYSFUN.

Returns the remainder of the first argument divided by the second argument. The result is negative only if first argument is negative.

The result of the function is:
- SMALLINT if both arguments are SMALLINT
- INTEGER if one argument is INTEGER and the other is INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT.

The result can be null; if any argument is null, the result is the null value.

MONTH

```
►►──MONTH──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

The MONTH function returns the month part of a value.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a date, timestamp, or a valid string representation of a date or timestamp:
  - The result is the month part of the value, which is an integer between 1 and 12.
- If the argument is a date duration or timestamp duration:
  - The result is the month part of the value, which is an integer between –99 and 99. A nonzero result has the same sign as the argument.

Example:

- Select all rows from the EMPLOYEE table for people who were born (BIRTHDATE) in DECEMBER.

```
SELECT * FROM EMPLOYEE
  WHERE MONTH(BIRTHDATE) = 12
```

MONTHNAME

```
►►──MONTHNAME──(──expression──)──────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns a mixed case character string containing the name of the month (for example, January) for the month portion of the argument, based on the locale when the database was started.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is VARCHAR(100). The result can be null; if the argument is null, the result is the null value.

MULTIPLY_ALT

```
►►──MULTIPLY_ALT──(──exact_numeric_expression──,──exact_numeric_expression──)─────►◄
```

The schema is SYSIBM.

The MULTIPLY_ALT scalar function returns the product of the two arguments as a decimal value. It is provided as an alternative to the multiplication operator, especially when the sum of the precisions of the arguments exceeds 31.

The arguments can be any built-in exact numeric data type (DECIMAL, BIGINT, INTEGER, or SMALLINT).

The result of the function is a DECIMAL. The precision and scale of the result are determined as follows, using the symbols *p* and *s* to denote the precision and scale of the first argument, and the symbols *p*' and *s*' to denote the precision and scale of the second argument.

The precision is 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 is a sample comparing the result types using MULTIPLY_ALT and the multiplication operator.

Type of argument 1	**Type of argument 2**	**Result using MULTIPLY_ALT**	**Result using multiplication operator**
DECIMAL(31,3)	DECIMAL(15,8)	DECIMAL(31,3)	DECIMAL(31,11)
DECIMAL(26,23)	DECIMAL(10,1)	DECIMAL(31,19)	DECIMAL(31,24)
DECIMAL(18,17)	DECIMAL(20,19)	DECIMAL(31,29)	DECIMAL(31,31)
DECIMAL(16,3)	DECIMAL(17,8)	DECIMAL(31,9)	DECIMAL(31,11)
DECIMAL(26,5)	DECIMAL(11,0)	DECIMAL(31,3)	DECIMAL(31,5)
DECIMAL(21,1)	DECIMAL(15,1)	DECIMAL(31,2)	DECIMAL(31,2)

Example:

Multiply two values where the data type of the first argument is DECIMAL(26,3) and the data type of the second argument is DECIMAL(9,8). The data type of the result is DECIMAL(31,7).

```
values multiply_alt(98765432109876543210987.654,5.43210987)
1
---------------------------------
536504678578875294857887.5277415
```

Note that the complete product of these two numbers is `536504678578875294857887.52774154498`, but the last 4 digits are truncated to match the scale of the result data type. Using the multiplication operator with the same values will cause an arithmetic overflow, since the result data type is DECIMAL(31,11) and the result value has 24 digits left of the decimal, but the result data type only supports 20 digits.

NULLIF

```
▸▸──NULLIF──(──expression──,──expression──)──────────────────────────▸◂
```

The schema is SYSIBM.

The NULLIF function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

The arguments must be comparable. They can be of either a built-in (other than a long string or DATALINK) or distinct data type (other than based on a long string or DATALINK). (This function cannot be used as a source function when creating a user-defined function. Because this function accepts any compatible data types as arguments, it is not necessary to create additional signatures to support user-defined distinct types.) The attributes of the result are the attributes of the first argument.

The result of using NULLIF(e1,e2) is the same as using the expression

```
CASE WHEN e1=e2 THEN NULL ELSE e1 END
```

Note that when e1=e2 evaluates to unknown (because one or both arguments is NULL), CASE expressions consider this not true. Therefore, in this situation, NULLIF returns the value of the first argument.

Example:

- Assume host variables PROFIT, CASH, and LOSSES have DECIMAL data types with the values 4500.00, 500.00, and 5000.00 respectively:

  ```
  NULLIF (:PROFIT + :CASH , :LOSSES )
  ```

  Returns a null value.

Related reference:

- "Assignments and comparisons" on page 99

POSSTR

```
►►──POSSTR──(──source-string──,──search-string──)──────────────────────►◄
```

The schema is SYSIBM.

The POSSTR function returns the starting position of the first occurrence of one string (called the *search-string*) within another string (called the *source-string*). Numbers for the *search-string* position start at 1 (not 0).

The result of the function is a large integer. If either of the arguments can be null, the result can be null; if either of the arguments is null, the result is the null value.

source-string
: An expression that specifies the source string in which the search is to take place.

  The expression can be specified by any one of:
  - a constant
  - a special register
  - a host variable (including a locator variable or a file reference variable)
  - a scalar function
  - a large object locator
  - a column name
  - an expression concatenating any of the above

search-string
: An expression that specifies the string that is to be searched for.

  The expression can be specified by any one of:
  - a constant
  - a special register
  - a host variable
  - a scalar function whose operands are any of the above
  - an expression concatenating any of the above

  with the restrictions that:
  - No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.
  - The actual length of *search-string* cannot be more than 4 000 bytes.

Both *search-string* and *source-string* have zero or more contiguous positions. If the strings are character or binary strings, a position is a byte. If the strings are graphic strings, a position is a graphic (DBCS) character.

The POSSTR function accepts mixed data strings. However, POSSTR operates on a strict byte-count basis, oblivious to changes between single and multi-byte characters.

The following rules apply:

- The data types of *source-string* and *search-string* must be compatible, otherwise an error is raised (SQLSTATE 42884).
  - If *source-string* is a character string, then *search-string* must be a character string, but not a CLOB or LONG VARCHAR, with an actual length of 32 672 bytes or less.
  - If *source-string* is a graphic string, then *search-string* must be a graphic string, but not a DBCLOB or LONG VARGRAPHIC, with an actual length of 16 336 double-byte characters or less.
  - If *source-string* is a binary string, then *search-string* must be a binary string with an actual length of 32 672 bytes or less.
- If *search-string* has a length of zero, the result returned by the function is 1.
- Otherwise:
  - If *source-string* has a length of zero, the result returned by the function is zero.
  - Otherwise:
    - If the value of *search-string* is equal to an identical length substring of contiguous positions from the value of *source-string*, then the result returned by the function is the starting position of the first such substring within the *source-string* value.
    - Otherwise, the result returned by the function is 0.

Example

- Select RECEIVED and SUBJECT columns as well as the starting position of the words 'GOOD BEER' within the NOTE_TEXT column for all entries in the IN_TRAY table that contain these words.

```
SELECT RECEIVED, SUBJECT, POSSTR(NOTE_TEXT, 'GOOD BEER')
  FROM IN_TRAY
  WHERE POSSTR(NOTE_TEXT, 'GOOD BEER') <> 0
```

POWER

```
►►──POWER──(──expression1──,──expression2──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the value of *expression1* to the power of *expression2*.

The arguments can be of any built-in numeric data type. DECIMAL and REAL arguments are converted to a double-precision floating-point number.

The result of the function is:
- INTEGER if both arguments are INTEGER or SMALLINT
- BIGINT if one argument is BIGINT and the other argument is BIGINT, INTEGER or SMALLINT
- DOUBLE otherwise.

The result can be null; if any argument is null, the result is the null value.

QUARTER

```
►►──QUARTER──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns an integer value in the range 1 to 4, representing the quarter of the year for the date specified in the argument.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

RADIANS

```
►►──RADIANS──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the number of radians converted from argument which is expressed in degrees.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

RAISE_ERROR

```
►►──RAISE_ERROR──(──sqlstate──,──diagnostic-string──)──────────────────────►◄
```

The schema is SYSIBM.

The RAISE_ERROR function causes the statement that includes the function to return an error with the specified SQLSTATE, SQLCODE -438, and *diagnostic-string*. The RAISE_ERROR function always returns NULL with an undefined data type. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

sqlstate
: A character string containing exactly 5 characters. It must be of type CHAR defined with a length of 5 or type VARCHAR defined with a length of 5 or greater. The *sqlstate* value must follow the rules for application-defined SQLSTATEs as follows:

- Each character must be from the set of digits (’0’ through ’9’) or non-accented upper case letters (’A’ through ’Z’)
- The SQLSTATE class (first two characters) cannot be ’00’, ’01’ or ’02’ since these are not error classes.
- If the SQLSTATE class (first two characters) starts with the character ’0’ through ’6’ or ’A’ through ’H’, then the subclass (last three characters) must start with a letter in the range ’I’ through ’Z’
- If the SQLSTATE class (first two characters) starts with the character ’7’, ’8’, ’9’ or ’I’ though ’Z’, then the subclass (last three characters) can be any of ’0’ through ’9’ or ’A’ through ’Z’.

If the SQLSTATE does not conform to these rules an error occurs (SQLSTATE 428B3).

diagnostic-string
: An expression of type CHAR or VARCHAR that returns a character string of up to 70 bytes that describes the error condition. If the string is longer than 70 bytes, it will be truncated.

To use this function in a context where the rules for result data types do not apply (such as alone in a select list), a cast specification must be used to give the null returned value a data type. A CASE expression is where the RAISE_ERROR function will be most useful.

Example:

List employee numbers and education levels as Post Graduate, Graduate and Diploma. If an education level is greater than 20, raise an error.

```
SELECT EMPNO,
  CASE WHEN EDUCLVL < 16 THEN 'Diploma'
       WHEN EDUCLVL < 18 THEN 'Graduate'
       WHEN EDUCLVL < 21 THEN 'Post Graduate'
       ELSE RAISE_ERROR('70001',
         'EDUCLVL has a value greater than 20')
  END
FROM EMPLOYEE
```

RAND

```
>>──RAND──(──┬────────────┬──)──────────────────────────────────────><
             └─expression─┘
```

The schema is SYSFUN.

The RAND function returns a floating point value between 0 and 1.

If an expression is specified, it is used as the seed value. The expression must be a built-in SMALLINT or INTEGER data type with a value between 0 and 2 147 483 647.

The data type of the result is double-precision floating point. If the argument is null, the result is the null value.

A specific seed value will produce the same sequence of random numbers for a specific instance of a RAND function in a query each time the query is executed. If a seed value is not specified, a different sequence of random numbers is produced each time the query is executed within the same session. To produce a set of random numbers that varies from session to session, specify a random seed; for example, one that is based on the current time.

RAND is a non-deterministic function.

REAL

```
►►──REAL──(──numeric-expression──)─────────────────────────────────►◄
```

The schema is SYSIBM.

The REAL function returns a single-precision floating-point representation of a number.

The argument is an expression that returns a value of any built-in numeric data type.

The result of the function is a single-precision floating-point number. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the same number that would occur if the argument were assigned to a single-precision floating-point column or variable.

Example:

Using the EMPLOYEE table, find the ratio of salary to commission for employees whose commission is not zero. The columns involved (SALARY and COMM) have DECIMAL data types. The result is desired in single-precision floating point. 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

```
>>--REC2XML--(--decimal-constant--,--format-string--,--row-tag-string----------->

    .-------------------.
    V                   |
>-----,--column-name-----+--)-------------------------------------------------><
```

The schema is SYSIBM.

The REC2XML function returns a string formatted with XML tags, containing column names and column 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.

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-->----------------------------------------------------->

    .--------------------------------------------------------------------.
    V                                                                    |
>-----<--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 above types

The same column name cannot be specified more than once (SQLSTATE 42734).

The result of the function is VARCHAR. The maximum length is 32 672 bytes (SQLSTATE 54006).

Consider the following invocation:

REC2XML (dc, fs, rt, c_1, c_2, ..., c_n)

If the value of "fs" is either "COLATTVAL" or "COLATTVAL_XML", then the result is the same as this expression:

'<' CONCAT rt CONCAT '>' CONCAT y_1 CONCAT y_2
CONCAT ... CONCAT y_n CONCAT '</' CONCAT rt CONCAT '>'

where y_n is equivalent to:

'<column name="' CONCAT xvc_n CONCAT vn

and vn is equivalent to:

```
'">' CONCAT r_n 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 26 on page 394 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 25

Table 25. 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 26, 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 26. Character Replacements for XML Attribute Values and Element Values

Character	Replacement
<	&lt;
>	&gt;
"	&quot;
&	&amp;
'	&apos;

Examples:

Note: REC2XML does not insert new line characters in the output. All example output is formatted for the sake of readability.

- Using the DEPARTMENT table in the sample database, format the department table row, except the DEPTNAME and LOCATION columns, for department 'D01' into an XML string. Since the data does not contain any of the characters which require replacement, the expansion factor will be 1.0 (no expansion). Also note that the MGRNO value is null for this row.

  ```
  SELECT REC2XML (1.0, 'COLATTVAL', '', DEPTNO, MGRNO, ADMRDEPT)
    FROM DEPARTMENT
    WHERE DEPTNO = 'D01'
  ```

  This example returns the following VARCHAR(117) string:

```
<row>
<column name="DEPTNO">D01</column>
<column name="MGRNO" null="true"/>
<column name="ADMRDEPT">A00</column>
</row>
```

- A 5-day university schedule introduces a class named ’&43<FIE’ to a table called CL_SCHED, with a new format for the CLASS_CODE column. Using the REC2XML function, this example formats an XML string with this new class data, except for the class end time.

  The length attribute for the REC2XML call (see below) with an expansion factor of 1.0 would be 128 (11 for the ’<row>’ and ’</row>’ overhead, 21 for the column names, 75 for the ’<column name=’, ’>’, ’</column>’ and double quotes, 7 for the CLASS_CODE data, 6 for the DAY data, and 8 for the STARTING data). Since the ’&’ and ’<’ characters will be replaced, an expansion factor of 1.0 will not be sufficient. The length attribute of the function will need to support an increase from 7 to 14 characters 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>
  ```

- 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 characters in length. The following query is used to select the XML version of the resumes from the EMP_RESUME table and format it into an XML document fragment.

  ```
  SELECT REC2XML (1.0, 'COLATTVAL_XML', 'row', EMPNO, RESUME_XML)
    FROM (SELECT EMPNO, CAST(RESUME AS VARCHAR(3500)) AS RESUME_XML
      FROM EMP_RESUME
      WHERE RESUME_FORMAT = 'XML')
    AS EMP_RESUME_XML
  ```

  This example returns a row for each employee who has a resume in XML format. Each returned row will be a string with the following format:

  ```
  <row>
  <column name="EMPNO">{employee number}</column>
  <column name="RESUME_XML">{resume in XML}</column>
  </row>
  ```

  Where ″{employee number}″ is the actual EMPNO value for the column and ″{resume in XML}″ is the actual XML fragment string value that is the resume.

REPEAT

```
►►──REPEAT──(──expression──,──expression──)──────────────────────────────►◄
```

The schema is SYSFUN.

Returns a character string composed of the first argument repeated the number of times specified by the second argument. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The first argument is a character string or binary string type. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB or a binary string the maximum length is 1 048 576 bytes. The second argument can be SMALLINT or INTEGER.

The result of the function is:
- VARCHAR(4000) if the first argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:
- List the phrase 'REPEAT THIS' five times.
  ```
  VALUES CHAR(REPEAT('REPEAT THIS', 5), 60)
  ```
  This example return the following:
  ```
  1
  ------------------------------------------------------------
  REPEAT THISREPEAT THISREPEAT THISREPEAT THISREPEAT THIS
  ```
  As mentioned, the output of the REPEAT function is VARCHAR(4000). For this example, the CHAR function has been used to limit the output of REPEAT to 60 bytes.

REPLACE

```
▶▶──REPLACE──(──expression1──,──expression2──,──expression3──)──────────────▶◀
```

The schema is SYSFUN.

Replaces all occurrences of *expression2* in *expression1* with *expression3*.

The first argument can be of any built-in character string or binary string type. 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 type of the second and third arguments is identical to that of the first argument.

The result of the function is:

- VARCHAR(4000) if the first, second, and third arguments are VARCHAR or CHAR
- CLOB(1M) if the first, second, and third arguments are CLOB or LONG VARCHAR
- BLOB(1M) if the first, second, and third arguments are BLOB.

The result can be null; if any argument is null, the result is the null value.

Example:

Replace all occurrence of the letter ’N’ in the word ’DINING’ with ’VID’.

```
   VALUES CHAR (REPLACE ('DINING', 'N', 'VID'), 10)
```

This example returns the following:

```
1
----------
DIVIDIVIDG
```

As mentioned, the output of the REPLACE function is VARCHAR(4000). For this example, the CHAR function has been used to limit the output of REPLACE to 10 bytes.

RIGHT

```
►►—RIGHT—(—expression1—,—expression2—)———————————————————————————►◄
```

Returns a string consisting of the rightmost *expression2* bytes in *expression1*. The *expression1* value is effectively padded on the right with the necessary number of blank characters so that the specified substring of *expression1* always exists.

The first argument is a character string or a 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. The second argument can be INTEGER or SMALLINT.

The result of the function is:
- VARCHAR(4000) if the first argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the first argument is CLOB or LONG VARCHAR
- BLOB(1M) if the first argument is BLOB.

The result can be null; if any argument is null, the result is the null value.

ROUND

```
►►—ROUND—(—expression1—,—expression2—)——————————————————————————►◄
```

The schema is SYSIBM. (The SYSFUN version of the ROUND function continues to be available.)

The ROUND function returns *expression1* rounded to *expression2* places to the right of the decimal point if *expression2* is positive, or to the left of the decimal point if *expression2* is zero or negative.

If *expression1* is positive, a digit value of 5 or greater is an indication to round to the next higher positive number. For example, ROUND(3.5,0) = 4. If *expression1* is negative, a digit value of 5 or greater is an indication to round to the next lower negative number. For example, ROUND(-3.5,0) = -4.

expression1
: An expression that returns a value of any built-in numeric data type.

expression2
: An expression that returns a small or large integer. When the value of *expression2* is not negative, it specifies rounding to that number of places to the right of the decimal separator. When the value of *expression2* is negative, it specifies rounding to the absolute value of *expression2* places to the left of the decimal separator.

  If *expression2* is not negative, expression1 is rounded to the absolute value of *expression2* number of places to the right of the decimal point. If the value of *expression2* is greater than the scale of *expression1* then the value is unchanged except that the result value has a precision that is larger by 1. For example, ROUND(748.58,5) = 748.58 where the precision is now 6 and the scale remains 2.

  If *expression2* is negative, *expression1* is rounded to the absolute value of *expression2*+1 number of places to the left of the decimal point.

  If the absolute value of a negative *expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example, ROUND(748.58,-4) = 0.

The data type and length attribute of the result are the same as the data type and length attribute of the first argument, except that the precision is increased by one if the *expression1* is DECIMAL and the precision is less than 31.

For example, an argument with a data type of DECIMAL(5,2) results in DECIMAL(6,2). An argument with a data type of DECIMAL(31,2) results in DECIMAL(31,2). The scale is the same as the scale of the first argument.

If either argument can be null or 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.

Examples:

Calculate the value of 873.726, rounded to 2, 1, 0, -1, -2, -3, and -4 decimal places, respectively.

```
   VALUES (
     ROUND(873.726, 2),
     ROUND(873.726, 1),
     ROUND(873.726, 0),
     ROUND(873.726,-1),
     ROUND(873.726,-2),
     ROUND(873.726,-3),
     ROUND(873.726,-4) )
```

This example returns:

```
1         2         3         4         5         6         7
--------- --------- --------- --------- --------- --------- ---------
  873.730   873.700   874.000   870.000   900.000  1000.000     0.000
```

Calculate using both positive and negative numbers.

```
   VALUES (
     ROUND(3.5, 0),
     ROUND(3.1, 0),
     ROUNDROUND(-3.1, 0),
     ROUND(-3.5,0) )
```

This example returns:

```
1    2    3    4
---- ---- ---- ----
 4.0  3.0 -3.0 -4.0
```

RTRIM

```
►►──RTRIM──(──string-expression──)─────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of this function continues to be available with support for LONG VARCHAR and CLOB arguments.)

The RTRIM function removes blanks from the end of *string-expression*.

The argument can be a CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC data type.
- If the argument is a graphic string in a DBCS or EUC database, then the trailing double byte blanks are removed.
- If the argument is a graphic string in a Unicode database, then the trailing UCS-2 blanks are removed.
- Otherwise, the trailing single byte blanks are removed.

The result data type of the function is:
- VARCHAR if the data type of *string-expression* is VARCHAR or CHAR
- VARGRAPHIC if the data type of *string-expression* is VARGRAPHIC or GRAPHIC

The length parameter of the returned type is the same as the length parameter of the argument data type.

The actual length of the result for character strings is the length of *string-expression* minus the number of bytes removed for blank characters. The actual length of the result for graphic strings is the length (in number of double byte characters) of *string-expression* minus the number of double byte blank characters removed. If all of the characters are removed, the result is an empty, varying-length string (length is zero).

If the argument can be null, the result can be null; if the argument is null, the result is the null value.

Example: Assume that host variable HELLO is defined as CHAR(9) and has a value of 'Hello'.

```
   VALUES RTRIM(:HELLO)
```

The result is 'Hello'.

Related reference:
- "RTRIM (SYSFUN schema)" on page 402

RTRIM (SYSFUN schema)

```
►►──RTRIM──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the characters of the argument with trailing blanks removed.

The argument can be of any built-in character string data types. For a VARCHAR the maximum length is 4 000 bytes and for a CLOB the maximum length is 1 048 576 bytes.

The result of the function is:
- VARCHAR(4000) if the argument is VARCHAR (not exceeding 4 000 bytes) or CHAR
- CLOB(1M) if the argument is CLOB or LONG VARCHAR.

The result can be null; if the argument is null, the result is the null

SECOND

```
►►──SECOND──(──expression──)───────────────────────────────────────────►◄
```

The schema is SYSIBM.

The SECOND function returns the seconds part of a value.

The argument must be a time, timestamp, time duration, timestamp duration, or a valid character string representation of a time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:

- If the argument is a time, timestamp or valid string representation of a time or timestamp:
  - The result is the seconds part of the value, which is an integer between 0 and 59.
- If the argument is a time duration or timestamp duration:
  - The result is the seconds part of the value, which is an integer between –99 and 99. A nonzero result has the same sign as the argument.

Examples:

- Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.

  ```
  SECOND(:TIME_DUR)
  ```

  Returns the value 45.
- Assume that the column RECEIVED (timestamp) has an internal value equivalent to 1988-12-25-17.12.30.000000.

  ```
  SECOND(RECEIVED)
  ```

  Returns the value 30.

SIGN

```
►►──SIGN──(──expression──)──────────────────────────────────────────►◄
```

Returns an indicator of the sign of the argument. If the argument is less than zero, −1 is returned. If argument equals zero, 0 is returned. If argument is greater than zero, 1 is returned.

The argument can be of any built-in numeric data type. DECIMAL and REAL values are converted to double-precision floating-point numbers for processing by the function.

The result of the function is:
- SMALLINT if the argument is SMALLINT
- INTEGER if the argument is INTEGER
- BIGINT if the argument is BIGINT
- DOUBLE otherwise.

The result can be null; if the argument is null, the result is the null value.

SIN

```
►►──SIN──(──expression──)──────────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the SIN function continues to be available.)

Returns the sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

SINH

```
►►──SINH──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

SMALLINT

```
►►──SMALLINT──(──┬─numeric-expression───┬──)─────────────────────────────────►◄
                 └─character-expression─┘
```

The schema is SYSIBM.

The SMALLINT function returns a small integer representation of a number or character string in the form of a small integer constant. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

numeric-expression
: An expression that returns a value of any built-in numeric data type.

  If the argument is a *numeric-expression*, the result is the same number that would occur if the argument were assigned to a small integer column or variable. If the whole part of the argument is not within the range of small integers, an error occurs. The decimal part of the argument is truncated if present.

character-expression
: An expression that returns a character string value of length not greater than the maximum length of a character constant. Leading and trailing blanks are eliminated and the resulting string must conform to the rules for forming an SQL integer constant (SQLSTATE 22018). However, the value of the constant must be in the range of small integers (SQLSTATE 22003). The character string cannot be a long string.

  If the argument is a *character-expression*, the result is the same number that would occur if the corresponding integer constant were assigned to a small integer column or variable.

The result of the function is a small integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

SOUNDEX

```
►►──SOUNDEX──(──expression──)────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns a 4-character code representing the sound of the words in the argument. The result can be used to compare with the sound of other strings.

The argument can be a character string that is either a CHAR or VARCHAR not exceeding 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 result of the function is CHAR(4). The result can be null; if the argument is null, the result is the null value.

The SOUNDEX function is useful for finding strings for which the sound is known but the precise spelling is not. It makes assumptions about the way that letters and combinations of letters sound that can help to search out words with similar sounds. The comparison can be done directly or by passing the strings as arguments to the DIFFERENCE function.

Example:

Using the EMPLOYEE table, find the EMPNO and LASTNAME of the employee with a surname that sounds like 'Loucesy'.

```
   SELECT EMPNO, LASTNAME FROM EMPLOYEE
     WHERE SOUNDEX(LASTNAME) = SOUNDEX('Loucesy')
```

This example returns the following:

```
EMPNO  LASTNAME
------ ---------------
000110 LUCCHESSI
```

Related reference:

- "DIFFERENCE" on page 317

SPACE

```
►►──SPACE──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns a character string consisting of blanks with length specified by the second argument.

The argument can be SMALLINT or INTEGER.

The result of the function is VARCHAR(4000). The result can be null; if the argument is null, the result is the null value.

SQRT

```
▸▸—SQRT—(—expression—)———————————————————————————————————————▸◂
```

The schema is SYSFUN.

Returns the square root of the argument.

The argument can be any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null; if the argument is null, the result is the null value.

SUBSTR

```
►►──SUBSTR──(──string──,──start──┬────────────┬──)─────────────────────────►◄
                                 └─,──length──┘
```

The SUBSTR function returns a substring of a string.

If *string* is a character string, the result of the function is a character string represented in the code page of its first argument. If it is a binary string, the result of the function is a binary string. If it is a graphic string, the result of the function is a graphic string represented in the code page of its first argument. If the first argument is a host variable, the code page of the result is the database code page. If any argument of the SUBSTR function can be null, the result can be null; if any argument is null, the result is the null value.

string
: An expression that specifies the string from which the result is derived.

  If *string* is either a character string or a binary string, a substring of *string* is zero or more contiguous bytes of *string*. If *string* is a graphic string, a substring of *string* is zero or more contiguous double-byte characters of *string*.

start
: An expression that specifies the position of the first byte of the result for a character string or a binary string or the position of the first character of the result for a graphic string. *start* must be an integer between 1 and the length or maximum length of *string*, depending on whether *string* is fixed-length or varying-length (SQLSTATE 22011, if out of range). It must be specified as number of bytes in the context of the database code page and not the application code page.

length
: An expression that specifies the length of the result. If specified, *length* must be a binary integer in the range 0 to *n*, where *n* equals (the length attribute of *string*) − *start* + 1 (SQLSTATE 22011, if out of range).

  If *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of blank characters (single-byte for character strings; double-byte for graphic strings) or hexadecimal zero characters (for BLOB strings) so that the specified substring of *string* always exists. The default for *length* is the number of bytes from the byte specified by the *start* to the last byte of *string* in the case of character string or binary string or the number of double-byte characters from the character specified by the *start* to the last character of *string* in the case of a graphic string. However, if *string* is a varying-length string with a length less than *start*, the default is zero and the result is the empty string. It must be specified as number of bytes in the context of the database code page and not the application code page. (For example, the column NAME with a data type of VARCHAR(18) and a value of 'MCKNIGHT' will yield an empty string with `SUBSTR(NAME,10)`).

Table 27 on page 412 shows that the result type and length of the SUBSTR function depend on the type and attributes of its inputs.

Table 27. Data Type and Length of SUBSTR Result

String Argument Data Type	Length Argument	Result Data Type
CHAR(A)	constant (*l*<255)	CHAR(*l*)
CHAR(A)	not specified but *start* argument is a constant	CHAR(A-*start*+1)
CHAR(A)	not a constant	VARCHAR(A)
VARCHAR(A)	constant (*l*<255)	CHAR(*l*)
VARCHAR(A)	constant (254<*l*<32673)	VARCHAR(*l*)
VARCHAR(A)	not a constant or not specified	VARCHAR(A)
LONG VARCHAR	constant (*l*<255)	CHAR(*l*)
LONG VARCHAR	constant (254<*l*<4001)	VARCHAR(*l*)
LONG VARCHAR	constant (*l*>4000)	LONG VARCHAR
LONG VARCHAR	not a constant or not specified	LONG VARCHAR
CLOB(A)	constant (*l*)	CLOB(*l*)
CLOB(A)	not a constant or not specified	CLOB(A)
GRAPHIC(A)	constant (*l*<128)	GRAPHIC(*l*)
GRAPHIC(A)	not specified but *start* argument is a constant	GRAPHIC(A-*start*+1)
GRAPHIC(A)	not a constant	VARGRAPHIC(A)
VARGRAPHIC(A)	constant (*l*<128)	GRAPHIC(*l*)
VARGRAPHIC(A)	constant (127<*l*<16337)	VARGRAPHIC(*l*)
VARGRAPHIC(A)	not a constant	VARGRAPHIC(A)
LONG VARGRAPHIC	constant (*l*<128)	GRAPHIC(*l*)
LONG VARGRAPHIC	constant (127<*l*<2001)	VARGRAPHIC(*l*)
LONG VARGRAPHIC	constant (*l*>2000)	LONG VARGRAPHIC
LONG VARGRAPHIC	not a constant or not specified	LONG VARGRAPHIC
DBCLOB(A)	constant (*l*)	DBCLOB(*l*)
DBCLOB(A)	not a constant or not specified	DBCLOB(A)
BLOB(A)	constant (*l*)	BLOB(*l*)
BLOB(A)	not a constant or not specified	BLOB(A)

If *string* is a fixed-length string, omission of *length* is an implicit specification of LENGTH(*string*) - *start* + 1. If *string* is a varying-length string, omission of *length* is an implicit specification of zero or LENGTH(*string*) - *start* + 1, whichever is greater.

Examples:

- Assume the host variable NAME (VARCHAR(50)) has a value of 'BLUE JAY' and the host variable SURNAME_POS (int) has a value of 6.

  ```
  SUBSTR(:NAME, :SURNAME_POS)
  ```

  Returns the value 'JAY'

  ```
  SUBSTR(:NAME, :SURNAME_POS,1)
  ```

  Returns the value 'J'.
- Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION'.

  ```
  SELECT * FROM PROJECT
    WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '
  ```

  The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

Notes:

1. In dynamic SQL, *string*, *start*, and *length* may be represented by a parameter marker (?). If a parameter marker is used for *string*, the data type of the operand will be VARCHAR, and the operand will be nullable.
2. Though not explicitly stated in the result definitions above, it follows from these semantics that if *string* is a mixed single- and multi-byte character string, the result may contain fragments of multi-byte characters, depending upon the values of *start* and *length*. That is, the result could possibly begin with the second byte of a double-byte character, and/or end with the first byte of a double-byte character. The SUBSTR function does not detect such fragments, nor provides any special processing should they occur.

TABLE_NAME

```
►►──TABLE_NAME──(──objectname──┬──────────────────┬──)──────────────────────────►◄
                               └─,──objectschema──┘
```

The schema is SYSIBM.

The TABLE_NAME function returns an unqualified name of the object found after any alias chains have been resolved. The specified *objectname* (and *objectschema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the unqualified name of the starting point is returned. The resulting name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

objectname
A character expression representing the unqualified name (usually of an existing alias) to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 characters.

objectschema
A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 characters.

If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If *objectname* can be null, the result can be null; if *objectname* is null, the result is the null value. If *objectschema* is the null value, the default schema name is used. The result is the character string representing an unqualified name. The result name could represent one of the following:

table The value for *objectname* was either a table name (the input value is returned) or an alias name that resolved to the table whose name is returned.

view The value for *objectname* was either a view name (the input value is returned) or an alias name that resolved to the view whose name is returned.

undefined object
The value for *objectname* was either an undefined object (the input value is returned) or an alias name that resolved to the undefined object whose name is returned.

Therefore, if a non-null value is given to this function, a value is always returned, even if no object with the result name exists.

TABLE_SCHEMA

```
►►──TABLE_SCHEMA──(──objectname──┬──────────────────┬──)──────────────────────►◄
                                 └─,──objectschema──┘
```

The schema is SYSIBM.

The TABLE_SCHEMA function returns the schema name of the object found after any alias chains have been resolved. The specified *objectname* (and *objectschema*) are used as the starting point of the resolution. If the starting point does not refer to an alias, the schema name of the starting point is returned. The resulting schema name may be of a table, view, or undefined object. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

objectname
: A character expression representing the unqualified name (usually of an existing alias) to be resolved. *objectname* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 characters.

objectschema
: A character expression representing the schema used to qualify the supplied *objectname* value before resolution. *objectschema* must have a data type of CHAR or VARCHAR and a length greater than 0 and less than 129 characters.

  If *objectschema* is not supplied, the default schema is used for the qualifier.

The data type of the result of the function is VARCHAR(128). If *objectname* can be null, the result can be null; if *objectname* is null, the result is the null value. If *objectschema* is the null value, the default schema name is used. The result is the character string representing a schema name. The result schema could represent the schema name for one of the following:

table
: The value for *objectname* was either a table name (the input or default value of *objectschema* is returned) or an alias name that resolved to a table for which the schema name is returned.

view
: The value for *objectname* was either a view name (the input or default value of *objectschema* is returned) or an alias name that resolved to a view for which the schema name is returned.

undefined object
: The value for *objectname* was either an undefined object (the input or default value of *objectschema* is returned) or an alias name that resolved to an undefined object for which the schema name is returned.

Therefore, if a non-null *objectname* value is given to this function, a value is always returned, even if the object name with the result schema name does not exist. For example, `TABLE_SCHEMA('DEPT', 'PEOPLE')` returns 'PEOPLE ' if the catalog entry is not found.

Examples:

- PBIRD tries to select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A1 defined on the table HEDGES.T1.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('A1')
    AND TABSCHEMA = TABLE_SCHEMA ('A1')
  ```

The requested statistics for HEDGES.T1 are retrieved from the catalog.

- Select the statistics for an object called HEDGES.X1 from SYSCAT.TABLES using HEDGES.X1. Use TABLE_NAME and TABLE_SCHEMA since it is not known whether HEDGES.X1 is an alias or a table.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('X1','HEDGES')
    AND TABSCHEMA = TABLE_SCHEMA ('X1','HEDGES')
  ```

  Assuming that HEDGES.X1 is a table, the requested statistics for HEDGES.X1 are retrieved from the catalog.

- Select the statistics for a given table from SYSCAT.TABLES using an alias PBIRD.A2 defined on HEDGES.T2 where HEDGES.T2 does not exist.

  ```
  SELECT NPAGES, CARD FROM SYSCAT.TABLES
    WHERE TABNAME = TABLE_NAME ('A2','PBIRD')
    AND TABSCHEMA = TABLE_SCHEMA ('A2',PBIRD')
  ```

  The statement returns 0 records as no matching entry is found in SYSCAT.TABLES where TABNAME = 'T2' and TABSCHEMA = 'HEDGES'.

- Select the qualified name of each entry in SYSCAT.TABLES along with the final referenced name for any alias entry.

  ```
  SELECT TABSCHEMA AS SCHEMA,  TABNAME AS NAME,
    TABLE_SCHEMA (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_SCHEMA,
    TABLE_NAME (BASE_TABNAME, BASE_TABSCHEMA) AS REAL_NAME
    FROM SYSCAT.TABLES
  ```

  The statement returns the qualified name for each object in the catalog and the final referenced name (after alias has been resolved) for any alias entries. For all non-alias entries, BASE_TABNAME and BASE_TABSCHEMA are null so the REAL_SCHEMA and REAL_NAME columns will contain nulls.

TAN

```
►►──TAN──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM. (The SYSFUN version of the TAN function continues to be available.)

Returns the tangent of the argument, where the argument is an angle expressed in radians.

The argument can be any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

TANH

```
►►──TANH──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

Returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.

The argument can be of any built-in numeric data type. It is converted to a double-precision floating-point number for processing by the function.

The result of the function is a double-precision floating-point number. The result can be null if the argument can be null or the database is configured with DFT_SQLMATHWARN set to YES; the result is the null value if the argument is null.

TIME

```
►►──TIME──(──expression──)─────────────────────────────────────────►◄
```

The schema is SYSIBM.

The TIME function returns a time from a value.

The argument must be a time, timestamp, or a valid string representation of a time or timestamp that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.

Only Unicode databases support an argument that is a graphic string representation of a time or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a time. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument:
- If the argument is a time:
  - The result is that time.
- If the argument is a timestamp:
  - The result is the time part of the timestamp.
- If the argument is a string:
  - The result is the time represented by the string.

Example:
- Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.
  ```
  SELECT * FROM IN_TRAY
    WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
  ```

TIMESTAMP

```
►►──TIMESTAMP──(──expression──┬───────────────┬──)──────────────────────────►◄
                              └─,expression───┘
```

The schema is SYSIBM.

The TIMESTAMP function returns a timestamp from a value or a pair of values.

Only Unicode databases support an argument that is a graphic string representation of a date, a time, or a timestamp. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The rules for the arguments depend on whether the second argument is specified.

- If only one argument is specified:
  - It must be a timestamp, a valid string representation of a timestamp, or a string of length 14 that is not a CLOB, LONG VARCHAR, DBCLOB, or LONG VARGRAPHIC.

    A string of length 14 must be a string of digits that represents a valid date and time in the form *yyyyxxddhhmmss*, where *yyyy* is the year, *xx* is the month, *dd* is the day, *hh* is the hour, *mm* is the minute, and *ss* is the seconds.
- If both arguments are specified:
  - The first argument must be a date or a valid string representation of a date and the second argument must be a time or a valid string representation of a time.

The result of the function is a timestamp. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The other rules depend on whether the second argument is specified:

- If both arguments are specified:
  - The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp:
  - The result is that timestamp.
- If only one argument is specified and it is a string:
  - The result is the timestamp represented by that string. If the argument is a string of length 14, the timestamp has a microsecond part of zero.

Example:

- Assume the column START_DATE (date) has a value equivalent to 1988-12-25, and the column START_TIME (time) has a value equivalent to 17.12.30.

  ```
  TIMESTAMP(START_DATE, START_TIME)
  ```

  Returns the value '1988-12-25-17.12.30.000000'.

TIMESTAMP_FORMAT

```
►►──TIMESTAMP_FORMAT──(──string-expression──,format-string──)─────────────────►◄
```

The schema is SYSIBM.

The TIMESTAMP_FORMAT function returns a timestamp from a character string that has been interpreted using a character template. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

string-expression
: A character expression representing a timestamp value in the format specified by *format-string*. (If *string-expression* is an untyped parameter marker, the type is assumed to be VARCHAR with a maximum length of 254.) The string expression returns a CHAR or a VARCHAR value whose maximum length is not greater than 254 (SQLSTATE 42815). Leading and trailing blanks are removed from *string-expression*, and the resulting substring is interpreted as a timestamp using the format specified by *format-string*. Leading zeros can be omitted from any timestamp components except the year. Blanks can be used in place of leading zeros for these components. For example, with a format string of `'YYYY-MM-DD HH24:MI:SS'`, each of the following strings is an acceptable specification for 9 a.m. on January 1, 2000:

```
'2000-1-01 09:00:00'    (single digit for month)
'2000- 1-01 09:00:00'   (single digit - preceded by a blank -
                         for month)
'2000-1-1 09:00:00'     (single digits for month and day)
'2000-01-01 9:00:00'    (single digit for hour)
'2000-01-01 09:0:0'     (single digits for minutes and seconds)
'2000- 1- 1 09: 0: 0'   (single digit - preceded by a blank -
                         for month, day, minutes, and seconds)
'2000-01-01 09:00:00'   (maximum number of digits for each element)
```

format-string
: A character constant that contains a template for how the string expression is to be interpreted as a timestamp value. The length of the format string must not be greater than 254 (SQLSTATE 42815). Leading and trailing blanks are removed from *format-string*, and the resulting substring must be a valid template for a timestamp value (SQLSTATE 42815). The content of *format-string* can be specified in mixed case.

  Valid format strings are:

```
'YYYY-MM-DD HH24:MI:SS'
```

  where *YYYY* represents a 4-digit year value; *MM* represents a 2-digit month value (01-12; January=01); *DD* represents a 2-digit day of the month value (01-31); *HH24* represents a 2-digit hour of the day value (00-24; If the hour is 24, the minutes and seconds values are zero.); *MI* represents a 2-digit minute value (00-59); and *SS* represents a 2-digit seconds value (00-59).

The result of the function is a timestamp. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Example:

- 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'))
```

TIMESTAMP_ISO

```
►►──TIMESTAMP_ISO──(──expression──)──────────────────────────────────►◄
```

The schema is SYSFUN.

Returns a timestamp value based on a date, time, or timestamp argument. If the argument is a date, it inserts zero for all the time elements. If the argument is a time, it inserts the value of the CURRENT DATE special register for the date elements, and zero for the fractional time element.

The argument must be a date, time, or timestamp, or a valid character string representation of a date, time or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is TIMESTAMP. The result can be null; if the argument is null, the result is the null value.

TIMESTAMPDIFF

```
►►──TIMESTAMPDIFF──(──expression──,──expression──)──────────────────────►◄
```

The schema is SYSFUN.

Returns an estimated number of intervals of the type defined by the first argument, based on the difference between two timestamps.

The first argument can be either INTEGER or SMALLINT. Valid values of interval (the first argument) are:

1	Fractions of a second
2	Seconds
4	Minutes
8	Hours
16	Days
32	Weeks
64	Months
128	Quarters
256	Years

The second argument is the result of subtracting two timestamps and converting the result to CHAR(22). In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

The following assumptions may be used in estimating a difference:
- There are 365 days in a year.
- There are 30 days in a month.
- There are 24 hours in a day.
- There are 60 minutes in an hour.
- There are 60 seconds in a minute.

These assumptions are used when converting the information in the second argument, which is a timestamp duration, to the interval type specified in the first argument. The returned estimate may vary by a number of days. For example, if the number of days (interval 16) is requested for the difference between ’1997-03-01-00.00.00’ and ’1997-02-01-00.00.00’, the result is 30. This is because the difference between the timestamps is 1 month, and the assumption of 30 days in a month applies.

Example:

The following example returns 4277, the number of minutes between two timestamps:

```
TIMESTAMPDIFF(4,CHAR(TIMESTAMP('2001-09-29-11.25.42.483219') -
  TIMESTAMP('2001-09-26-12.07.58.065497')))
```

TO_CHAR

```
►►──TO_CHAR──(──timestamp-expression──,format-string──)──────────────────────►◄
```

The schema is SYSIBM.

The TO_CHAR function returns a character representation of a timestamp that has been formatted using a character template.

TO_CHAR is a synonym for VARCHAR_FORMAT.

Related reference:

- “VARCHAR_FORMAT” on page 438

TO_DATE

```
►►──TO_DATE──(──string-expression──,format-string──)─────────────────────────►◄
```

The schema is SYSIBM.

The TO_DATE function returns a timestamp from a character string that has been interpreted using a character template.

TO_DATE is a synonym for TIMESTAMP_FORMAT.

Related reference:
- "TIMESTAMP_FORMAT" on page 421

TRANSLATE

character string expression:

```
►►──TRANSLATE──(──char-string-exp─────────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────────┬──)────────────►◄
   │                                          ┌─,──' '───┐    │
   └─,──to-string-exp──,──from-string-exp─────┼──────────┼────┘
                                              └─,──pad-char─┘
```

graphic string expression:

```
►►──TRANSLATE──(──graphic-string-exp──,──to-string-exp──,──from-string-exp────►

   ┌─,──' '───┐
►──┼──────────┼──)────────────────────────────────────────────────────────────►◄
   └─,──pad-char─┘
```

The schema is SYSIBM.

The TRANSLATE function returns a value in which one or more characters in a string expression may have been translated into other characters.

The result of the function has the same data type and code page as the first argument. If the first argument is a host variable, the code page of the result is the database code page. The length attribute of the result is the same as that of the first argument. If any specified expression can be NULL, the result can be NULL. If any specified expression is NULL, the result will be NULL.

char-string-exp or *graphic-string-exp*
: A string to be translated.

to-string-exp
: Is a string of characters to which certain characters in the *char-string-exp* will be translated.

: If the *to-string-exp* is not present, and the data type is not graphic, all characters in *char-string-exp* will be in monocase; that is, the characters a-z will be translated to the characters A-Z, and characters with diacritical marks will be translated 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.

from-string-exp
: Is a string of characters which, if found in the *char-string-exp*, will be translated to the corresponding character in the *to-string-exp*. If the *from-string-exp* contains duplicate characters, the first one found will be used, and the duplicates will be ignored. If the *to-string-exp* is longer than the *from-string-exp*, the surplus characters will be ignored. If the *to-string-exp* is present, the *from-string-exp* must also be present.

pad-char-exp
: Is a single character that will be used to pad the *to-string-exp* if the *to-string-exp*

is shorter than the *from-string-exp*. The *pad-char-exp* must have a length attribute of one, or an error is returned. If not present, it will be taken to be a single-byte blank.

The arguments may be either strings of data type CHAR or VARCHAR, or graphic strings of data type GRAPHIC or VARGRAPHIC. They may not have data type LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, or DBCLOB.

With *graphic-string-exp*, only the *pad-char-exp* is optional (if not provided, it will be taken to be the double-byte blank), and each argument, including the pad character, must be of graphic data type.

The result is the string that occurs after translating all the characters in the *char-string-exp* or *graphic-string-exp* that occur in the *from-string-exp* to the corresponding character in the *to-string-exp* or, if no corresponding character exists, to the pad character specified by the *pad-char-exp*.

The code page of the result of TRANSLATE is the same as the code page of the first operand. As of Version 8, if the first operand is a host variable, the code page of the result is the database code page. Each of the other operands is converted to the result code page unless it or the first operand is defined as FOR BIT DATA (in which case there is no conversion).

If the arguments are of data type CHAR or VARCHAR, the corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes. For example, it is not valid to translate a single-byte character to a multi-byte character or vice versa. An error will result if an attempt is made to do this. The *pad-char-exp* must not be the first byte of a valid multi-byte character, or SQLSTATE 42815 is returned. If the *pad-char-exp* is not present, it will be taken to be a single-byte blank.

If only the *char-string-exp* is specified, single-byte characters will be monocased and multi-byte characters will remain unchanged.

Examples:

- Assume the host variable SITE (VARCHAR(30)) has a value of 'Hanauma Bay'.

  ```
  TRANSLATE(:SITE)
  ```

  Returns the value 'HANAUMA BAY'.

  ```
  TRANSLATE(:SITE 'j','B')
  ```

  Returns the value 'Hanauma jay'.

  ```
  TRANSLATE(:SITE,'ei','aa')
  ```

  Returns the value 'Heneume Bey'.

  ```
  TRANSLATE(:SITE,'bA','Bay','%')
  ```

  Returns the value 'HAnAumA bA%'.

  ```
  TRANSLATE(:SITE,'r','Bu')
  ```

  Returns the value 'Hana ma ray'.

TRUNCATE or TRUNC

```
>>-+-TRUNCATE-+--(--expression1--,--expression2--)-------------><
   '-TRUNC----'
```

The schema is SYSIBM. (The SYSFUN version of the TRUNCATE or TRUNC function continues to be available.)

Returns *expression1* truncated to *expression2* places to the right of the decimal point if *expression2* is positive, or to the left of the decimal point if *expression2* is zero or negative.

expression1
: An expression that returns a value of any built-in numeric data type.

expression2
: An expression that returns a small or a large integer. The absolute value of the integer specifies the number of places to the right of the decimal point for the result if *expression2* is not negative, or to left of the decimal point if *expression2* is negative.

  If the absolute value of *expression2* is larger than the number of digits to the left of the decimal point, the result is 0. For example:

  ```
  TRUNCATE(748.58,-4) = 0
  ```

The data type and length attribute of the result are the same as the data type and length attribute of the first argument.

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.

Examples:

- Using the EMPLOYEE table, calculate the average monthly salary for the highest paid employee. Truncate the result two places to the right of the decimal point.

  ```
  SELECT TRUNCATE(MAX(SALARY)/12,2)
    FROM EMPLOYEE;
  ```

  Because the highest paid employee earns $52750.00 per year, the example returns 4395.83.
- Display the number 873.726 truncated 2, 1, 0, -1, and -2 decimal places, respectively.

  ```
  VALUES (
    TRUNC(873.726,2),
    TRUNC(873.726,1),
    TRUNC(873.726,0),
    TRUNC(873.726,-1),
    TRUNC(873.726,-2),
    TRUNC(873.726,-3) );
  ```

  This example returns 873.720, 873.700, 873.000, 870.000, 800.000, and 0.000.

TYPE_ID

```
►►──TYPE_ID──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

The TYPE_ID function returns the internal type identifier of the dynamic data type of the *expression*.

The argument must be a user-defined structured type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

The data type of the result of the function is INTEGER. If *expression* can be null, the result can be null; if *expression* is null, the result is the null value.

The value returned by the TYPE_ID function is not portable across databases. The value may be different, even though the type schema and type name of the dynamic data type are the same. When coding for portability, use the TYPE_SCHEMA and TYPE_NAME functions to determine the type schema and type name.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the internal type identifier of the row that corresponds to the reference.

  ```
  SELECT TASK, WHO_RESPONSIBLE->NAME,
      TYPE_ID(DEREF(WHO_RESPONSIBLE))
    FROM ACTIVITIES
  ```

  The DEREF function is used to return the object corresponding to the row.

TYPE_NAME

```
►►—TYPE_NAME—(—expression—)——————————————————————————————————————►◄
```

The schema is SYSIBM.

The TYPE_NAME function returns the unqualified name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type. (This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.)

The data type of the result of the function is VARCHAR(18). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_SCHEMA function to determine the schema name of the type name returned by TYPE_NAME.

Examples:

- A table hierarchy exists having root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table ACTIVITIES includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE. For each reference in ACTIVITIES, display the type of the row that corresponds to the reference.

  ```
  SELECT TASK, WHO_RESPONSIBLE->NAME,
      TYPE_NAME(DEREF(WHO_RESPONSIBLE)),
      TYPE_SCHEMA(DEREF(WHO_RESPONSIBLE))
    FROM ACTIVITIES
  ```

  The DEREF function is used to return the object corresponding to the row.

TYPE_SCHEMA

```
►►──TYPE_SCHEMA──(──expression──)─────────────────────────────────►◄
```

The schema is SYSIBM.

The TYPE_SCHEMA function returns the schema name of the dynamic data type of the *expression*.

The argument must be a user-defined structured type. This function cannot be used as a source function when creating a user-defined function. Because it accepts any structured data type as an argument, it is not necessary to create additional signatures to support different user-defined types.

The data type of the result of the function is VARCHAR(128). If *expression* can be null, the result can be null; if *expression* is null, the result is the null value. Use the TYPE_NAME function to determine the type name associated with the schema name returned by TYPE_SCHEMA.

Related reference:
- "TYPE_NAME" on page 432

UCASE or UPPER

```
>>──┬─UCASE─┬──(──expression──)──────────────────────────────────────────><
    └─UPPER─┘
```

The schema is SYSIBM. (The SYSFUN version of this function continues to be available for upward compatibility. See Version 5 documentation for a description.)

The UCASE or UPPER function is identical to the TRANSLATE function except that only the first argument (*char-string-exp*) is specified.

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.

Related reference:

- “TRANSLATE” on page 428

VALUE

```
                               ┌──────────────┐
                               ▼              │
►►──VALUE──(──expression─────,expression──┴──)──────────────────────────────►◄
```

The schema is SYSIBM.

The VALUE function returns the first argument that is not null.

VALUE is a synonym for COALESCE.

Related reference:
- "COALESCE" on page 294

VARCHAR

Character to Varchar:

```
>>—VARCHAR—(—character-expression—┬───────────┬—)———————————————————><
                                  └—,—integer—┘
```

Graphic to Varchar:

```
>>—VARCHAR—(—graphic-expression—┬───────────┬—)—————————————————————><
                                └—,—integer—┘
```

Datetime to Varchar:

```
>>—VARCHAR—(—datetime-expression—)——————————————————————————————————><
```

The schema is SYSIBM.

The VARCHAR function returns a varying-length character string representation of:

- A character string, if the first argument is any type of character string
- A graphic string (Unicode databases only), if the first argument is any type of graphic string
- A datetime value, if the argument is a date, time, or timestamp

In a Unicode database, when the output string is truncated part-way through a multiple-byte character:

- If the input was a character string, the partial character is replaced with one or more blanks
- If the input was a graphic string, the partial character is replaced by the empty string

Do not rely on either of these behaviors, because they might change in a future release.

The result of the function is a varying-length character string. If the first argument can be null, the result can be null. If the first argument is null, the result is the null value.

Character to Varchar

character-expression
: An expression whose value must be of a character string data type with a maximum length of 32 672 bytes.

integer
: The length attribute of the resulting varying-length character string. The value must be between 0 and 32 672. If this argument is not specified, the length attribute of the result is the same as the length attribute of the argument.

Graphic to Varchar

graphic-expression
An expression whose value must be of a graphic string data type other than LONG VARGRAPHIC or DBCLOB, and whose maximum length is 16 336 double-byte characters.

integer
The length attribute of the resulting varying-length character string. The value must be between 0 and 32 672. If this argument is not specified, the length attribute of the result is the same as the length attribute of the argument.

Datetime to Varchar

datetime-expression
An expression whose value must be of the DATE, TIME, or TIMESTAMP data type.

Example:

- 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'
```

Related reference:

- “CHAR” on page 288

VARCHAR_FORMAT

```
►►──VARCHAR_FORMAT──(──timestamp-expression──,format-string──)──────────────────►◄
```

The schema is SYSIBM.

The VARCHAR_FORMAT function returns a character representation of a timestamp that has been formatted using a character template. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

timestamp-expression
An expression that results in a timestamp. The argument must be a timestamp or a string representation of a timestamp that is neither a CLOB nor a LONG VARCHAR. (If *string-expression* is an untyped parameter marker, the type is assumed to be TIMESTAMP.) The string expression returns a CHAR or a VARCHAR value whose maximum length is not greater than 254 (SQLSTATE 42815). Leading and trailing blanks are removed from *string-expression*, and the resulting substring is interpreted as a timestamp using the format specified by *format-string*. Leading zeros can be omitted from any timestamp components except the year. Blanks can be used in place of leading zeros for these components. For example, with a format string of `'YYYY-MM-DD HH24:MI:SS'`, each of the following strings is an acceptable specification for 9 a.m. on January 1, 2000:

```
'2000-1-01 09:00:00'    (single digit for month)
'2000- 1-01 09:00:00'   (single digit - preceded by a blank - for month)
'2000-1-1 09:00:00'     (single digits for month and day)
'2000-01-01 9:00:00'    (single digit for hour)
'2000-01-01 09:0:0'     (single digits for minutes and seconds)
'2000- 1- 1 09: 0: 0'   (single digit - preceded by a blank - for month,
                         day, minutes, and seconds)
'2000-01-01 09:00:00'   (maximum number of digits for each element)
```

format-string
A character constant that contains a template for how the result is to be formatted. The length of the format string must not be greater than 254 (SQLSTATE 42815). Leading and trailing blanks are removed from *format-string*, and the resulting substring must be a valid template for a timestamp value (SQLSTATE 42815). The content of *format-string* can be specified in mixed case.

Valid format strings are:

```
'YYYY-MM-DD HH24:MI:SS'
```

where *YYYY* represents a 4-digit year value; *MM* represents a 2-digit month value (01-12; January=01); *DD* represents a 2-digit day of the month value (01-31); *HH24* represents a 2-digit hour of the day value (00-24; If the hour is 24, the minutes and seconds values are zero.); *MI* represents a 2-digit minute value (00-59); and *SS* represents a 2-digit seconds value (00-59).

The result of the function is a varying-length character string containing a formatted timestamp expression. The format string also determines the length attribute and the actual length of the result. If format-string is 'YYYY-MM-DD HH24:MI:SS', the length attribute is 19. The result is 19 characters of the form:

```
YYYY-MM-DD HH:MI:SS
```

For example, with format `'YYYY-MM-DD HH24:MI:SS'` and a time and date of 10 a.m. on January 1, 2000, the following is returned:

```
'2000-01-01 10:00:00'
```

Even though the values for month and day only require a single digit, in this example, each significant digit is preceded with a leading zero. And, even though the minutes and seconds values are both zero, the maximum number of digits are used for each, and '00' is returned for each of these parts in the result.

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 CCSID of the result is the SBCS CCSID of the system.

Example:

- Display the table names and creation timestamps for all of the system tables whose name starts with 'SYSU'.

  ```
  SELECT VARCHAR(name, 20) AS TABLE_NAME,
    VARCHAR_FORMAT(ctime, 'YYYY-MM-DD HH24:MI:SS') AS CREATION_TIME
    FROM SYSCAT.TABLES
    WHERE name 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
  ```

VARGRAPHIC

Graphic to Vargraphic:

```
►►──VARGRAPHIC──(──graphic-expression──┬─────────────┬──)──────────────────────►◄
                                        └─,──integer──┘
```

Character to Vargraphic:

```
►►──VARGRAPHIC──(──character-expression──)─────────────────────────────────────►◄
```

Datetime to Vargraphic:

```
►►──VARGRAPHIC──(──datetime-expression──)──────────────────────────────────────►◄
```

The schema is SYSIBM.

The VARGRAPHIC function returns a varying-length graphic string representation of:

- A graphic string, if the first argument is any type of graphic string
- A character string, converting single-byte characters to double-byte characters, if the first argument is any type of character string
- A datetime value (Unicode databases only), if the argument is a date, time, or timestamp

In a Unicode database, if a supplied argument is a character string, it is first converted to a graphic string before the function is executed. When the output string is truncated, such that the last character is a high surrogate, that surrogate is converted to the blank character (X'0020'). Do not rely on this behavior, because it might change in a future release.

The result of the function is a varying-length graphic string (VARGRAPHIC data type). If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

Graphic to Vargraphic

graphic-expression
An expression that returns a value that is a graphic string.

integer
An integer value specifying the length attribute of the resulting VARGRAPHIC data type. The value must be between 0 and 16 336. If a value is not specified, the length attribute of the result is the same as the length attribute of the first argument.

If the length of the graphic expression is greater than the length attribute of the result, the result is truncated. A warning is returned (SQLSTATE 01004), unless the truncated characters were all blanks, and the graphic expression was not a long string (LONG VARGRAPHIC or DBCLOB).

Character to Vargraphic

character-expression
An expression whose value must be of a character string data type other than LONG VARCHAR or CLOB, and whose maximum length is 16 336 bytes.

The length attribute of the result is equal to the length attribute of the argument.

Each single-byte character in *character-expression* is converted to its equivalent double-byte representation or to the double-byte substitution character in the result. Each double-byte character in *character-expression* is mapped without additional conversion. If the first byte of a double-byte character appears as the last byte of *character-expression*, it is converted to the double-byte substitution character. The sequential order of the characters in *character-expression* is preserved.

For a Unicode database, this function converts the character string from the code page of the operand to UCS-2. Every character of the operand, including double-byte characters, is converted. If a value for the second argument is provided, it specifies the required length of the resulting string (in UCS-2 characters).

The conversion to double-byte code points by the VARGRAPHIC function is based on the code page of the operand.

Double-byte characters of the operand are not converted. All other characters are converted to their corresponding double-byte equivalents. If there is no corresponding double-byte equivalent, the double-byte substitution character for the code page is used.

No warning or error code is generated if one or more double-byte substitution characters are returned in the result.

Datetime to Vargraphic

datetime-expression
An expression whose value must be of the DATE, TIME, or TIMESTAMP data type.

Related reference:

- “GRAPHIC” on page 347
- Appendix N, “Japanese and traditional-Chinese extended UNIX code (EUC) considerations,” on page 755

WEEK

```
►►──WEEK──(──expression──)────────────────────────────────────────►◄
```

Returns the week of the year of the argument as an integer value in range 1-54. The week starts with Sunday.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

WEEK_ISO

```
►►──WEEK_ISO──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSFUN.

Returns the week of the year of the argument as an integer value in the range 1-53. The week starts with Monday and always includes 7 days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. It is therefore possible to have up to 3 days at the beginning of a year appear in the last week of the previous year. Conversely, up to 3 days at the end of a year may appear in the first week of the next year.

The argument must be a date, timestamp, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is INTEGER. The result can be null; if the argument is null, the result is the null value.

Example:

The following list shows examples of the result of WEEK_ISO and DAYOFWEEK_ISO.

```
DATE       WEEK_ISO    DAYOFWEEK_ISO
---------- ----------- -------------
1997-12-28          52             7
1997-12-31           1             3
1998-01-01           1             4
1999-01-01          53             5
1999-01-04           1             1
1999-12-31          52             5
2000-01-01          52             6
2000-01-03           1             1
```

YEAR

```
►►──YEAR──(──expression──)──────────────────────────────────────────►◄
```

The schema is SYSIBM.

The YEAR function returns the year part of a value.

The argument must be a date, timestamp, date duration, timestamp duration, or a valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. In a Unicode database, if a supplied argument is a graphic string, it is first converted to a character string before the function is executed.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The other rules depend on the data type of the argument specified:

- If the argument is a date, timestamp, or valid string representation of a date or timestamp:
  - The result is the year part of the value, which is an integer between 1 and 9 999.
- If the argument is a date duration or timestamp duration:
  - The result is the year part of the value, which is an integer between −9 999 and 9 999. A nonzero result has the same sign as the argument.

Examples:

- Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

  ```
  SELECT * FROM PROJECT
    WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
  ```
- Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

  ```
  SELECT * FROM PROJECT
    WHERE YEAR(PRENDATE - PRSTDATE) < 1
  ```

Table functions

A table function can be used only in the FROM clause of a statement. Table functions return columns of a table, resembling a table created through a simple CREATE TABLE statement. Table functions can be qualified with a schema name.

All monitoring table functions use a separate instance connection, which is different from the one the current session uses. Therefore, only default database manager monitor switches are in effect. Any monitor switches that are turned on or off dynamically from the current session or application are not effective.

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.

The argument or arguments of a procedure are individual scalar values, which can be of different types and can have different meanings. The arguments can be used to pass values into the procedure, receive return values from the procedure, or both.

User-defined procedures are procedures that are registered to a database in SYSCAT.ROUTINES, using the CREATE PROCEDURE statement. One such set of functions is provided with the database manager, in a schema called SYSFUN, and another in a schema called SYSPROC.

Procedures can be qualified with the schema name.

User-defined functions

```
>>─function-name─(─┬─────────────────┬─)──────────────────────────────><
                   │ ┌─,──────────┐  │
                   │ ▼            │  │
                   └───expression─┴──┘
```

User-defined functions (UDFs) are extensions or additions to the existing built-in functions of the SQL language. A user-defined function can be a scalar function, which returns a single value each time it is called; a column function, which is passed a set of like values and returns a single value for the set; a row function, which returns one row; or a table function, which returns a table.

A number of user-defined functions are provided in the SYSFUN and SYSPROC schemas.

A UDF can be a column function only if it is sourced on an existing column function. A UDF is referenced by means of a qualified or unqualified function name, followed by parentheses enclosing the function arguments (if any). A user-defined column or scalar function registered with the database can be referenced in the same contexts in which any built-in function can appear. A user-defined row function can be referenced only implicitly when registered as a transform function for a user-defined type. A user-defined table function registered with the database can be referenced only in the FROM clause of a SELECT statement.

Function arguments must correspond in number and position to the parameters specified for the user-defined function when it was registered with the database. In addition, the arguments must be of data types that are promotable to the data types of the corresponding defined parameters.

The result of the function is specified in the RETURNS clause. The RETURNS clause, defined when the UDF was registered, determines whether or not a function is a table function. If the RETURNS NULL ON NULL INPUT clause is specified (or defaulted to) when the function is registered, the result is null if any argument is null. In the case of table functions, this is interpreted to mean a return table with no rows (that is, an empty table).

Following are some examples of user-defined functions:

- A scalar UDF called ADDRESS extracts the home address from resumes stored in script format. The ADDRESS function expects a CLOB argument and returns a VARCHAR(4000) value:
  ```
  SELECT EMPNO, ADDRESS(RESUME) FROM EMP_RESUME
    WHERE RESUME_FORMAT = 'SCRIPT'
  ```
- Table T2 has a numeric column A. Invoking the scalar UDF called ADDRESS from the previous example:
  ```
  SELECT ADDRESS(A) FROM T2
  ```

  raises an error (SQLSTATE 42884), because no function with a matching name and with a parameter that is promotable from the argument exists.
- A table UDF called WHO returns information about the sessions on the server machine that were active at the time that the statement is executed. The WHO function is invoked from within a FROM clause that includes the keyword

TABLE and a mandatory correlation variable. The column names of the WHO() table were defined in the CREATE FUNCTION statement.

```
SELECT ID, START_DATE, ORIG_MACHINE
  FROM TABLE( WHO() ) AS QQ
  WHERE START_DATE LIKE 'MAY%'
```

Related reference:

- “Subselect” on page 450
- “CREATE FUNCTION statement” in the *SQL Reference, Volume 2*

Chapter 4. Queries

SQL queries

A *query* specifies a result table. A query is a component of certain SQL statements. The three forms of a query are:

- subselect
- fullselect
- select-statement.

Authorization

For each table, view, or nickname referenced in the query, the authorization ID of the statement must have at least one of the following:

- SYSADM or DBADM authority
- CONTROL privilege
- SELECT privilege.

Group privileges, with the exception of PUBLIC, are not checked for queries that are contained in static SQL statements.

For nicknames, authorization requirements of the data source for the object referenced by the nickname are applied when the query is processed. The authorization ID of the statement may be mapped to a different authorization ID at the data source.

Related reference:

- "SELECT INTO statement" in the *SQL Reference, Volume 2*

Subselect

```
►►──select-clause──from-clause──┬──────────────┬──┬─────────────────┬──────────────►
                                └─where-clause─┘  └─group-by-clause─┘

►──┬───────────────┬──┬─────────────────┬──┬────────────────────┬──────────────────►◄
   └─having-clause─┘  └─order-by-clause─┘  └─fetch-first-clause─┘
```

The *subselect* is a component of the fullselect.

A subselect specifies a result table derived from the tables, views or nicknames identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is only a way of describing the subselect. The method used to perform the derivation may be quite different from this description.)

The clauses of the subselect are processed in the following sequence:
1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause
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 a materialized query table.
- Unless the subselect is enclosed in parenthesis.

For example, the following is not valid (SQLSTATE 428FJ):

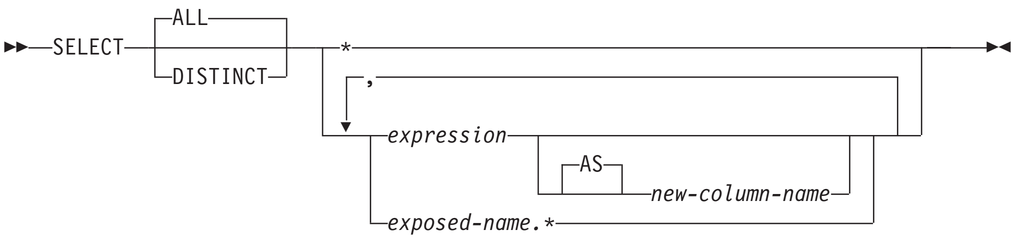
```
SELECT * FROM T1
   ORDER BY C1
UNION
SELECT * FROM T2
   ORDER BY C1
```

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.

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 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 LONG VARCHAR, LONG VARGRAPHIC, DATALINK, LOB type, distinct type on any of these types, or structured type. DISTINCT may be used more than once in a subselect. This includes SELECT DISTINCT, the use of DISTINCT in a column function of the select list or HAVING clause, and subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value of the second. For determining duplicates, two null values are considered equal.

Select list notation:

* Represents a list of names that identify the columns of table R. The first name in the list identifies the first column of R, the second name identifies the second column of R, and so on.

The list of names is established when the program containing the SELECT clause is bound. Hence * (the asterisk) does not identify any columns that have been added to a table after the statement containing the table reference has been bound.

expression
Specifies the values of a result column. Can be any expression that is a valid SQL language element, but commonly includes column names. Each column name used in the select list must unambiguously identify a column of R.

new-column-name or **AS** *new-column-name*
Names or renames the result column. The name must not be qualified and does not have to be unique. Subsequent usage of column-name is limited as follows:

- A new-column-name specified in the AS clause can be used in the order-by-clause, provided the name is unique.
- A new-column-name specified in the AS clause of the select list cannot be used in any other clause within the subselect (where-clause, group-by-clause or having-clause).
- A new-column-name specified in the AS clause cannot be used in the update-clause.

Select list notation:

- A new-column-name specified in the AS clause is known outside the fullselect of nested table expressions, common table expressions and CREATE VIEW.

*name.**
: Represents the list of names that identify the columns of the result table identified by *exposed-name*. The *exposed-name* may be a table name, view name, nickname, or correlation name, and must designate a table, view or nickname named in the FROM clause. The first name in the list identifies the first column of the table, view or nickname, the second name in the list identifies the second column of the table, view or nickname, and so on.

  The list of names is established when the statement containing the SELECT clause is bound. Therefore, * does not identify any columns that have been added to a table after the statement has been bound.

  The number of columns in the result of SELECT is the same as the number of expressions in the operational form of the select list (that is, the list established when the statement is prepared), and cannot exceed 500 for a 4K page size or 1012 for an 8K, 16K, or 32K page size.

Limitations on string columns

For limitations on the select list, see "Restrictions Using Varying-Length Character Strings".

Applying the select list

Some of the results of applying the select list to R depend on whether or not GROUP BY or HAVING is used. The results are described in two separate lists:

If GROUP BY or HAVING is used:

- An expression *X* (not a column function) used in the select list must have a GROUP BY clause with:
  - a *grouping-expression* in which each column-name unambiguously identifies a column of R (see "group-by-clause" on page 463) or
  - each column of R referenced in *X* as a separate *grouping-expression*.
- The select list is applied to each group of R, and the result contains as many rows as there are groups in R. When the select list is applied to a group of R, that group is the source of the arguments of the column functions in the select list.

If neither GROUP BY nor HAVING is used:

- Either the select list must not include any column functions, or each *column-name* in the select list must be specified within a column function or must be a correlated column reference.
- If the select does not include column functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of column functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns: Result columns do not allow null values if they are derived from:

- A column that does not allow null values
- A constant

- The COUNT or COUNT_BIG function
- A host variable that does not have an indicator variable
- A scalar function or expression that does not include an operand that allows nulls.

Result columns allow null values if they are derived from:
- Any column function except COUNT or COUNT_BIG
- A column that allows null values
- A scalar function or expression that includes an operand that allows nulls
- A NULLIF function with arguments containing equal values.
- A host variable that has an indicator variable.
- A result of a set operation if at least one of the corresponding items in the select list is nullable.
- An arithmetic expression or view column that is derived from an arithmetic expression and the database is configured with DFT_SQLMATHWARN set to yes
- A dereference operation.

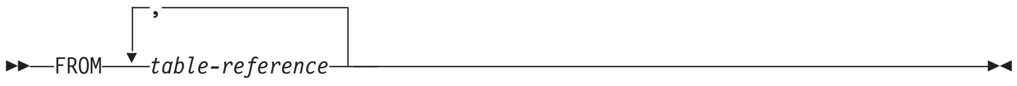
Names of result columns:
- If the AS clause is specified, the name of the result column is the name specified on the AS clause.
- If the AS clause is not specified and the result column is derived from a column, then the result column name is the unqualified name of that column.
- If the AS clause is not specified and the result column is derived using a dereference operation, then the result column name is the unqualified name of the target column of the dereference operation.
- All other result column names are unnamed. 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.
an integer constant	INTEGER.
a decimal constant	DECIMAL, with the precision and scale of the constant.
a floating-point constant	DOUBLE.
the name of any numeric variable	the same as the data type of the variable, with the same precision and scale for DECIMAL variables.
a hexadecimal constant representing n bytes	VARCHAR(n); the code page is the database code page.
the name of any string column	the same as the data type of the column, with the same length attribute.

When the expression is ...	The data type of the result column is ...
the name of any string variable	the same as the data type of the variable, with the same length attribute; if the data type of the variable is not identical to an SQL data type (for example, a NUL-terminated string in C), the result column is a varying-length string.
a character string constant of length *n*	VARCHAR(*n*).
a graphic string constant of length *n*	VARGRAPHIC(*n*).
the name of a datetime column	the same as the data type of the column.
the name of a user-defined type column	the same as the data type of the column.
the name of a reference type column	the same as the data type of the column.

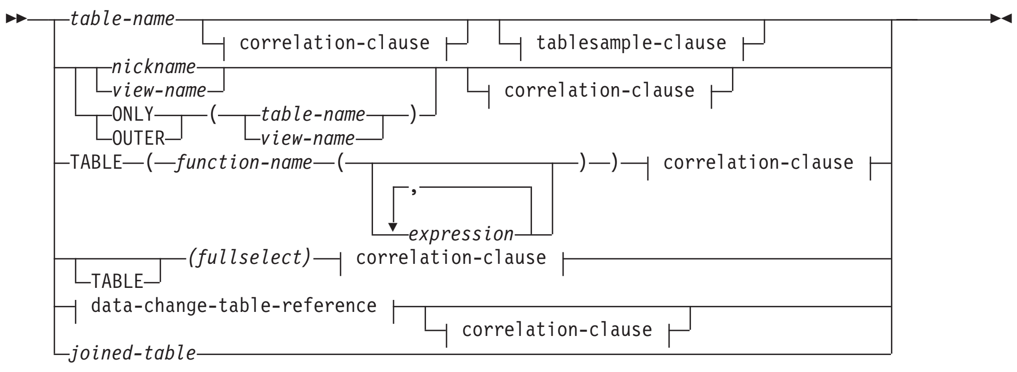
from-clause



The FROM clause specifies an intermediate result table.

If one table-reference is specified, the intermediate result table is simply the result of that table-reference. If more than one table-reference is specified, the intermediate result table consists of all possible combinations of the rows of the specified table-references (the Cartesian product). Each row of the result is a row from the first table-reference concatenated with a row from the second table-reference, concatenated in turn with a row from the third, and so on. The number of rows in the result is the product of the number of rows in all the individual table-references. For a description of *table-reference*, see “table-reference.”

table-reference



data-change-table-reference:

```
├──┬─┬─FINAL─┬──TABLE──(──insert-statement──)──────────┬──────────────────────────┤
   │ └─NEW───┘                                          │
   ├─┬─FINAL─┬──TABLE──(──searched-update-statement──)──┤
   │ ├─NEW───┤                                          │
   │ └─OLD───┘                                          │
   └─OLD TABLE──(──searched-delete-statement──)─────────┘
```

correlation-clause:

```
   ┌─AS─┐
├──┴────┴──correlation-name──┬─────────────────────────────┬──────────────────────┤
                             │      ┌─,───────────┐        │
                             │      ▼             │        │
                             └──(───┴column-name──┴──)─────┘
```

tablesample-clause:

```
├──TABLESAMPLE──┬─BERNOULLI─┬──(──numeric-expression1──)──────────────────────────►
                └─SYSTEM────┘

►──┬─────────────────────────────────────────────┬────────────────────────────────┤
   └─REPEATABLE──(──numeric-expression2──)───────┘
```

Each *table-name*, *view-name* or *nickname* specified as a table-reference must identify an existing table, view or nickname at the application server or the *table-name* of a common table expression defined preceding the fullselect containing the table-reference. If the *table-name* references a typed table, the name denotes the UNION ALL of the table with all its subtables, with only the columns of the *table-name*. Similarly, if the *view-name* references a typed view, the name denotes the UNION ALL of the view with all its subviews, with only the columns of the *view-name*.

The use of ONLY(*table-name*) or ONLY(*view-name*) means that the rows of the proper subtables or subviews are not included. If the *table-name* used with ONLY does not have subtables, then ONLY(*table-name*) is equivalent to specifying *table-name*. If the *view-name* used with ONLY does not have subviews, then ONLY(*view-name*) is equivalent to specifying *view-name*.

The use of OUTER(*table-name*) or OUTER(*view-name*) represents a virtual table. If the *table-name* or *view-name* used with OUTER does not have subtables or subviews, then specifying OUTER is equivalent to not specifying OUTER. OUTER(*table-name*) is derived from *table-name* as follows:

- The columns include the columns of *table-name* followed by the additional columns introduced by each of its subtables (if any). The additional columns are added on the right, traversing the subtable hierarchy in depth-first order. Subtables that have a common parent are traversed in creation order of their types.
- The rows include all the rows of *table-name* and all the rows of its subtables. Null values are returned for columns that are not in the subtable for the row.

The previous points also apply to OUTER(*view-name*), substituting *view-name* for *table-name* and subview for subtable.

The use of ONLY or OUTER requires the SELECT privilege on every subtable of *table-name* or subview of *view-name*.

Each *function-name* together with the types of its arguments, specified as a table reference must resolve to an existing table function at the application server.

A fullselect in parentheses followed by a correlation name is called a *nested table expression*.

A *joined-table* specifies an intermediate result set that is the result of one or more join operations. For more information, see "joined-table" on page 461.

The exposed names of all table references should be unique. An exposed name is:
- A *correlation-name*,
- A *table-name* that is not followed by a *correlation-name*,
- A *view-name* that is not followed by a *correlation-name*,
- A *nickname* that is not followed by a *correlation-name*,
- An *alias-name* that is not followed by a *correlation-name*.

Each *correlation-name* is defined as a designator of the immediately preceding *table-name*, *view-name*, *nickname*, *function-name* reference or nested table expression. Any qualified reference to a column for a table, view, table function or nested table expression must use the exposed name. If the same table name, view or nickname name is specified twice, at least one specification should be followed by a *correlation-name*. The *correlation-name* is used to qualify references to the columns of the table, view or nickname. When a *correlation-name* is specified, *column-names* can also be specified to give names to the columns of the *table-name*, *view-name*, *nickname*, *function-name* reference or nested table expression.

In general, table functions and nested table expressions can be specified on any from-clause. Columns from the table functions and nested table expressions can be referenced in the select list and in the rest of the subselect using the correlation name which must be specified. The scope of this correlation name is the same as correlation names for other table, view or nickname in the FROM clause. A nested table expression can be used:
- In place of a view to avoid creating the view (when general use of the view is not required)
- When the desired result table is based on host variables

An expression in the select list of a nested table expression that is referenced within, or is the target of, a data change statement within a fullselect is only valid when it does not include:
- A function that reads or modifies SQL data
- A function that is non-deterministic
- A function that has external action
- An OLAP function

If a view is referenced directly in, or as the target of a nested table expression in a data change statement within a FROM clause, the view must either be symmetric (have WITH CHECK OPTION specified) or satisfy the restriction for a WITH CHECK OPTION view.

If the target of a data change statement within a FROM clause is a nested table expression, the modified rows are not requalified, WHERE clause predicates are not re-evaluated, and ORDER BY or FETCH FIRST operations are not redone.

The optional tablesample-clause can be used to obtain a random subset (a sample) of the rows from the specified *table-name*, rather than the entire contents of that *table-name*, for this query. This sampling is in addition to any predicates that are specified in the *where-clause*. Unless the optional REPEATABLE clause is specified, each execution of the query will usually yield a different sample, except in degenerate cases where the table is so small relative to the sample size that any sample must return the same rows. The size of the sample is controlled by the *numeric-expression1* in parentheses, representing an approximate percentage (P) of the table to be returned. The method by which the sample is obtained is specified after the TABLESAMPLE keyword, and can be either BERNOULLI or SYSTEM. For both methods, the exact number of rows in the sample may be different for each execution of the query, but on average should be approximately P percent of the table, before any predicates further reduce the number of rows.

The *table-name* must be a stored table. It can be a materialized query table (MQT) name, but not a subselect or table expression for which an MQT has been defined, because there is no guarantee that the database manager will route to the MQT for that subselect.

Semantically, sampling of a table occurs before any other query processing, such as applying predicates or performing joins. Repeated accesses of a sampled table within a single execution of a query (such as in a nested-loop join or a correlated subquery) will return the same sample. More than one table may be sampled in a query.

BERNOULLI sampling considers each row individually. It includes each row in the sample with probability P/100 (where P is the value of *numeric-expression1*), and excludes each row with probability 1 - P/100, independently of the other rows. So if the *numeric-expression1* evaluated to the value 10, representing a ten percent sample, each row would be included with probability 0.1, and excluded with probability 0.9.

SYSTEM sampling permits the database manager to determine the most efficient manner in which to perform the sampling. In most cases, SYSTEM sampling applied to a *table-name* means that each page of *table-name* is included in the sample with probability P/100, and excluded with probability 1 - P/100. All rows on each page that is included qualify for the sample. SYSTEM sampling of a *table-name* generally executes much faster than BERNOULLI sampling, because fewer data pages need to be retrieved; however, it can often yield less accurate estimates for aggregate functions (SUM(SALES), for example), especially if the rows of *table-name* are clustered on any columns referenced in that query. The optimizer may in certain circumstances decide that it is more efficient to perform SYSTEM sampling as if it were BERNOULLI sampling, for example when a predicate on *table-name* can be applied by an index and is much more selective than the sampling rate P.

The *numeric-expression1* specifies the size of the sample to be obtained from *table-name*, expressed as a percentage. It must be a constant numeric expression that cannot contain columns, parameter markers, or host variables. The expression must evaluate to a positive number that is less than or equal to 100, but can be between 1 and 0. For example, a value of 0.01 represents one one-hundredth of a percent, meaning that 1 row in 10 000 would be sampled, on average. A *numeric-expression1* that evaluates to 100 is handled as if the tablesample-clause were not specified. If *numeric-expression1* evaluates to the null value, or to a value that is greater than 100 or less than 0, an error is returned (SQLSTATE 2202H).

It is sometimes desirable for sampling to be repeatable from one execution of the query to the next; for example, during regression testing or query ″debugging″. This can be accomplished by specifying the REPEATABLE clause. The REPEATABLE clause requires the specification of a *numeric-expression2* in parentheses, which serves the same role as the seed in a random number generator. Adding the REPEATABLE clause to the tablesample-clause of any *table-name* ensures that repeated executions of that query (using the same value for *numeric-expression2*) return the same sample, assuming, of course, that the data itself has not been updated, reorganized, or repartitioned. To guarantee that the same sample of *table-name* is used across multiple queries, use of a global temporary table is recommended. Alternatively, the multiple queries could be combined into one query, with multiple references to a sample that is defined using the WITH clause.

Following are some examples:

Example 1: Request a 10% Bernoulli sample of the Sales table for auditing purposes.

```
SELECT * FROM Sales
  TABLESAMPLE BERNOULLI(10)
```

Example 2: Compute the total sales revenue in the Northeast region for each product category, using a random 1% SYSTEM sample of the Sales table. The semantics of SUM are for the sample itself, so to extrapolate the sales to the entire Sales table, the query must divide that SUM by the sampling rate (0.01).

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

Example 3: Using the REPEATABLE clause, modify the previous query to ensure that the same (yet random) result is obtained each time the query is executed. (The value of the constant enclosed by parentheses is arbitrary.)

```
SELECT SUM(Sales.Revenue) / (0.01)
  FROM Sales TABLESAMPLE SYSTEM(1) REPEATABLE(3578231)
  WHERE Sales.RegionName = 'Northeast'
  GROUP BY Sales.ProductCategory
```

Table function references

In general, a table function, together with its argument values, can be referenced in the FROM clause of a SELECT in exactly the same way as a table or view. There are, however, some special considerations which apply.

- Table Function Column Names

  Unless alternate column names are provided following the *correlation-name*, the column names for the table function are those specified in the RETURNS clause of the CREATE FUNCTION statement. This is analogous to the names of the columns of a table, which are defined in the CREATE TABLE statement.

- Table Function Resolution

  The arguments specified in a table function reference, together with the function name, are used by an algorithm called *function resolution* to determine the exact function to be used. This is no different from what happens with other functions (such as scalar functions) that are used in a statement.

- Table Function Arguments

  As with scalar function arguments, table function arguments can in general be any valid SQL expression. The following examples are valid syntax:

```
Example 1:  SELECT c1
            FROM TABLE( tf1('Zachary') ) AS z
            WHERE c2 = 'FLORIDA';

Example 2:  SELECT c1
            FROM TABLE( tf2 (:hostvar1, CURRENT DATE) ) AS z;

Example 3:  SELECT c1
            FROM t
            WHERE c2 IN
                     (SELECT c3 FROM
                      TABLE( tf5(t.c4) ) AS z  -- correlated reference
                     )                         -- to previous FROM clause
```

- Table Functions That Modify SQL Data

  Table functions that are specified with the MODIFIES SQL DATA option can only be used as the last table reference in a *select-statement*, *common-table-expression*, or RETURN statement that is a subselect, a SELECT INTO, or a *row-fullselect* in a SET statement. Only one table function is allowed in one FROM clause, and the table function arguments must be correlated to all other table references in the subselect (SQLSTATE 429BL). The following examples have valid syntax for a table function with the MODIFIES SQL DATA property:

```
Example 1:  SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 2:  SELECT c1
            FROM t1, t2, TABLE( tfmod(t1.c1, t2.c1) ) AS z

Example 3:  SET var =
            (SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 4:  RETURN SELECT c1
            FROM TABLE( tfmod('Jones') ) AS z

Example 5:  WITH v1(c1) AS
            (SELECT c1
            FROM TABLE( tfmod(:hostvar1) ) AS z)
            SELECT c1
            FROM v1, t1 WHERE v1.c1 = t1.c1
```

Correlated references in table-references

Correlated references can be used in nested table expressions or as arguments to table functions. The basic rule that applies for both these cases is that the correlated reference must be from a *table-reference* at a higher level in the hierarchy of subqueries. This hierarchy includes the table-references that have already been resolved in the left-to-right processing of the FROM clause. For nested table expressions, the TABLE keyword must appear before the fullselect. So the following examples are valid syntax:

```
Example 1:  SELECT t.c1, z.c5
            FROM t, TABLE( tf3(t.c2) ) AS z      -- t precedes tf3
            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 tf3
            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,
                 TABLE (SELECT AVG(e.salary) AS avgsal,
                               COUNT(*) AS empcount
                        FROM employee e           -- department precedes
                        WHERE e.workdept=d.deptno -- and TABLE is
                       ) AS empinfo;              -- specified, so
                                                  -- d.deptno is known
```

But the following examples are not valid:

```
Example 4:  SELECT t.c1, z.c5
            FROM TABLE( tf6(t.c2) ) AS z, t  -- cannot resolve t in t.c2!
            WHERE t.c3 = z.c4;               -- compare to Example 1 above.

Example 5:  SELECT a.c1, b.c5
            FROM TABLE( tf7a(b.c2) ) AS a, TABLE( tf7b(a.c6) ) AS b
            WHERE a.c3 = b.c4;               -- cannot resolve b in b.c2!

Example 6:  SELECT d.deptno, d.deptname,
                   empinfo.avgsal, empinfo.empcount
            FROM department d,
                 (SELECT AVG(e.salary) AS avgsal,
                         COUNT(*) AS empcount
                        FROM employee e           -- department precedes
                        WHERE e.workdept=d.deptno -- but TABLE is not
                       ) AS empinfo;              -- specified, so
                                                  -- d.deptno is unknown
```

Data change table references

A *data-change-table-reference* clause specifies an intermediate result table. This table is based on the rows that are directly changed by the searched UPDATE, searched DELETE, or INSERT statement that is included in the clause. A *data-change-table-reference* can be specified as the only *table-reference* in the FROM clause of the outer fullselect that is used in a *select-statement*, a SELECT INTO statement, or a common table expression. A *data-change-table-reference* can be specified as the only table reference in the only fullselect in a SET Variable statement (SQLSTATE 428FL). The target table or view of the data change statement is considered to be a table or view that is referenced in the query; therefore, the authorization ID of the query must have SELECT privilege on that target table or view.

The target of the UPDATE, DELETE, or INSERT statement cannot be a temporary view defined in a common table expression (SQLSTATE 42807).

FINAL TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they appear at the completion of the data change statement. If there are AFTER triggers or referential constraints that result in further operations on the table that is the target of the SQL data change statement, an error is returned (SQLSTATE 57058, SQLSTATE 560C6). If the target of the SQL data change statement is a view that is defined with an INSTEAD OF trigger for the type of data change, an error is returned (SQLSTATE 428G3).

NEW TABLE
Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement prior to the application of referential constraints and AFTER triggers. Data in the target table at the

completion of the statement might not match the data in the intermediate result table because of additional processing for referential constraints and AFTER triggers.

OLD TABLE
: Specifies that the rows of the intermediate result table represent the set of rows that are changed by the SQL data change statement as they existed prior to the application of the data change statement.

(searched-update-statement)
: Specifies a searched UPDATE statement. A WHERE clause or a SET clause in the UPDATE statement cannot contain correlated references to columns outside of the UPDATE statement.

(searched-delete-statement)
: Specifies a searched DELETE statement. A WHERE clause in the DELETE statement cannot contain correlated references to columns outside of the DELETE statement.

(insert-statement)
: Specifies an INSERT statement. A fullselect in the INSERT statement cannot contain correlated references to columns outside of the fullselect of the INSERT statement.

The content of the intermediate result table for a *data-change-table-reference* is determined when the cursor opens. The intermediate result table contains all manipulated rows, including all the columns in the specified target table or view. All the columns of the target table or view for an SQL data change statement are accessible using the column names from the target table or view. If an INCLUDE clause was specified within a data change statement, the intermediate result table will contain these additional columns.

joined-table

```
                            ┌─INNER─────┐
►►──┬─table-reference───────┼───────────┼──JOIN──table-reference──ON──join-condition──┬──────►◄
    │                       └─┤ outer ├─┘                                              │
    └─(──joined-table──)───────────────────────────────────────────────────────────────┘
```

outer:

```
            ┌─OUTER─┐
├───┬─LEFT──┬──┴───────┴──────────────────────────────────────────────────────────────┤
    ├─RIGHT─┤
    └─FULL──┘
```

A *joined table* specifies an intermediate result table that is the result of either an inner join or an outer join. The table is derived by applying one of the join operators: INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER to its operands.

Inner joins can be thought of as the cross product of the tables (combine each row of the left table with every row of the right table), keeping only the rows where the join condition is true. The result table may be missing rows from either or both of the joined tables. Outer joins include the inner join and preserve these missing rows. There are three types of outer joins:

- ***left outer join*** includes rows from the left table that were missing from the inner join.
- ***right outer join*** includes rows from the right table that were missing from the inner join.
- ***full outer join*** includes rows from both the left and right tables that were missing from the inner join.

If a join-operator is not specified, INNER is implicit. The order in which multiple joins are performed can affect the result. Joins can be nested within other joins. The order of processing for joins is generally from left to right, but based on the position of the required join-condition. Parentheses are recommended to make the order of nested joins more readable. For example:

```
TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1
  RIGHT JOIN TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1
    ON TB1.C1=TB3.C1
```

is the same as:

```
(TB1 LEFT JOIN TB2 ON TB1.C1=TB2.C1)
  RIGHT JOIN (TB3 LEFT JOIN TB4 ON TB3.C1=TB4.C1)
    ON TB1.C1=TB3.C1
```

A joined table can be used in any context in which any form of the SELECT statement is used. A view or a cursor is read-only if its SELECT statement includes a joined table.

A *join-condition* is a *search-condition* except that:
- it cannot contain any subqueries, scalar or otherwise
- it cannot include any dereference operations or the DEREF function where the reference value is other than the object identifier column.
- it cannot include an SQL function
- any column referenced in an expression of the *join-condition* must be a column of one of the operand tables of the associated join (in the scope of the same joined-table clause)
- any function referenced in an expression of the *join-condition* of a full outer join must be deterministic and have no external action.

An error occurs if the join condition does not comply with these rules (SQLSTATE 42972).

Column references are resolved using the rules for resolution of column name qualifiers. The same rules that apply to predicates apply to join conditions.

Join operations

A *join-condition* specifies pairings of T1 and T2, where T1 and T2 are the left and right operand tables of the JOIN operator of the *join-condition*. For all possible combinations of rows of T1 and T2, a row of T1 is paired with a row of T2 if the *join-condition* is true. When a row of T1 is joined with a row of T2, a row in the result consists of the values of that row of T1 concatenated with the values of that row of T2. The execution might involve the generation of a null row. The null row of a table consists of a null value for each column of the table, regardless of whether the columns allow null values.

The following summarizes the result of the join operations:

- The result of T1 INNER JOIN T2 consists of their paired rows where the join-condition is true.
- The result of T1 LEFT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T2 allow null values.
- The result of T1 RIGHT OUTER JOIN T2 consists of their paired rows where the join-condition is true and, for each unpaired row of T2, the concatenation of that row with the null row of T1. All columns derived from T1 allow null values.
- The result of T1 FULL OUTER JOIN T2 consists of their paired rows and, for each unpaired row of T2, the concatenation of that row with the null row of T1 and, for each unpaired row of T1, the concatenation of that row with the null row of T2. All columns derived from T1 and T2 allow null values.

where-clause

```
►►──WHERE──search-condition──────────────────────────────────────────►◄
```

The WHERE clause specifies an intermediate result table that consists of those rows of R for which the *search-condition* is true. R is the result of the FROM clause of the subselect.

The *search-condition* must conform to the following rules:

- Each *column-name* must unambiguously identify a column of R or be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.
- A column function must not be specified unless the WHERE clause is specified in a subquery of a HAVING clause and the argument of the function is a correlated reference to a group.

Any subquery in the *search-condition* is effectively executed for each row of R, and the results are used in the application of the *search-condition* to the given row of R. A subquery is actually executed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references may be executed just once, whereas a subquery with a correlated reference may have to be executed once for each row.

group-by-clause

```
                  ┌─,──────────────────────┐
                  ▼                        │
►►──GROUP BY────┬─grouping-expression─┬──┴──────────────────────────►◄
                ├─grouping-sets───────┤
                └─super-groups────────┘
```

The GROUP BY clause specifies an intermediate result table that consists of a grouping of the rows of R. R is the result of the previous clause of the subselect.

In its simplest form, a GROUP BY clause contains a *grouping expression*. A grouping expression is an *expression* used in defining the grouping of R. Each *column name* included in grouping-expression must unambiguously identify a column of R

(SQLSTATE 42702 or 42703). A grouping expression cannot include a scalar-fullselect (SQLSTATE 42822) or any function that is variant or has an external action (SQLSTATE 42845).

More complex forms of the GROUP BY clause include *grouping-sets* and *super-groups*. For a description of these forms, see "grouping-sets" and "super-groups" on page 465, respectively.

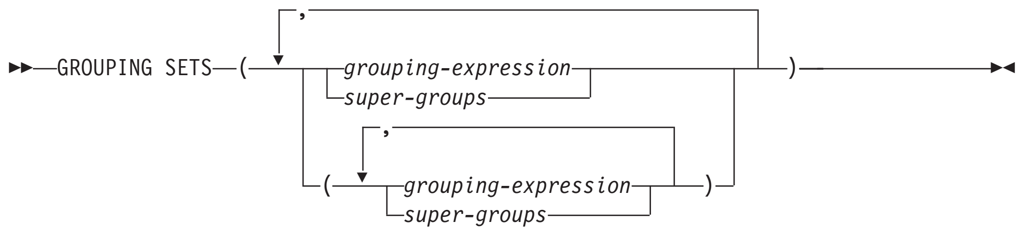
The result of GROUP BY is a set of groups of rows. Each row in this result represents the set of rows for which the *grouping-expression* is equal. For grouping, all null values from a *grouping-expression* are considered equal.

A *grouping-expression* can be used in a search condition in a HAVING clause, in an expression in a SELECT clause or in a *sort-key-expression* of an ORDER BY clause (see "order-by-clause" on page 470 for details). In each case, the reference specifies only one value for each group. For example, if the *grouping-expression* is *col1+col2*, then an allowed expression in the select list would be *col1+col2+3*. Associativity rules for expressions would disallow the similar expression, *3+col1+col2*, unless parentheses are used to ensure that the corresponding expression is evaluated in the same order. Thus, *3+(col1+col2)* would also be allowed in the select list. If the concatenation operator is used, the *grouping-expression* must be used exactly as the expression was specified in the select list.

If the *grouping-expression* contains varying-length strings with trailing blanks, the values in the group can differ in the number of trailing blanks and may not all have the same length. In that case, a reference to the *grouping-expression* still specifies only one value for each group, but the value for a group is chosen arbitrarily from the available set of values. Thus, the actual length of the result value is unpredictable.

As noted, there are some cases where the GROUP BY clause cannot refer directly to a column that is specified in the SELECT clause as an expression (scalar-fullselect, variant or external action functions). To group using such an expression, use a nested table expression or a common table expression to first provide a result table with the expression as a column of the result. For an example using nested table expressions, see "Example A9" on page 474.

grouping-sets



A *grouping-sets* specification allows multiple grouping clauses to be specified in a single statement. This can be thought of as the union of two or more groups of rows into a single result set. It is logically equivalent to the union of multiple subselects with the group by clause in each subselect corresponding to one grouping set. A grouping set can be a single element or can be a list of elements delimited by parentheses, where an element is either a grouping-expression or a super-group. Using *grouping-sets* allows the groups to be computed with a single pass over the base table.

The *grouping-sets* specification allows either a simple *grouping-expression* to be used, or the more complex forms of *super-groups*. For a description of *super-groups*, see “super-groups.”

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
```

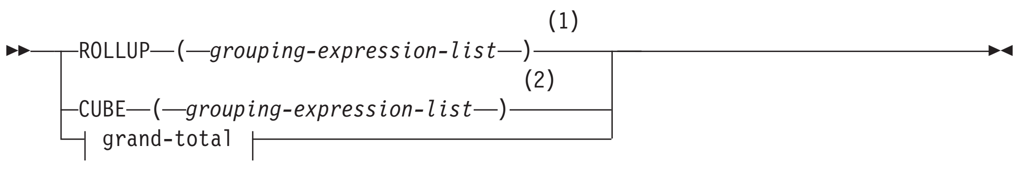
is the same as

```
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.

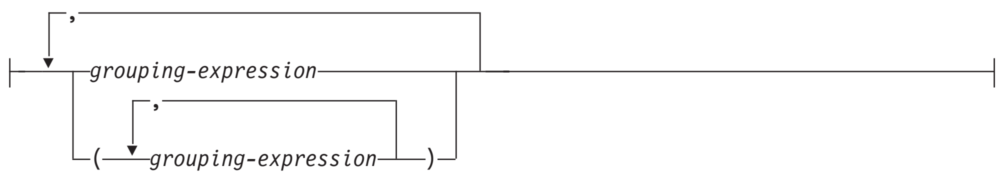
“Example C2” on page 478 through “Example C7” on page 482 illustrate the use of grouping sets.

super-groups

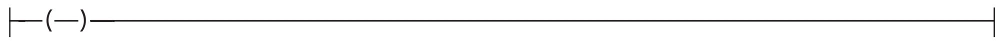


ROLLUP—(—grouping-expression-list—)
(1)
CUBE—(—grouping-expression-list—)
(2)
grand-total

grouping-expression-list:



grand-total:



Notes:

1 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH ROLLUP.

2 Alternate specification when used alone in group-by-clause is: grouping-expression-list WITH CUBE.

ROLLUP (*grouping-expression-list* **)**

A *ROLLUP grouping* is an extension to the GROUP BY clause that produces a result set containing *sub-total* rows in addition to the “regular” grouped rows. *Sub-total* rows are “super-aggregate” rows that contain further aggregates whose values are derived by applying the same column functions that were used to obtain the grouped rows. These rows are called sub-total rows, because that is their most common use; however, any column function can be used for the aggregation. For instance, MAX and AVG are used in “Example C8” on page 483.

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:

```
GROUP BY ROLLUP(a,b)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b)
                       (a)
                       () )
```

while

```
GROUP BY ROLLUP(b,a)
```

is the same as

```
GROUP BY GROUPING SETS((b,a)
                       (b)
                       () )
```

The ORDER BY clause is the only way to guarantee the order of the rows in the result set. “Example C3” on page 478 illustrates the use of ROLLUP.

CUBE (*grouping-expression-list*)

A *CUBE grouping* is an extension to the GROUP BY clause that produces a result set that contains all the rows of a ROLLUP aggregation and, in addition, contains ″cross-tabulation″ rows. *Cross-tabulation* rows are additional ″super-aggregate″ rows that are not part of an aggregation with sub-totals.

Like a ROLLUP, a CUBE grouping can also be thought of as a series of *grouping-sets*. In the case of a CUBE, all permutations of the cubed *grouping-expression-list* are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to 2***n* (2 to the power *n*) *grouping-sets*. For instance, a specification of

```
GROUP BY CUBE(a,b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (b,c)
                       (a)
                       (b)
                       (c)
                       () )
```

Notice that the 3 elements of the CUBE translate to 8 grouping sets.

The order of specification of elements does not matter for CUBE. 'CUBE (DayOfYear, Sales_Person)' and 'CUBE (Sales_Person, DayOfYear)' yield the same result sets. The use of the word 'same' applies to content of the result set, not to its order. The ORDER BY clause is the only way to guarantee the order of the rows in the result set. "Example C4" on page 479 illustrates the use of CUBE.

grouping-expression-list

A *grouping-expression-list* is used within a CUBE or ROLLUP clause to define the number of elements in the CUBE or ROLLUP operation. This is controlled by using parentheses to delimit elements with multiple *grouping-expression*s.

The rules for a *grouping-expression* are described in "group-by-clause" on page 463. For example, suppose that a query is to return the total expenses for the ROLLUP of City within a Province but not within a County. However the clause:

```
GROUP BY ROLLUP(Province, County, City)
```

results in unwanted sub-total rows for the County. In the clause

```
GROUP BY ROLLUP(Province, (County, City))
```

the composite (County, City) forms one element in the ROLLUP and, therefore, a query that uses this clause will yield the desired result. In other words, the two element ROLLUP

```
GROUP BY ROLLUP(Province, (County, City))
```

generates

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province)
                       () )
```

while the 3 element ROLLUP would generate

```
GROUP BY GROUPING SETS((Province, County, City)
                       (Province, County)
                       (Province)
                       () )
```

"Example C2" on page 478 also utilizes composite column values.

grand-total

Both CUBE and ROLLUP return a row which is the overall (grand total) aggregation. This may be separately specified with empty parentheses within the GROUPING SET clause. It may also be specified directly in the GROUP BY clause, although there is no effect on the result of the query. "Example C4" on page 479 uses the grand-total syntax.

Combining grouping sets

This can be used to combine any of the types of GROUP BY clauses. When simple *grouping-expression* fields are combined with other groups, they are ″appended″ to the beginning of the resulting *grouping sets*. When ROLLUP or CUBE expressions are combined, they operate like ″multipliers″ on the remaining expression, forming additional grouping set entries according to the definition of either ROLLUP or CUBE.

For instance, combining *grouping-expression* elements acts as follows:

```
GROUP BY a, ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a) )
```

Or similarly,

```
GROUP BY a, b, ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b) )
```

Combining of *ROLLUP* elements acts as follows:

```
GROUP BY ROLLUP(a), ROLLUP(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a)
                       (b,c)
                       (b)
                       () )
```

Similarly,

```
GROUP BY ROLLUP(a), CUBE(b,c)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c)
                       (a,b)
                       (a,c)
                       (a)
                       (b,c)
                       (b)
                       (c)
                       () )
```

Combining of *CUBE* and *ROLLUP* elements acts as follows:

```
GROUP BY CUBE(a,b), ROLLUP(c,d)
```

is equivalent to

```
GROUP BY GROUPING SETS((a,b,c,d)
                       (a,b,c)
                       (a,b)
                       (a,c,d)
                       (a,c)
                       (a)
```

```
                         (b,c,d)
                         (b,c)
                         (b)
                         (c,d)
                         (c)
                         () )
```

Like a simple *grouping-expression*, combining grouping sets also eliminates duplicates within each grouping set. For instance,

```
  GROUP BY a, ROLLUP(a,b)
```

is equivalent to

```
  GROUP BY GROUPING SETS((a,b)
                         (a) )
```

A more complete example of combining grouping sets is to construct a result set that eliminates certain rows that would be returned for a full CUBE aggregation.

For example, consider the following GROUP BY clause:

```
  GROUP BY Region,
           ROLLUP(Sales_Person, WEEK(Sales_Date)),
           CUBE(YEAR(Sales_Date), MONTH (Sales_Date))
```

The column listed immediately to the right of GROUP BY is simply grouped, those within the parenthesis following ROLLUP are rolled up, and those within the parenthesis following CUBE are cubed. Thus, the above clause results in a cube of MONTH within YEAR which is then rolled up within WEEK within Sales_Person within the Region aggregation. It does not result in any grand total row or any cross-tabulation rows on Region, Sales_Person or WEEK(Sales_Date) so produces fewer rows than the clause:

```
  GROUP BY ROLLUP (Region, Sales_Person, WEEK(Sales_Date),
                   YEAR(Sales_Date), MONTH(Sales_Date) )
```

having-clause

```
►►──HAVING──search-condition──────────────────────────────────────────►◄
```

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the *search-condition* is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered to be a single group with no grouping columns.

Each *column-name* in the search condition must do one of the following:

- Unambiguously identify a grouping column of R.
- Be specified within a column function.
- Be a correlated reference. A *column-name* is a correlated reference if it identifies a column of a *table-reference* in an outer subselect.

A group of R to which the search condition is applied supplies the argument for each column function in the search condition, except for any function whose argument is a correlated reference.

If the search condition contains a subquery, the subquery can be thought of as being executed each time the search condition is applied to a group of R, and the results used in applying the search condition. In actuality, the subquery is executed for each group only if it contains a correlated reference. For an illustration of the difference, see "Example A6" on page 474 and "Example A7" on page 474.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

When HAVING is used without GROUP BY, the select list can only be a column name within a column function, a correlated column reference, a literal, or a special register.

order-by-clause

```
              ┌─,──────────────────────────────┐
              │                  ┌─ASC──┐       │
►►──ORDER BY──▼─┬─┤ sort-key ├──┼──────┼─────┬─┴──────────────────────►◄
                │                  └─DESC─┘     │
                ├─ORDER OF──table-designator────┤
                └─INPUT SEQUENCE────────────────┘
```

sort-key:

```
├──┬─simple-column-name──┬──────────────────────────────────────────────┤
   ├─simple-integer──────┤
   └─sort-key-expression─┘
```

The ORDER BY clause specifies an ordering of the rows of the result table. If a single sort specification (one *sort-key* with associated direction) is identified, the rows are ordered by the values of that sort specification. If more than one sort specification is identified, the rows are ordered by the values of the first identified sort specification, then by the values of the second identified sort specification, and so on. Each *sort-key* cannot have a data type of LONG VARCHAR, CLOB, LONG VARGRAPHIC, DBCLOB, BLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

A named column in the select list may be identified by a *sort-key* that is a *simple-integer* or a *simple-column-name*. An unnamed column in the select list must be identified by 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. The null value is higher than all other values. If the ORDER BY clause does not completely order the rows, rows with duplicate values of all identified columns are displayed in an arbitrary order.

simple-column-name
: Usually identifies a column of the result table. In this case, *simple-column-name* must be the column name of a named column in the select list.

  The *simple-column-name* may also identify a column name of a table, view, or nested table identified in the FROM clause if the query is a subselect. An error occurs if the subselect:

  - Specifies DISTINCT in the select-clause (SQLSTATE 42822)

- Produces a grouped result and the *simple-column-name* is not a *grouping-expression* (SQLSTATE 42803).

Determining which column is used for ordering the result is described under "Column names in sort keys" below.

simple-integer
Must be greater than 0 and not greater than the number of columns in the result table (SQLSTATE 42805). The integer *n* identifies the *n*th column of the result table.

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" below.

There are a number of special cases that further restrict the expressions that can be specified.

- DISTINCT is specified in the SELECT clause of the subselect (SQLSTATE 42822).

  The sort-key-expression must match exactly with an expression in the select list of the subselect (scalar-fullselects are never matched).
- The subselect is grouped (SQLSTATE 42803).

  The sort-key-expression can:
  – be an expression in the select list of the subselect,
  – include a *grouping-expression* from the GROUP BY clause of the subselect
  – include a column function, constant or host variable.

ASC
Uses the values of the column in ascending order. This is the default.

DESC
Uses the values of the column in descending order.

ORDER OF *table-designator*
Specifies that the same ordering used in *table-designator* should be applied to the result table of the subselect. There must be a table reference matching *table-designator* in the FROM clause of the subselect that specifies this clause (SQLSTATE 42703). The subselect (or fullselect) corresponding to the specified *table-designator* must include an ORDER BY clause that is dependant on the data (SQLSTATE 428FI). The ordering that is applied is the same as if the columns of the ORDER BY clause in the nested subselect (or fullselect) were included in the outer subselect (or fullselect), and these columns were specified in place of the ORDER OF clause.

Note that this form is not allowed in a fullselect (other than the degenerative form of a fullselect). For example, the following is not valid:

```
(SELECT C1 FROM T1
   ORDER BY C1)
UNION
SELECT C1 FROM T2
   ORDER BY ORDER OF T1
```

The following example *is* valid:

```
SELECT C1 FROM
   (SELECT C1 FROM T1
      UNION
    SELECT C1 FROM T2
    ORDER BY C1 ) AS UTABLE
ORDER BY ORDER OF UTABLE
```

INPUT SEQUENCE
Specifies that, for an INSERT statement, the result table will reflect the input order of ordered data rows. INPUT SEQUENCE ordering can only be specified if an INSERT statement is used in a FROM clause (SQLSTATE 428G4). See "table-reference" on page 454. If INPUT SEQUENCE is specified and the input data is not ordered, the INPUT SEQUENCE clause is ignored.

Notes:

- **Column names in sort keys**:
  - The column name is qualified.

    The query must be a *subselect* (SQLSTATE 42877). The column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the subselect (SQLSTATE 42702). The value of the column is used to compute the value of the sort specification.
  - The column name is unqualified.
    - The query is a subselect.

      If the column name is identical to the name of more than one column of the result table, the column name must unambiguously identify a column of some table, view or nested table in the FROM clause of the ordering subselect (SQLSTATE 42702). If the column name is identical to one column, that column is used to compute the value of the sort specification. If the column name is not identical to a column of the result table, then it must unambiguously identify a column of some table, view or nested table in the FROM clause of the fullselect in the select-statement (SQLSTATE 42702).
    - The query is not a subselect (it includes set operations such as union, except or intersect).

      The column name must not be identical to the name of more than one column of the result table (SQLSTATE 42702). The column name must be identical to exactly one column of the result table (SQLSTATE 42707), and this column is used to compute the value of the sort specification.
- **Limits**: The use of a *sort-key-expression* or a *simple-column-name* where the column is not in the select list may result in the addition of the column or expression to the temporary table used for sorting. This may result in reaching the limit of the number of columns in a table or the limit on the size of a row in a table. Exceeding these limits will result in an error if a temporary table is required to perform the sorting operation.

fetch-first-clause

```
                  .-1-------.
>>--FETCH FIRST---+---------+---+-ROW--+--ONLY----------------------------><
                  '-integer-'   '-ROWS-'
```

The *fetch-first-clause* sets a maximum number of rows that can be retrieved. It lets the database manager know that the application does not want to retrieve more than *integer* rows, regardless of how many rows there might be in the result table when this clause is not specified. An attempt to fetch beyond *integer* rows is handled the same way as normal end of data (SQLSTATE 02000). The value of *integer* must be a positive integer (not zero).

Limiting the result table to the first *integer* rows can improve performance. The database manager will cease processing the query once it has determined the first *integer* rows. If both the *fetch-first-clause* and the *optimize-for-clause* are specified, the lower of the *integer* values from these clauses is used to influence the communications buffer size. The values are considered independently for optimization purposes.

| If the fullselect contains an SQL data change statement in the FROM clause, all the
| rows are modified regardless of the limit on the number of rows to fetch.

Examples of subselects

Example A1: Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example A2: Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
 SELECT EMP_ACT.*, LASTNAME
   FROM EMP_ACT, EMPLOYEE
   WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
```

Example A3: Join the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee surname (LASTNAME), department number (WORKDEPT in the EMPLOYEE table and DEPTNO in the DEPARTMENT table) and department name (DEPTNAME) of all employees who were born (BIRTHDATE) earlier than 1930.

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
   FROM EMPLOYEE, DEPARTMENT
   WHERE WORKDEPT = DEPTNO
   AND YEAR(BIRTHDATE) < 1930
```

Example A4: Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27000.

```
 SELECT JOB, MIN(SALARY), MAX(SALARY)
   FROM EMPLOYEE
   GROUP BY JOB
   HAVING COUNT(*) > 1
   AND MAX(SALARY) >= 27000
```

Example A5: Select all the rows of EMP_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
SELECT *
  FROM EMP_ACT
  WHERE EMPNO IN
           (SELECT EMPNO
               FROM EMPLOYEE
               WHERE WORKDEPT = 'E11')
```

Example A6: From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                          FROM EMPLOYEE)
```

The subquery in the HAVING clause would only be executed once in this example.

Example A7: Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
   FROM EMPLOYEE EMP_COR
    GROUP BY WORKDEPT
    HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                          FROM EMPLOYEE
                          WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

In contrast to "Example A6," the subquery in the HAVING clause would need to be executed for each group.

Example A8: Determine the employee number and salary of sales representatives along with the average salary and head count of their departments.

This query must first create a nested table expression (DINFO) in order to get the AVGSALARY and EMPCOUNT columns, as well as the DEPTNO column that is used in the WHERE clause.

```
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY, DINFO.AVGSALARY, DINFO.EMPCOUNT
 FROM EMPLOYEE THIS_EMP,
       (SELECT OTHERS.WORKDEPT AS DEPTNO,
                          AVG(OTHERS.SALARY) AS AVGSALARY,
                          COUNT(*) AS EMPCOUNT
         FROM EMPLOYEE OTHERS
         GROUP BY OTHERS.WORKDEPT
        ) AS DINFO
 WHERE THIS_EMP.JOB = 'SALESREP'
  AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Using a nested table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the query, only the rows for the department of the sales representatives need to be considered by the view.

Example A9: Display the average education level and salary for 5 random groups of employees.

This query requires the use of a nested table expression to set a random value for each employee so that it can subsequently be used in the GROUP BY clause.

```
SELECT RANDID , AVG(EDLEVEL), AVG(SALARY)
  FROM ( SELECT EDLEVEL, SALARY, INTEGER(RAND()*5) AS RANDID
           FROM EMPLOYEE
       ) AS EMPRAND
  GROUP BY RANDID
```

Example A10: Query the EMP_ACT table and return those project numbers that have an employee whose salary is in the top 10 of all employees.

```
 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)
```

Examples of joins

Example B1: This example illustrates the results of the various joins using tables J1 and J2. These tables contain rows as shown.

```
SELECT * FROM J1

W   X
--- ------
A       11
B       12
C       13

SELECT * FROM J2

Y   Z
--- ------
A       21
C       22
D       23
```

The following query does an inner join of J1 and J2 matching the first column of both tables.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
```

In this inner join example the row with column W=′C′ from J1 and the row with column Y=′D′ from J2 are not included in the result because they do not have a match in the other table. Note that the following alternative form of an inner join query produces the same result.

```
SELECT * FROM J1, J2 WHERE W=Y
```

The following left outer join will get back the missing row from J1 with nulls for the columns of J2. Every row from J1 is included.

```
SELECT * FROM J1 LEFT OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
B       12 -        -
C       13 C       22
```

The following right outer join will get back the missing row from J2 with nulls for the columns of J1. Every row from J2 is included.

```
SELECT * FROM J1 RIGHT OUTER JOIN J2 ON W=Y

W   X      Y   Z
```

```
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
```

The following full outer join will get back the missing rows from both J1 and J2 with nulls where appropriate. Every row from both J1 and J2 is included.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y

W   X      Y   Z
--- ------ --- ------
A       11 A       21
C       13 C       22
-        - D       23
B       12 -        -
```

Example B2: Using the tables J1 and J2 from the previous example, examine what happens when and additional predicate is added to the search condition.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The additional condition caused the inner join to select only 1 row compared to the inner join in "Example B1" on page 475.

Notice what the impact of this is on the full outer join.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=13

W   X      Y   Z
--- ------ --- ------
-        - A       21
C       13 C       22
-        - D       23
A       11 -        -
B       12 -        -
```

The result now has 5 rows (compared to 4 without the additional predicate) since there was only 1 row in the inner join and all rows of both tables must be returned.

The following query illustrates that placing the same additional predicate in WHERE clause has completely different results.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=13

W   X      Y   Z
--- ------ --- ------
C       13 C       22
```

The WHERE clause is applied after the intermediate result of the full outer join. This intermediate result would be the same as the result of the full outer join query in "Example B1" on page 475. The WHERE clause is applied to this intermediate result and eliminates all but the row that has X=13. Choosing the location of a predicate when performing outer joins can have significant impact on the results. Consider what happens if the predicate was X=12 instead of X=13. The following inner join returns no rows.

```
SELECT * FROM J1 INNER JOIN J2 ON W=Y AND X=12
```

Hence, the full outer join would return 6 rows, 3 from J1 with nulls for the columns of J2 and 3 from J2 with nulls for the columns of J1.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y AND X=12

W   X      Y   Z
--- ------ --- ------
-        - A       21
-        - C       22
-        - D       23
A       11 -        -
B       12 -        -
C       13 -        -
```

If the additional predicate is in the WHERE clause instead, 1 row is returned.

```
SELECT * FROM J1 FULL OUTER JOIN J2 ON W=Y
   WHERE X=12

W   X      Y   Z
--- ------ --- ------
B       12 -        -
```

Example B3: List every department with the employee number and last name of the manager, including departments without a manager.

```
SELECT DEPTNO, DEPTNAME, EMPNO, LASTNAME
  FROM DEPARTMENT LEFT OUTER JOIN EMPLOYEE
       ON MGRNO = EMPNO
```

Example B4: List every employee number and last name with the employee number and last name of their manager, including employees without a manager.

```
SELECT E.EMPNO, E.LASTNAME, M.EMPNO, M.LASTNAME
  FROM EMPLOYEE E LEFT OUTER JOIN
                                       DEPARTMENT INNER JOIN EMPLOYEE M
       ON MGRNO = M.EMPNO
       ON E.WORKDEPT = DEPTNO
```

The inner join determines the last name for any manager identified in the DEPARTMENT table and the left outer join guarantees that each employee is listed even if a corresponding department is not found in DEPARTMENT.

Examples of grouping sets, cube, and rollup

The queries in "Example C1" on page 478 through "Example C4" on page 479 use a subset of the rows in the SALES tables based on the predicate 'WEEK(SALES_DATE) = 13'.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SALES AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
```

which results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 LUCCHESSI                 3
         13           6 LUCCHESSI                 1
         13           6 LEE                       2
         13           6 LEE                       2
         13           6 LEE                       3
         13           6 LEE                       5
         13           6 GOUNOT                    3
         13           6 GOUNOT                    1
         13          6 GOUNOT                    7
         13          7 LUCCHESSI                 1
         13          7 LUCCHESSI                 2
         13          7 LUCCHESSI                 1
         13          7 LEE                       7
         13          7 LEE                       3
         13          7 LEE                       7
         13          7 LEE                       4
         13          7 GOUNOT                    2
         13          7 GOUNOT                   18
         13          7 GOUNOT                    1
```

Example C1: Here is a query with a basic GROUP BY clause over 3 columns:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
```

Example C2: Produce the result based on two different grouping sets of rows from the SALES table.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
FROM SALES
WHERE WEEK(SALES_DATE) = 13
GROUP BY GROUPING SETS ( (WEEK(SALES_DATE), SALES_PERSON),
                         (DAYOFWEEK(SALES_DATE), SALES_PERSON))
ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           - GOUNOT                   32
         13           - LEE                      33
         13           - LUCCHESSI                 8
          -           6 GOUNOT                   11
          -           6 LEE                      12
          -           6 LUCCHESSI                 4
          -           7 GOUNOT                   21
          -           7 LEE                      21
          -           7 LUCCHESSI                 4
```

The rows with WEEK 13 are from the first grouping set and the other rows are from the second grouping set.

Example C3: If you use the 3 distinct columns involved in the grouping sets of "Example C2" and perform a ROLLUP, you can see grouping sets for (WEEK,DAY_WEEK,SALES_PERSON), (WEEK, DAY_WEEK), (WEEK) and grand total.

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SALES_PERSON, SUM(SALES) AS UNITS_SOLD
  FROM SALES
  WHERE WEEK(SALES_DATE) = 13
  GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE), SALES_PERSON )
  ORDER BY WEEK, DAY_WEEK, SALES_PERSON
```

This results in:

```
WEEK        DAY_WEEK    SALES_PERSON    UNITS_SOLD
----------- ----------- --------------- -----------
         13           6 GOUNOT                   11
         13           6 LEE                      12
         13           6 LUCCHESSI                 4
         13           6 -                        27
         13           7 GOUNOT                   21
         13           7 LEE                      21
         13           7 LUCCHESSI                 4
         13           7 -                        46
         13           - -                        73
          -           - -                        73
```

Example C4: If you run the same query as "Example C3" on page 478 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 C5: Obtain a result set which includes a grand-total of selected rows from the SALES table together with a group of rows aggregated by SALES_PERSON and MONTH.

```
SELECT SALES_PERSON,
       MONTH(SALES_DATE) AS MONTH,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY GROUPING SETS ( (SALES_PERSON, MONTH(SALES_DATE)),
                         ()
                       )
ORDER BY SALES_PERSON, MONTH
```

This results in:

```
SALES_PERSON    MONTH       UNITS_SOLD
--------------- ----------- -----------
GOUNOT                    3          35
GOUNOT                    4          14
GOUNOT                   12           1
LEE                       3          60
LEE                       4          25
LEE                      12           6
LUCCHESSI                 3           9
LUCCHESSI                 4           4
LUCCHESSI                12           1
-                         -         155
```

Example C6: This example shows two simple ROLLUP queries followed by a query which treats the two ROLLUPs as grouping sets in a single result set and specifies row ordering for each column involved in the grouping sets.

Example C6-1:

```
SELECT WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) )
ORDER BY WEEK, DAY_WEEK
```

results in:

```
WEEK        DAY_WEEK    UNITS_SOLD
----------- ----------- -----------
         13           6          27
         13           7          46
         13           -          73
         14           1          31
         14           2          43
         14           -          74
         53           1           8
         53           -           8
          -           -         155
```

Example C6-2:

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD
FROM SALES
GROUP BY ROLLUP ( MONTH(SALES_DATE), REGION );
ORDER BY MONTH, REGION
```

results in:

```
  MONTH       REGION          UNITS_SOLD
  ----------- --------------- -----------
            3 Manitoba                 22
            3 Ontario-North             8
            3 Ontario-South            34
            3 Quebec                   40
            3 -                       104
            4 Manitoba                 17
            4 Ontario-North             1
            4 Ontario-South            14
            4 Quebec                   11
            4 -                        43
           12 Manitoba                  2
           12 Ontario-South             4
           12 Quebec                    2
           12 -                         8
            - -                       155
```

Example C6-3:

```
  SELECT WEEK(SALES_DATE) AS WEEK,
         DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
         MONTH(SALES_DATE) AS MONTH,
         REGION,
         SUM(SALES) AS UNITS_SOLD
  FROM SALES
  GROUP BY GROUPING SETS ( ROLLUP( WEEK(SALES_DATE), DAYOFWEEK(SALES_DATE) ),
                           ROLLUP( MONTH(SALES_DATE), REGION )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
  WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
  ----------- ----------- ----------- --------------- -----------
           13           6           - -                        27
           13           7           - -                        46
           13           -           - -                        73
           14           1           - -                        31
           14           2           - -                        43
           14           -           - -                        74
           53           1           - -                         8
           53           -           - -                         8
            -           -           3 Manitoba                 22
            -           -           3 Ontario-North             8
            -           -           3 Ontario-South            34
            -           -           3 Quebec                   40
            -           -           3 -                       104
            -           -           4 Manitoba                 17
            -           -           4 Ontario-North             1
            -           -           4 Ontario-South            14
            -           -           4 Quebec                   11
            -           -           4 -                        43
            -           -          12 Manitoba                  2
            -           -          12 Ontario-South             4
            -           -          12 Quebec                    2
            -           -          12 -                         8
            -           -           - -                       155
            -           -           - -                       155
```

Using the two ROLLUPs as grouping sets causes the result to include duplicate rows. There are even two grand total rows.

Observe how the use of ORDER BY has affected the results:

- In the first grouped set, week 53 has been repositioned to the end.

- In the second grouped set, month 12 has now been positioned to the end and the regions now appear in alphabetic order.
- Null values are sorted high.

Example C7: In queries that perform multiple ROLLUPs in a single pass (such as "Example C6-3" on page 481) you may want to be able to indicate which grouping set produced each row. The following steps demonstrate how to provide a column (called GROUP) which indicates the origin of each row in the result set. By origin, we mean which one of the two grouping sets produced the row in the result set.

Step 1: Introduce a way of ″generating″ new data values, using a query which selects from a VALUES clause (which is an alternate form of a fullselect). This query shows how a table can be derived called ″X″ having 2 columns ″R1″ and ″R2″ and 1 row of data.

```
SELECT R1,R2
FROM (VALUES('GROUP 1','GROUP 2')) AS X(R1,R2);
```

results in:

```
R1      R2
------- -------
GROUP 1 GROUP 2
```

Step 2: Form the cross product of this table ″X″ with the SALES table. This add columns ″R1″ and ″R2″ to every row.

```
SELECT R1, R2, WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION,
       SALES AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
```

This add columns ″R1″ and ″R2″ to every row.

Step 3: Now we can combine these columns with the grouping sets to include these columns in the rollup analysis.

```
SELECT R1, R2,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                         DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY WEEK, DAY_WEEK, MONTH, REGION
```

results in:

```
R1      R2      WEEK     DAY_WEEK  MONTH     REGION          UNITS_SOLD
------- ------- -------- --------- --------- --------------- -----------
GROUP 1 -             13         6         - -                       27
GROUP 1 -             13         7         - -                       46
GROUP 1 -             13         -         - -                       73
GROUP 1 -             14         1         - -                       31
GROUP 1 -             14         2         - -                       43
GROUP 1 -             14         -         - -                       74
GROUP 1 -             53         1         - -                        8
GROUP 1 -             53         -         - -                        8
-       GROUP 2        -         -         3 Manitoba                22
-       GROUP 2        -         -         3 Ontario-North            8
-       GROUP 2        -         -         3 Ontario-South           34
-       GROUP 2        -        -         3 Quebec                40
-       GROUP 2        -        -         3 -                    104
-       GROUP 2        -        -         4 Manitoba              17
-       GROUP 2        -        -         4 Ontario-North          1
-       GROUP 2        -        -         4 Ontario-South         14
-       GROUP 2        -        -         4 Quebec                11
-       GROUP 2        -        -         4 -                     43
-       GROUP 2        -        -        12 Manitoba               2
-       GROUP 2        -        -        12 Ontario-South          4
-       GROUP 2        -        -        12 Quebec                 2
-       GROUP 2        -        -        12 -                      8
-       GROUP 2        -        -         - -                    155
GROUP 1 -              -        -         - -                    155
```

Step 4: Notice that because R1 and R2 are used in different grouping sets, whenever R1 is non-null in the result, R2 is null and whenever R2 is non-null in the result, R1 is null. That means you can consolidate these columns into a single column using the COALESCE function. You can also use this column in the ORDER BY clause to keep the results of the two grouping sets together.

```
SELECT COALESCE(R1,R2) AS GROUP,
       WEEK(SALES_DATE) AS WEEK,
       DAYOFWEEK(SALES_DATE) AS DAY_WEEK,
       MONTH(SALES_DATE) AS MONTH,
       REGION, SUM(SALES) AS UNITS_SOLD
FROM SALES,(VALUES('GROUP 1','GROUP 2')) AS X(R1,R2)
GROUP BY GROUPING SETS ((R1, ROLLUP(WEEK(SALES_DATE),
                         DAYOFWEEK(SALES_DATE))),
                       (R2,ROLLUP( MONTH(SALES_DATE), REGION ) )  )
ORDER BY GROUP, WEEK, DAY_WEEK, MONTH, REGION;
```

results in:

```
GROUP   WEEK        DAY_WEEK    MONTH       REGION          UNITS_SOLD
------- ----------- ----------- ----------- --------------- -----------
GROUP 1          13           6           - -                       27
GROUP 1          13           7           - -                       46
GROUP 1          13           -           - -                       73
GROUP 1          14           1           - -                       31
GROUP 1          14           2           - -                       43
GROUP 1          14           -           - -                       74
GROUP 1          53           1           - -                        8
GROUP 1          53           -           - -                        8
GROUP 1           -           -           - -                      155
GROUP 2           -           -           3 Manitoba                22
GROUP 2           -           -           3 Ontario-North            8
GROUP 2           -           -           3 Ontario-South           34
GROUP 2           -           -           3 Quebec                  40
GROUP 2           -           -           3 -                      104
GROUP 2           -           -           4 Manitoba                17
GROUP 2           -           -           4 Ontario-North            1
GROUP 2           -           -           4 Ontario-South           14
GROUP 2           -           -           4 Quebec                  11
GROUP 2           -           -           4 -                       43
GROUP 2           -           -          12 Manitoba                 2
GROUP 2           -           -          12 Ontario-South            4
GROUP 2           -           -          12 Quebec                   2
GROUP 2           -           -          12 -                        8
GROUP 2           -           -           - -                      155
```

Example C8: The following example illustrates the use of various column functions when performing a CUBE. The example also makes use of cast functions and rounding to produce a decimal result with reasonable precision and scale.

```
SELECT MONTH(SALES_DATE) AS MONTH,
       REGION,
       SUM(SALES) AS UNITS_SOLD,
       MAX(SALES) AS BEST_SALE,
       CAST(ROUND(AVG(DECIMAL(SALES)),2) AS DECIMAL(5,2)) AS AVG_UNITS_SOLD
FROM SALES
GROUP BY CUBE(MONTH(SALES_DATE),REGION)
ORDER BY MONTH, REGION
```

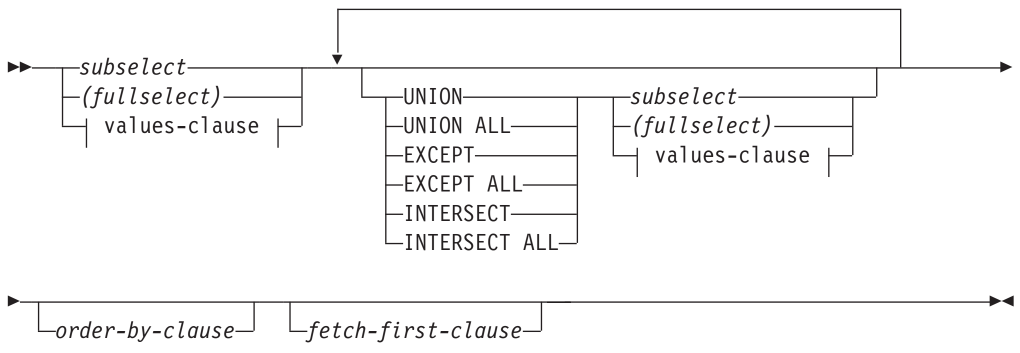
This results in:

```
MONTH       REGION          UNITS_SOLD  BEST_SALE   AVG_UNITS_SOLD
----------- --------------- ----------- ----------- --------------
          3 Manitoba                 22           7           3.14
          3 Ontario-North             8           3           2.67
          3 Ontario-South            34          14           4.25
          3 Quebec                   40          18           5.00
          3 -                       104          18           4.00
          4 Manitoba                 17           9           5.67
          4 Ontario-North             1           1           1.00
          4 Ontario-South            14           8           4.67
          4 Quebec                   11           8           5.50
          4 -                        43           9           4.78
         12 Manitoba                  2           2           2.00
         12 Ontario-South             4           3           2.00
         12 Quebec                    2           1           1.00
         12 -                         8           3           1.60
          - Manitoba                 41           9           3.73
          - Ontario-North             9           3           2.25
          - Ontario-South            52          14           4.00
          - Quebec                   53          18           4.42
          - -                       155          18           3.87
```

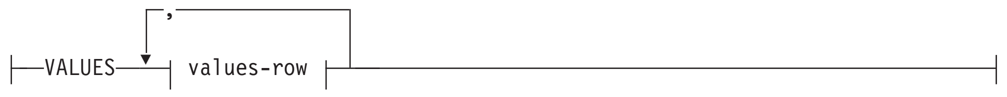
Related reference:

- "Identifiers" on page 58
- "Functions" on page 152
- "GROUPING" on page 265
- "Fullselect" on page 485
- "Select-statement" on page 490
- "DELETE statement" in the *SQL Reference, Volume 2*
- "INSERT statement" in the *SQL Reference, Volume 2*
- "UPDATE statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (SQL Scalar, Table, or Row) statement" in the *SQL Reference, Volume 2*
- "CREATE FUNCTION (External Table) statement" in the *SQL Reference, Volume 2*
- "Character strings" on page 81
- "Assignments and comparisons" on page 99
- "Predicates" on page 204

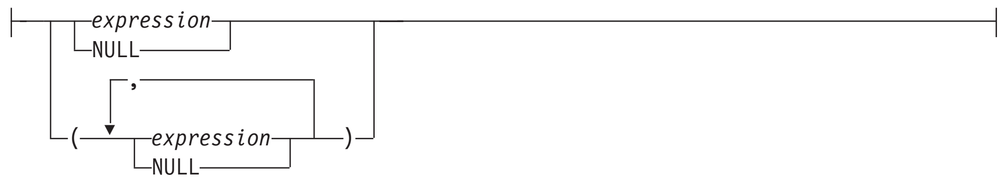
Fullselect



values-clause:



values-row:



The *fullselect* is a component of the select-statement, the INSERT statement, and the CREATE VIEW statement. It is also a component of certain predicates which, in turn, are components of a statement. A fullselect that is a component of a predicate is called a *subquery*, and a fullselect that is enclosed in parentheses is sometimes called a subquery.

The set operators UNION, EXCEPT, and INTERSECT correspond to the relational operators union, difference, and intersection.

A fullselect specifies a result table. If a set operator is not used, the result of the fullselect is the result of the specified subselect or values-clause.

values-clause

Derives a result table by specifying the actual values, using expressions, for each column of a row in the result table. Multiple rows may be specified.

NULL can only be used with multiple specifications of *values-row*, and at least one row in the same column must not be NULL (SQLSTATE 42826).

A *values-row* is specified by:

- A single expression for a single column result table or,
- *n* expressions (or NULL) separated by commas and enclosed in parentheses, where *n* is the number of columns in the result table.

A multiple row VALUES clause must have the same number of expressions in each *values-row* (SQLSTATE 42826).

The following are examples of values-clauses and their meaning.

```
VALUES (1),(2),(3)          - 3 rows of 1 column
VALUES 1, 2, 3              - 3 rows of 1 column
VALUES (1, 2, 3)            - 1 row of 3 columns
VALUES (1,21),(2,22),(3,23) - 3 rows of 2 columns
```

A values-clause that is composed of *n* specifications of *values-row*, RE_1 to RE_n, where *n* is greater than 1, is equivalent to:

RE_1 UNION ALL RE_2 ... UNION ALL RE_n

This means that the corresponding expressions of each *values-row* must be comparable (SQLSTATE 42825).

UNION or **UNION ALL**
Derives a result table by combining two other result tables (R1 and R2). If UNION ALL is specified, the result consists of all rows in R1 and R2. If UNION is specified without the ALL option, the result is the set of all rows in either R1 or R2, with the duplicate rows eliminated. In either case, however, each row of the UNION table is either a row from R1 or a row from R2.

EXCEPT or **EXCEPT ALL**
Derives a result table by combining two other result tables (R1 and R2). If EXCEPT ALL is specified, the result consists of all rows that do not have a corresponding row in R2, where duplicate rows are significant. If EXCEPT is specified without the ALL option, the result consists of all rows that are only in R1, with duplicate rows in the result of this operation eliminated.

INTERSECT or **INTERSECT ALL**
Derives a result table by combining two other result tables (R1 and R2). If INTERSECT ALL is specified, the result consists of all rows that are in both R1 and R2. If INTERSECT is specified without the ALL option, the result consists of all rows that are in both R1 and R2, with the duplicate rows eliminated.

order-by-clause
A fullselect that contains an ORDER BY or FETCH FIRST clause cannot be specified in:

- A materialized query table
- The outermost fullselect of a view (SQLSTATE 428FJ).

Note: An ORDER BY clause in a fullselect does not affect the order of the rows returned by a query. An ORDER BY clause only affects the order of the rows returned if it is specified in the outermost fullselect.

The number of columns in the result tables R1 and R2 must be the same (SQLSTATE 42826). If the ALL keyword is not specified, R1 and R2 must not include any columns having a data type of LONG VARCHAR, CLOB, LONG VARGRAPHIC, DBCLOB, BLOB, DATALINK, distinct type on any of these types, or structured type (SQLSTATE 42907).

The columns of the result are named as follows:

- If the *n*th column of R1 and the *n*th column of R2 have the same result column name, then the *n*th column of R has the result column name.

- If the *n*th column of R1 and the *n*th column of R2 have different result column names, a name is generated. This name cannot be used as the column name in an ORDER BY or UPDATE clause.

The generated name can be determined by performing a DESCRIBE of the SQL statement and consulting the SQLNAME field.

Two rows are duplicates of one another if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

When multiple operations are combined in an expression, operations within parentheses are performed first. If there are no parentheses, the operations are performed from left to right with the exception that all INTERSECT operations are performed before UNION or EXCEPT operations.

In the following example, the values of tables R1 and R2 are shown on the left. The other headings listed show the values as a result of various set operations on R1 and R2.

R1	R2	UNION ALL	UNION	EXCEPT ALL	EXCEPT	INTER-SECT ALL	INTER-SECT
1	1	1	1	1	2	1	1
1	1	1	2	2	5	1	3
1	3	1	3	2		3	4
2	3	1	4	2		4	
2	3	1	5	4			
2	3	2		5			
3	4	2					
4		2					
4		3					
5		3					
		3					
		3					
		3					
		4					
		4					
		4					
		5					

Examples of a fullselect

Example 1: Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2: List the employee numbers (EMPNO) of all employees in the EMPLOYEE table whose department number (WORKDEPT) either begins with 'E'

or who are assigned to projects in the EMP_ACT table whose project number (PROJNO) equals 'MA2100', 'MA2110', or 'MA2112'.

```
SELECT EMPNO
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 3: Make the same query as in example 2, and, in addition, “tag” the rows from the EMPLOYEE table with 'emp' and the rows from the EMP_ACT table with 'emp_act'. Unlike the result from example 2, this query may return the same EMPNO more than once, identifying which table it came from by the associated “tag”.

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   WHERE WORKDEPT LIKE 'E%'
UNION
SELECT EMPNO, 'emp_act' FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 4: Make the same query as in example 2, only use UNION ALL so that no duplicate rows are eliminated.

```
SELECT EMPNO
    FROM EMPLOYEE
    WHERE WORKDEPT LIKE 'E%'
UNION ALL
SELECT EMPNO
   FROM EMP_ACT
   WHERE PROJNO IN('MA2100', 'MA2110', 'MA2112')
```

Example 5: Make the same query as in Example 3, only include an additional two employees currently not in any table and tag these rows as ″new″.

```
SELECT EMPNO, 'emp'
   FROM EMPLOYEE
   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.

Related reference:

- "Rules for result data types" on page 113
- "Rules for string conversions" on page 117

Select-statement

```
>>─┬──────────────────────────────────────┬─fullselect──●──┬───────────────────┬──────>
   │        ┌─,─────────────────────┐     │                 ├─read-only-clause──┤
   └─WITH───▼─common-table-expression─┴───┘                 └─update-clause─────┘

>─●──┬──────────────────────┬──●──┬──────────────────┬──●──────────────────────────><
     └─optimize-for-clause──┘     └─isolation-clause─┘
```

The *select-statement* is the form of a query that can be directly specified in a DECLARE CURSOR statement, or prepared and then referenced in a DECLARE CURSOR statement. It can also be issued through the use of dynamic SQL statements using the command line processor (or similar tools), causing a result table to be displayed on the user's screen. In either case, the table specified by a *select-statement* is the result of the fullselect.

common-table-expression

```
>>──table-name──┬────────────────────────────────┬──AS──(──fullselect──)──────────><
                │     ┌─,───────────┐            │
                │     ▼             │      (1)   │
                └─(───┴─column-name─┴──)─────────┘
```

Notes:

1 If a common table expression is recursive, or if the fullselect results in duplicate column names, column names must be specified.

A *common table expression* permits defining a result table with a *table-name* that can be specified as a table name in any FROM clause of the fullselect that follows. Multiple common table expressions can be specified following the single WITH keyword. Each common table expression specified can also be referenced by name in the FROM clause of subsequent common table expressions.

If a list of columns is specified, it must consist of as many names as there are columns in the result table of the fullselect. Each *column-name* must be unique and unqualified. If these column names are not specified, the names are derived from the select list of the fullselect used to define the common table expression.

The *table-name* of a common table expression must be different from any other common table expression *table-name* in the same statement (SQLSTATE 42726). If the common table expression is specified in an INSERT statement the *table-name* cannot be the same as the table or view name that is the object of the insert (SQLSTATE 42726). A common table expression *table-name* can be specified as a table name in any FROM clause throughout the fullselect. A *table-name* of a common table expression overrides any existing table, view or alias (in the catalog) with the same qualified name.

If more than one common table expression is defined in the same statement, cyclic references between the common table expressions are not permitted (SQLSTATE 42835). A *cyclic reference* occurs when two common table expressions *dt1* and *dt2* are created such that *dt1* refers to *dt2* and *dt2* refers to *dt1*.

If the fullselect of a common table expression contains a *data-change-table-reference* in the FROM clause, the common table expression is said to modify data. A common table expression that modifies data is always evaluated when the statement is processed, regardless of whether the common table expression is used anywhere else in the statement. If there is at least one common table expression that reads or modifies data, all common table expressions are processed in the order in which they occur, and each common table expression that reads or modifies data is completely executed, including all constraints and triggers, before any subsequent common table expressions are executed.

The common table expression is also optional prior to the fullselect in the CREATE VIEW and INSERT statements.

A common table expression can be used:
- In place of a view to avoid creating the view (when general use of the view is not required and positioned updates or deletes are not used)
- To enable grouping by a column that is derived from a scalar subselect or function that is not deterministic or has external action
- When the desired result table is based on host variables
- When the same result table needs to be shared in a fullselect
- When the result needs to be derived using recursion
- When multiple SQL data change statements need to be processed within the query

If the fullselect of a common table expression contains a reference to itself in a FROM clause, the common table expression is a *recursive common table expression*. Queries using recursion are useful in supporting applications such as bill of materials (BOM), reservation systems, and network planning.

The following must be true of a recursive common table expression:
- Each fullselect that is part of the recursion cycle must start with SELECT or SELECT ALL. Use of SELECT DISTINCT is not allowed (SQLSTATE 42925). Furthermore, the unions must use UNION ALL (SQLSTATE 42925).
- The column names must be specified following the *table-name* of the common table expression (SQLSTATE 42908).
- The first fullselect of the first union (the initialization fullselect) must not include a reference to any column of the common table expression in any FROM clause (SQLSTATE 42836).
- If a column name of the common table expression is referred to in the iterative fullselect, the data type, length, and code page for the column are determined based on the initialization fullselect. The corresponding column in the iterative fullselect must have the same data type and length as the data type and length determined based on the initialization fullselect and the code page must match (SQLSTATE 42825). However, for character string types, the length of the two data types may differ. In this case, the column in the iterative fullselect must have a length that would always be assignable to the length determined from the initialization fullselect.
- Each fullselect that is part of the recursion cycle must not include any column functions, group-by-clauses, or having-clauses (SQLSTATE 42836).

  The FROM clauses of these fullselects can include at most one reference to a common table expression that is part of a recursion cycle (SQLSTATE 42836).
- The iterative fullselect and the overall recursive fullselect must not include an order-by-clause (SQLSTATE 42836).

- Subqueries (scalar or quantified) must not be part of any recursion cycles (SQLSTATE 42836).

When developing recursive common table expressions, remember that an infinite recursion cycle (loop) can be created. Check that recursion cycles will terminate. This is especially important if the data involved is cyclic. A recursive common table expression is expected to include a predicate that will prevent an infinite loop. The recursive common table expression is expected to include:

- In the iterative fullselect, an integer column incremented by a constant.
- A predicate in the where clause of the iterative fullselect in the form ″counter_col < constant″ or ″counter _col < :hostvar″.

A warning is issued if this syntax is not found in the recursive common table expression (SQLSTATE 01605).

Recursion example: bill of materials:

Bill of materials (BOM) applications are a common requirement in many business environments. To illustrate the capability of a recursive common table expression for BOM applications, consider a table of parts with associated subparts and the quantity of subparts required by the part. For this example, create the table as follows:

```
CREATE TABLE PARTLIST
               (PART VARCHAR(8),
                SUBPART VARCHAR(8),
                QUANTITY INTEGER);
```

To give query results for this example, assume that the PARTLIST table is populated with the following values:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
00       01                 5
00       05                 3
01       02                 2
01       03                 3
01       04                 4
01       06                 3
02       05                 7
02       06                 6
03       07                 6
04       08                10
04       09                11
05       10                10
05       11                10
06       12                10
06       13                10
07       14                 8
07       12                 8
```

Example 1: Single level explosion

The first example is called single level explosion. It answers the question, “What parts are needed to build the part identified by ’01’?”. The list will include the direct subparts, subparts of the subparts and so on. However, if a part is used multiple times, its subparts are only listed once.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
     (  SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
        FROM PARTLIST ROOT
        WHERE ROOT.PART = '01'
```

```
      UNION ALL
        SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
        FROM RPL PARENT, PARTLIST CHILD
        WHERE  PARENT.SUBPART = CHILD.PART
     )
SELECT DISTINCT PART, SUBPART, QUANTITY
 FROM RPL
  ORDER BY PART, SUBPART, QUANTITY;
```

The above query includes a common table expression, identified by the name *RPL*, that expresses the recursive part of this query. It illustrates the basic elements of a recursive common table expression.

The first operand (fullselect) of the UNION, referred to as the *initialization fullselect*, gets the direct children of part '01'. The FROM clause of this fullselect refers to the source table and will never refer to itself (*RPL* in this case). The result of this first fullselect goes into the common table expression *RPL* (Recursive PARTLIST). As in this example, the UNION must always be a UNION ALL.

The second operand (fullselect) of the UNION uses *RPL* to compute subparts of subparts by having the FROM clause refer to the common table expression *RPL* and the source table with a join of a part from the source table (child) to a subpart of the current result contained in *RPL* (parent). The result goes back to *RPL* again. The second operand of UNION is then used repeatedly until no more children exist.

The SELECT DISTINCT in the main fullselect of this query ensures the same part/subpart is not listed more than once.

The result of the query is as follows:

```
PART     SUBPART  QUANTITY
-------- -------- -----------
01       02                 2
01       03                 3
01       04                 4
01       06                 3
02       05                 7
02       06                 6
03       07                 6
04       08                10
04       09                11
05       10                10
05       11                10
06       12                10
06       13                10
07       12                 8
07       14                 8
```

Observe in the result that from part '01' we go to '02' which goes to '06' and so on. Further, notice that part '06' is reached twice, once through '01' directly and another time through '02'. In the output, however, its subcomponents are listed only once (this is the result of using a SELECT DISTINCT) as required.

It is important to remember that with recursive common table expressions it is possible to introduce an *infinite loop*. In this example, an infinite loop would be created if the search condition of the second operand that joins the parent and child tables was coded as:

```
PARENT.SUBPART = CHILD.SUBPART
```

This example of causing an infinite loop is obviously a case of not coding what is intended. However, care should also be exercised in determining what to code so that there is a definite end of the recursion cycle.

The result produced by this example query could be produced in an application program without using a recursive common table expression. However, this approach would require starting of a new query for every level of recursion. Furthermore, the application needs to put all the results back in the database to order the result. This approach complicates the application logic and does not perform well. The application logic becomes even harder and more inefficient for other bill of material queries, such as summarized and indented explosion queries.

Example 2: Summarized explosion

The second example is a summarized explosion. The question posed here is, what is the total quantity of each part required to build part ’01’. The main difference from the single level explosion is the need to aggregate the quantities. The first example indicates the quantity of subparts required for the part whenever it is required. It does not indicate how many of the subparts are needed to build part ’01’.

```
WITH RPL (PART, SUBPART, QUANTITY) AS
   (
      SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
       FROM PARTLIST ROOT
       WHERE ROOT.PART = '01'
    UNION ALL
      SELECT PARENT.PART, CHILD.SUBPART, PARENT.QUANTITY*CHILD.QUANTITY
       FROM RPL PARENT, PARTLIST CHILD
       WHERE PARENT.SUBPART = CHILD.PART
   )
SELECT PART, SUBPART, SUM(QUANTITY) AS "Total QTY Used"
 FROM RPL
  GROUP BY PART, SUBPART
  ORDER BY PART, SUBPART;
```

In the above query, the select list of the second operand of the UNION in the recursive common table expression, identified by the name *RPL*, shows the aggregation of the quantity. To find out how much of a subpart is used, the quantity of the parent is multiplied by the quantity per parent of a child. If a part is used multiple times in different places, it requires another final aggregation. This is done by the grouping over the common table expression *RPL* and using the SUM column function in the select list of the main fullselect.

The result of the query is as follows:

```
  PART     SUBPART  Total Qty Used
  -------- -------- --------------
  01       02                    2
  01       03                    3
  01       04                    4
  01       05                   14
  01       06                   15
  01       07                   18
  01       08                   40
  01       09                   44
  01       10                  140
  01       11                  140
  01       12                  294
  01       13                  150
  01       14                  144
```

Looking at the output, consider the line for subpart ′06′. The total quantity used value of 15 is derived from a quantity of 3 directly for part ′01′ and a quantity of 6 for part ′02′ which is needed 2 times by part ′01′.

Example 3: Controlling depth

The question may come to mind, what happens when there are more levels of parts in the table than you are interested in for your query? That is, how is a query written to answer the question, "What are the first two levels of parts needed to build the part identified by ′01′?" For the sake of clarity in the example, the level is included in the result.

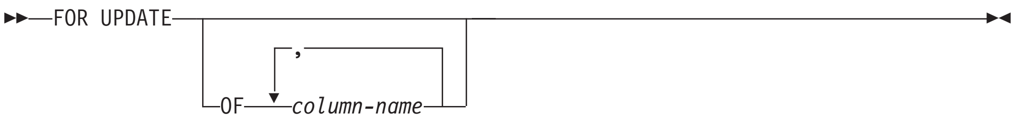
```
WITH RPL (LEVEL, PART, SUBPART, QUANTITY) AS
      (
         SELECT 1,              ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
          FROM PARTLIST ROOT
          WHERE ROOT.PART = '01'
       UNION ALL
         SELECT PARENT.LEVEL+1, CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
          FROM RPL PARENT, 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 updated in a subsequent Positioned UPDATE statement. Each *column-name* must be unqualified and must identify a column of the table or view identified in the first FROM clause of the fullselect. If the FOR UPDATE clause is specified without column names, all updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE clause cannot be used if one of the following is true:
- The cursor associated with the select-statement is not deletable .
- One of the selected columns is a non-updatable column of a catalog table and the FOR UPDATE clause has not been used to exclude that column.

read-only-clause

```
►►──FOR──┬─READ──┬──ONLY──────────────────────────────────────────────►◄
         └─FETCH─┘
```

The FOR READ ONLY clause indicates that the result table is read-only and therefore the cursor cannot be referred to in Positioned UPDATE and DELETE statements. FOR FETCH ONLY has the same meaning.

Some result tables are read-only by nature. (For example, a table based on a read-only view.) FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY (or FOR FETCH ONLY) can possibly improve the performance of FETCH operations by allowing the database manager to do blocking. For example, in programs that contain dynamic SQL statements without the FOR READ ONLY or ORDER BY clause, the database manager might open cursors as if the FOR UPDATE clause were specified. It is recommended, therefore, that the FOR READ ONLY clause be used to improve performance, except in cases where queries will be used in positioned UPDATE or DELETE statements.

A read-only result table must not be referred to in a Positioned UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR READ ONLY (FOR FETCH ONLY).

optimize-for-clause

```
►►──OPTIMIZE FOR──integer──┬─ROWS─┬───────────────────────────────────►◄
                           └─ROW──┘
```

The OPTIMIZE FOR clause requests special processing of the *select statement*. If the clause is omitted, it is assumed that all rows of the result table will be retrieved; if it is specified, it is assumed that the number of rows retrieved will probably not exceed *n*, where *n* is the value of *integer*. The value of *n* must be a positive integer. Use of the OPTIMIZE FOR clause influences query optimization, based on the assumption that *n* rows will be retrieved. In addition, for cursors that are blocked, this clause will influence the number of rows that will be returned in each block (that is, no more than *n* rows will be returned in each block). If both the *fetch-first-clause* and the *optimize-for-clause* are specified, the lower of the integer values from these clauses will be used to influence the communications buffer size. The values are considered independently for optimization purposes.

This clause does not limit the number of rows that can be fetched, or affect the result in any other way than performance. Using OPTIMIZE FOR *n* ROWS can improve performance if no more than *n* rows are retrieved, but may degrade performance if more than *n* rows are retrieved.

If the value of *n* multiplied by the size of the row exceeds the size of the communication buffer, the OPTIMIZE FOR clause will have no impact on the data buffers. The size of the communication buffer is defined by the RQRIOBLK or the ASLHEAPSZ configuration parameter.

isolation-clause

```
►►─┬──────────────────────────────────────────┬─────────────────────────────►◄
   └─WITH──┬─RR──┬──────────────────────┬─┬─┘
           │     └─lock-request-clause─┘ │
           ├─RS──┬──────────────────────┬─┤
           │     └─lock-request-clause─┘ │
           ├─CS───────────────────────────┤
           └─UR───────────────────────────┘
```

The optional *isolation-clause* specifies the isolation level at which the statement is executed, and whether a specific type of lock is to be acquired.

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability
- UR - Uncommitted Read

The default isolation level of the statement is the isolation level of the package in which the statement is bound.

lock-request-clause

```
►►──USE AND KEEP──┬─SHARE─────┬──LOCKS──────────────────────────────────────►◄
                  ├─UPDATE────┤
                  └─EXCLUSIVE─┘
```

The optional *lock-request-clause* specifies the type of lock that the database manager is to acquire and hold:

SHARE Concurrent processes can acquire SHARE or UPDATE locks on the data.

UPDATE Concurrent processes can acquire SHARE locks on the data, but no concurrent process can acquire an UPDATE or EXCLUSIVE lock.

EXCLUSIVE Concurrent processes cannot acquire a lock on the data.

The *lock-request-clause* applies to all base table and index scans required by the query, including those within subqueries, SQL functions and SQL methods. It has no affect on locks placed by procedures, external functions, or external methods. Any SQL function or SQL method invoked (directly or indirectly) by the statement must be created with INHERIT ISOLATION LEVEL WITH LOCK REQUEST (SQLSTATE 42601). The *lock-request-clause* cannot be used with a modifying query that might invoke triggers or that requires referential integrity checks (SQLSTATE 42601).

Examples of a select-statement

Example 1: Select all columns and rows from the EMPLOYEE table.

```
SELECT * FROM EMPLOYEE
```

Example 2: Select the project name (PROJNAME), start date (PRSTDATE), and end date (PRENDATE) from the PROJECT table. Order the result table by the end date with the most recent dates appearing first.

```
SELECT PROJNAME, PRSTDATE, PRENDATE
 FROM PROJECT
 ORDER BY PRENDATE DESC
```

Example 3: Select the department number (WORKDEPT) and average departmental salary (SALARY) for all departments in the EMPLOYEE table. Arrange the result table in ascending order by average departmental salary.

```
SELECT WORKDEPT, AVG(SALARY)
   FROM EMPLOYEE
   GROUP BY WORKDEPT
   ORDER BY 2
```

Example 4: Declare a cursor named UP_CUR to be used in a C program to update the start date (PRSTDATE) and the end date (PRENDATE) columns in the PROJECT table. The program must receive both of these values together with the project number (PROJNO) value for each row.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT PROJNO, PRSTDATE, PRENDATE
              FROM PROJECT
              FOR UPDATE OF PRSTDATE, PRENDATE;
```

Example 5: This example names the expression SAL+BONUS+COMM as TOTAL_PAY

```
 SELECT SALARY+BONUS+COMM AS TOTAL_PAY
   FROM EMPLOYEE
   ORDER BY TOTAL_PAY
```

Example 6: Determine the employee number and salary of sales representatives along with the average salary and head count of their departments. Also, list the average salary of the department with the highest average salary.

Using a common table expression for this case saves the overhead of creating the DINFO view as a regular view. During statement preparation, accessing the catalog for the view is avoided and, because of the context of the rest of the fullselect, only the rows for the department of the sales representatives need to be considered by the view.

```
WITH
   DINFO (DEPTNO, AVGSALARY, EMPCOUNT) AS
       (SELECT OTHERS.WORKDEPT, AVG(OTHERS.SALARY), COUNT(*)
         FROM EMPLOYEE OTHERS
         GROUP BY OTHERS.WORKDEPT
       ),
   DINFOMAX AS
       (SELECT MAX(AVGSALARY) AS AVGMAX FROM DINFO)
SELECT THIS_EMP.EMPNO, THIS_EMP.SALARY,
       DINFO.AVGSALARY, DINFO.EMPCOUNT, DINFOMAX.AVGMAX
 FROM EMPLOYEE THIS_EMP, DINFO, DINFOMAX
 WHERE THIS_EMP.JOB = 'SALESREP'
 AND THIS_EMP.WORKDEPT = DINFO.DEPTNO
```

Example 7: Given two tables, EMPLOYEE and PROJECT, replace employee SALLY with a new employee GEORGE, assign all projects lead by SALLY to GEORGE, and return the names of the updated projects.

```
WITH
  NEWEMP AS (SELECT EMPNO FROM NEW TABLE
              (INSERT INTO EMPLOYEE(EMPNO, FIRSTNME)
                  VALUES(NEXT VALUE FOR EMPNO_SEQ, 'GEORGE'))),
  OLDEMP AS (SELECT EMPNO FROM EMPLOYEE WHERE FIRSTNME = 'SALLY'),
  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 should be locked when the query executes.

```
SELECT DEPTNO, DEPTNAME, MGRNO
  FROM DEPT
  WHERE ADMRDEPT ='A00'
  FOR READ ONLY WITH RS USE AND KEEP EXCLUSIVE LOCKS
```

Related reference:

- "Subselect" on page 450
- "DECLARE CURSOR statement" in the *SQL Reference, Volume 2*

Appendix A. SQL limits

The following tables describe certain SQL limits. Adhering to the most restrictive case can help the programmer design application programs that are easily portable.

Table 28. Identifier Length Limits

Description	Limit in Bytes
Longest authorization name (can only be single-byte characters)	30
Longest constraint name	18
Longest correlation name	128
Longest condition name	64
Longest cursor name	18
Longest unqualified column name (nickname)	128
Longest data source index name	128
Longest data source name	128
Longest data source table name (*remote-table-name*)	128
Longest external program name	8
Longest host identifier [a]	255
Longest identifier of a data source user (*remote-authorization-name*)	30
Longest label name	64
Longest method name	18
Longest parameter name [b]	128
Longest password to access a data source	32
Longest savepoint name	128
Longest schema name [c]	30
Longest server (database alias) name	8
Longest SQL variable name	64
Longest statement name	18
Longest transform group name	18
Longest unqualified column name	30
Longest unqualified package name	8
Longest unqualified user-defined type, user-defined function, user-defined method, buffer pool, table space, database partition group, trigger, index, or index specification name	18
Longest unqualified table name, view name, stored procedure name, sequence name, nickname, or alias	128
Longest wrapper name	128

Table 28. Identifier Length Limits (continued)

Description	Limit in Bytes

Notes:

1. [a] Individual host language compilers may have a more restrictive limit on variable names.
2. [b] Parameter names in an SQL procedure are limited to 64 bytes.
3. [c] The schema name for a user-defined type is limited to 8 bytes.

Table 29. Numeric Limits

Description	Limit
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
Smallest SMALLINT value	−32 768
Largest SMALLINT value	+32 767
Largest decimal precision	31
Smallest DOUBLE value	−1.79769E+308
Largest DOUBLE value	+1.79769E+308
Smallest positive DOUBLE value	+2.225E−307
Largest negative DOUBLE value	−2.225E−307
Smallest REAL value	−3.402E+38
Largest REAL value	+3.402E+38
Smallest positive REAL value	+1.175E−37
Largest negative REAL value	−1.175E−37

Table 30. String Limits

Description	Limit
Maximum length of CHAR (in bytes)	254
Maximum length of VARCHAR (in bytes)	32 672
Maximum length of LONG VARCHAR (in bytes)	32 700
Maximum length of CLOB (in bytes)	2 147 483 647
Maximum length of GRAPHIC (in characters)	127
Maximum length of VARGRAPHIC (in characters)	16 336
Maximum length of LONG VARGRAPHIC (in characters)	16 350
Maximum length of DBCLOB (in characters)	1 073 741 823
Maximum length of BLOB (in bytes)	2 147 483 647
Maximum length of character constant	32 672
Maximum length of graphic constant	16 336
Maximum length of concatenated character string	2 147 483 647
Maximum length of concatenated graphic string	1 073 741 823
Maximum length of concatenated binary string	2 147 483 647

Table 30. String Limits (continued)

Description	Limit
Maximum number of hex constant digits	16 336
Maximum size of a catalog comment (in bytes)	254
Largest instance of a structured type column object at run time (in gigabytes)	1

Table 31. Datetime Limits

Description	Limit
Smallest DATE value	0001-01-01
Largest DATE value	9999-12-31
Smallest TIME value	00:00:00
Largest TIME value	24:00:00
Smallest TIMESTAMP value	0001-01-01-00.00.00.000000
Largest TIMESTAMP value	9999-12-31-24.00.00.000000

Table 32. Database Manager Limits

Description	Limit
Most columns in a table [g]	1 012
Most columns in a view [a]	5 000
Maximum length of a row including all overhead [b] [g]	32 677
Maximum size of a table per partition (in gigabytes) [c] [g]	512
Maximum size of an index per partition (in gigabytes)	512
Most rows in a table per partition	4×10^9
Longest index key including all overhead (in bytes)	1 024
Most columns in an index key	16
Most indexes on a table	32 767 or storage
Most tables referenced in an SQL statement or a view	storage
Most host variable declarations in a precompiled program [c]	storage
Most host variable references in an SQL statement	32 767
Longest host variable value used for insert or update (in bytes)	2 147 483 647
| Longest SQL statement (in bytes)	2 097 152
Most elements in a select list [g]	1 012
Most predicates in a WHERE or HAVING clause	storage
Maximum number of columns in a GROUP BY clause [g]	1 012
Maximum total length of columns in a GROUP BY clause (in bytes) [g]	32 677
Maximum number of columns in an ORDER BY clause [g]	1 012
Maximum total length of columns in an ORDER BY clause (in bytes) [g]	32 677
Maximum size of an SQLDA (in bytes)	storage
Maximum number of prepared statements	storage

Table 32. Database Manager Limits (continued)

Description	Limit
Most declared cursors in a program	storage
Maximum number of cursors opened at one time	storage
Most tables in an SMS table space	65 534
Maximum number of constraints on a table	storage
Maximum level of subquery nesting	storage
Maximum number of subqueries in a single statement	storage
Most values in an INSERT statement [g]	1 012
Most SET clauses in a single UPDATE statement [g]	1 012
Most columns in a UNIQUE constraint (supported via a UNIQUE index)	16
Maximum combined length of columns in a UNIQUE constraint (supported via a UNIQUE index) (in bytes)	1 024
Most referencing columns in a foreign key	16
Maximum combined length of referencing columns in a foreign key (in bytes)	1 024
Maximum length of a check constraint specification (in bytes)	65 535
Maximum number of columns in a partitioning key [e]	500
Maximum number of rows changed in a unit of work	storage
Maximum number of packages	storage
Most constants in a statement	storage
Maximum concurrent users of server [d]	64 000
Maximum number of parameters in a stored procedure	32 767
Maximum number of parameters in a user defined function	90
Maximum run-time depth of cascading triggers	16
Maximum number of simultaneously active event monitors	32
Maximum size of a regular DMS table space (in gigabytes) [c] [g]	512
Maximum size of a long DMS table space (in terabytes) [c]	2
Maximum size of a temporary DMS table space (in terabytes) [c]	2
Maximum number of databases per instance concurrently in use	256
Maximum number of concurrent users per instance	64 000
Maximum number of concurrent applications per database	60 000
Maximum number of connections per process within a DB2 client	512
Maximum depth of cascaded triggers	16
Maximum partition number	999
Most table objects in DMS table space [f]	51 000
Longest variable index key part (in bytes) [h]	1022 or storage
Maximum number of columns in a data source table or view that is referenced by a nickname	5 000

Table 32. Database Manager Limits (continued)

Description	Limit
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
Maximum number of nested levels for stored procedures	16
Maximum number of table spaces in a database	32 768
Maximum number of attributes in a structured type	4082
Maximum number of simultaneously opened LOB locators in a transaction	32 100

Notes:

1. [a] This maximum can be achieved using a join in the CREATE VIEW statement. Selecting from such a view is subject to the limit of most elements in a select list.
2. [b] The actual data for BLOB, CLOB, LONG VARCHAR, DBCLOB, and LONG VARGRAPHIC columns is not included in this count. However, information about the location of that data does take up some space in the row.
3. [c] The numbers shown are architectural limits and approximations. The practical limits may be less.
4. [d] The actual value will be the value of the MAXAGENTS configuration parameter.
5. [e] This is an architectural limit. The limit on the most columns in an index key should be used as the practical limit.
6. [f] Table objects include data, indexes, LONG VARCHAR or VARGRAPHIC columns, and LOB columns. Table objects that are in the same table space as the table data do not count extra toward the limit. However, each table object that is in a different table space than the table data does contribute one toward the limit for each table object type per table in the table space in which the table object resides.
7. [g] For page size-specific values, see Table 33.
8. [h] 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.

Table 33. Database Manager Page Size Specific Limits

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Most columns in a table	500	1 012	1 012	1 012
Maximum length of a row including all overhead	4 005	8 101	16 293	32 677
Maximum size of a table per partition (in gigabytes)	64	128	256	512
Maximum size of an index per partition (in gigabytes)	64	128	256	512
Most elements in a select list	500	1 012	1 012	1 012
Maximum number of columns in a GROUP BY clause	500	1 012	1 012	1 012
Maximum total length of columns in a GROUP BY clause (in bytes)	4 005	8 101	16 293	32 677
Maximum number of columns in an ORDER BY clause	500	1 012	1 012	1 012
Maximum total length of columns in an ORDER BY clause (in bytes)	4 005	8 101	16 293	32 677

Table 33. Database Manager Page Size Specific Limits (continued)

Description	4K page size limit	8K page size limit	16K page size limit	32K page size limit
Most values in an INSERT statement	500	1 012	1 012	1 012
Most SET clauses in a single UPDATE statement	500	1 012	1 012	1 012
Maximum size of a regular DMS table space (in gigabytes)	64	128	256	512

Related reference:

- "maxagents - Maximum number of agents configuration parameter" in the *Administration Guide: Performance*

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 34. Fields of the SQLCA. The field names shown are those present in an SQLCA that is obtained via an INCLUDE statement.

Name	Data Type	Field Values
sqlcaid	CHAR(8)	An "eye catcher" for storage dumps containing 'SQLCA'. The sixth byte is 'L' if line number information is returned from parsing an SQL procedure body.
sqlcabc	INTEGER	Contains the length of the SQLCA, 136.
sqlcode	INTEGER	Contains the SQL return code. **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.

Table 34. 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 may contain information on up to seven 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 digits indicating the version, release, and modification level of the product. For example, SQL08010 means DB2 Universal Database Version 8 Release 1 Modification level 0. 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.

Table 34. 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. 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 CREATE PROCEDURE for an SQL procedure is invoked, and an error is encountered when parsing the SQL procedure body, contains the line number where the error was encountered. The sixth byte of sqlcaid must be 'L' for this to be a valid line number.
sqlerrd(4)	INTEGER	If PREPARE is invoked and successful, contains a relative cost estimate of the resources required to process the statement. If compound SQL is invoked, contains a count of the number of successful sub-statements. If CONNECT is invoked, contains 0 for a one-phase commit from a down-level client; 1 for a one-phase commit; 2 for a one-phase, read-only commit; and 3 for a two-phase commit.
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 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 of 0 for server authentication; 1 for client authentication; 2 for authentication using DB2 Connect; 4 for SERVER_ENCRYPT authentication; 5 for authentication using DB2 Connect with encryption; 7 for KERBEROS authentication; 8 for KRB_SERVER_ENCRYPT authentication; 9 for GSSPLUGIN authentication; 10 for GSS_SERVER_ENCRYPT authentication; and 255 for unspecified authentication.
sqlerrd(6)	INTEGER	For a partitioned database, contains the partition number of the partition that encountered the error or warning. If no errors or warnings were encountered, this field contains the partition number of the coordinator node. The number in this field is the same as that specified for the partition in the db2nodes.cfg file.
sqlwarn	Array	A set of warning indicators, each containing a blank or W. If compound SQL is invoked, contains an accumulation of the warning indicators set for all sub-statements.

Table 34. 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
sqlwarn0	CHAR(1)	Blank if all other indicators are blank; contains W if at least one other indicator is not blank.
sqlwarn1	CHAR(1)	Contains W if the value of a string column was truncated when assigned to a host variable. Contains N if the null terminator was truncated. Contains A if the CONNECT or ATTACH is successful, and the authorization name for the connection is longer than 8 bytes. Contains P if the PREPARE statement relative cost estimate stored in sqlerrd(4) exceeded the value that could be stored in an INTEGER or was less than 1, and either the CURRENT EXPLAIN MODE or the CURRENT EXPLAIN SNAPSHOT special register is set to a value other than NO.
sqlwarn2	CHAR(1)	Contains W if null values were eliminated from the argument of a function. [a]
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)	Reserved for future use.
sqlwarn6	CHAR(1)	Contains W if the result of a date calculation was adjusted to avoid an impossible date.
sqlwarn7	CHAR(1)	Reserved for future use. If CONNECT is invoked and successful, contains 'E' if the DYN_QUERY_MGMT database configuration parameter is enabled.
sqlwarn8	CHAR(1)	Contains W if a character that could not be converted was replaced with a substitution character.
sqlwarn9	CHAR(1)	Contains W if arithmetic expressions with errors were ignored during column function processing.
sqlwarn10	CHAR(1)	Contains W if there was a conversion error when converting a character data value in one of the fields in the SQLCA.
sqlstate	CHAR(5)	A return code that indicates the outcome of the most recently executed SQL statement.

[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 partition, but has data on other partitions. SQLCODE +100 is only returned to the application if agents from all partitions return SQL0100W, either because the table is empty on all partitions, or there are no 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 partitions, and each agent may return a different SQLCA for different errors or warnings. The coordinator agent also has its own SQLCA.

To provide a consistent view for applications, all SQLCA values are merged into one structure, and SQLCA fields indicate global counts, such that:

- For all errors and warnings, the *sqlwarn* field contains the warning flags received from all agents.
- Values in the *sqlerrd* fields indicating row counts are accumulations from all agents.

Note that SQLSTATE 09000 may not be returned every time an error occurs during the processing of a triggered SQL statement.

Appendix C. SQLDA (SQL descriptor area)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement. The SQLDA variables are options that can be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH or EXECUTE statement.

SQLDAs are supported for all languages, but predefined declarations are provided only for C, REXX, FORTRAN, and COBOL.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA describes host variables.

In DESCRIBE and PREPARE, if any one of the columns being described is either a LOB type (LOB locators and file reference variables do not require doubled SQLDAs), reference type, or a user-defined type, the number of SQLVAR entries for the entire SQLDA will be doubled. For example:

- When describing a table with 3 VARCHAR columns and 1 INTEGER column, there will be 4 SQLVAR entries
- When describing a table with 2 VARCHAR columns, 1 CLOB column, and 1 integer column, there will be 8 SQLVAR entries

In EXECUTE, FETCH, and OPEN, if any one of the variables being described is a LOB type (LOB locators and file reference variables do not require doubled SQLDAs) or a structured type, the number of SQLVAR entries for the entire SQLDA must be doubled. (Distinct types and reference types are not relevant in these cases, because the additional information in the double entries is not required by the database.)

SQLDA field descriptions

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of variables collectively named SQLVAR. In OPEN, FETCH, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, each occurrence of SQLVAR describes a column of a result table or a parameter marker. There are two types of SQLVAR entries:

- **Base SQLVARs:** These entries are always present. They contain the base information about the column, parameter marker, or host variable such as data type code, length attribute, column name, host variable address, and indicator variable address.
- **Secondary SQLVARs:** These entries are only present if the number of SQLVAR entries is doubled as per the rules outlined above. For user-defined types (distinct or structured), they contain the user-defined type name. For reference types, they contain the target type of the reference. For LOBs, they contain the length attribute of the host variable and a pointer to the buffer that contains the actual length. (The distinct type and LOB information does not overlap, so distinct types can be based on LOBs without forcing the number of SQLVAR entries on a DESCRIBE to be tripled.) If locators or file reference variables are used to represent LOBs, these entries are not necessary.

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 517.

Fields in the SQLDA header

Table 35. Fields in the SQLDA Header

C Name	SQL Data Type	Usage in DESCRIBE and PREPARE (set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (set by the application prior to executing the statement)
sqldaid	CHAR(8)	The seventh byte of this field is a flag byte named SQLDOUBLED. The database manager sets SQLDOUBLED to the character ’2’ if two SQLVAR entries have been created for each column; otherwise it is set to a blank (X'20' in ASCII, X'40' in EBCDIC). See “Effect of DESCRIBE on the SQLDA” on page 517 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 36. 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 38 on page 519 lists the allowable values and their meanings. Note that for a distinct or reference type, the data type of the base type is placed into this field. For a structured type, the data type of the result of the FROM SQL transform function of the transform group (based on the CURRENT DEFAULT TRANSFORM GROUP special register) for the type is placed into this field. There is no indication in the base SQLVAR that it is part of the description of a user-defined type or reference type.	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 38 on page 519. 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 38 on page 519. 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.

Table 36. 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 using DB2 Connect to access the server, sqlname can be set to indicate a FOR BIT DATA string as follows: • the length of sqlname is 8 • the first four bytes of sqlname are X'00000000' • the remaining four bytes of sqlname are reserved (and currently ignored).

Fields in an occurrence of a secondary SQLVAR

Table 37. 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 characters for a DBCLOB.
reserve2	CHAR(3) for 32 bit, and CHAR(11) for 64 bit.	Not used.	Not used.
sqlflag4	CHAR(1)	The value is X’01’ if the SQLVAR represents a reference type with a target type named in sqldatatype_name. The value is X’12’ if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X’00’.	Set to X’01’ if the SQLVAR represents a reference type with a target type named in sqldatatype_name. Set to X’12’ if the SQLVAR represents a structured type, with the user-defined type name in sqldatatype_name. Otherwise, the value is X’00’.

Table 37. Fields in a Secondary SQLVAR (continued)

Name	Data Type	Usage in DESCRIBE and PREPARE	Usage in FETCH, OPEN, and EXECUTE
sqldatalen	pointer	Not used.	Used for BLOB, CLOB, and DBCLOB host variables only. If this field is NULL, then the actual length (in characters) should be stored in the 4 bytes immediately before the start of the data and SQLDATA should point to the first byte of the field length. If this field is not NULL, it contains a pointer to a 4 byte long buffer that contains the actual length *in bytes* (even for DBCLOB) of the data in the buffer pointed to from the SQLDATA field in the matching base SQLVAR. Note that, whether or not this field is used, the len.sqllonglen field must be set.
sqldatatype_name	VARCHAR(27)	For a 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 type or reference type, but there are no LOB entries or structured type entries

  The first SQLD SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +237 (SQLSTATE 01594) is issued.

The SQLVARs in the SQLDA are NOT set (requiring allocation of additional space and another DESCRIBE) in the following cases:

- SQLN < SQLD and no entry is either a LOB, user-defined type or reference type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +236 (SQLSTATE 01005) is issued.

  Allocate SQLD SQLVARs for a successful DESCRIBE.
- SQLN < SQLD and at least one entry is a distinct type or reference type, but there are no LOB entries or structured type entries

  No SQLVAR entries are set and SQLDOUBLED is set to blank. If the SQLWARN bind option is YES, a warning SQLCODE +239 (SQLSTATE 01005) is issued.

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE including the names of the distinct types and target types of reference types.
- SQLN < 2*SQLD and at least one entry is a LOB or a structured type

  No SQLVAR entries are set and SQLDOUBLED is set to blank. A warning SQLCODE +238 (SQLSTATE 01005) is issued (regardless of the setting of the SQLWARN bind option).

  Allocate 2*SQLD SQLVARs for a successful DESCRIBE.

References in the above lists to LOB entries include distinct type entries whose source type is a LOB type.

The SQLWARN option of the BIND or PREP command is used to control whether the DESCRIBE (or PREPARE INTO) will return the warning SQLCODEs +236, +237, +239. It is recommended that your application code always consider that these SQLCODEs could be returned. The warning SQLCODE +238 is always returned when there are LOB or structured type entries in the select list and there are insufficient SQLVARs in the SQLDA. This is the only way the application can know that the number of SQLVARs must be doubled because of a LOB or structured type entry in the result set.

If a structured type entry is being described, but no FROM SQL transform is defined (either because no TRANSFORM GROUP was specified using the CURRENT DEFAULT TRANSFORM GROUP special register (SQLSTATE 42741), or because the name group does not have a FROM SQL transform function defined (SQLSTATE 42744), the DESCRIBE will return an error. This error is the same error returned for a DESCRIBE of a table with a structured type entry.

SQLTYPE and SQLLEN

Table 38 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 38. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE

	For DESCRIBE and PREPARE INTO		**For FETCH, OPEN, and EXECUTE**	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
384/385	date	10	fixed-length character string representation of a date	length attribute of the host variable
388/389	time	8	fixed-length character string representation of a time	length attribute of the host variable
392/393	timestamp	26	fixed-length character string representation of a timestamp	length attribute of the host variable
396/397	DATALINK	length attribute of the column	DATALINK	length attribute of the host variable
400/401	N/A	N/A	NUL-terminated graphic string	length attribute of the host variable
404/405	BLOB	0 *	BLOB	Not used. *
408/409	CLOB	0 *	CLOB	Not used. *
412/413	DBCLOB	0 *	DBCLOB	Not used. *
448/449	varying-length character string	length attribute of the column	varying-length character string	length attribute of the host variable
452/453	fixed-length character string	length attribute of the column	fixed-length character string	length attribute of the host variable
456/457	long varying-length character string	length attribute of the column	long varying-length character string	length attribute of the host variable
460/461	N/A	N/A	NUL-terminated character string	length attribute of the host variable
464/465	varying-length graphic string	length attribute of the column	varying-length graphic string	length attribute of the host variable
468/469	fixed-length graphic string	length attribute of the column	fixed-length graphic string	length attribute of the host variable
472/473	long varying-length graphic string	length attribute of the column	long graphic string	length attribute of the host variable
480/481	floating point	8 for double precision, 4 for single precision	floating point	8 for double precision, 4 for single precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
492/493	big integer	8	big integer	8
496/497	large integer	4	large integer	4

Table 38. SQLTYPE and SQLLEN values for DESCRIBE, FETCH, OPEN, and EXECUTE (continued)

	For DESCRIBE and PREPARE INTO		For FETCH, OPEN, and EXECUTE	
SQLTYPE	**Column Data Type**	**SQLLEN**	**Host Variable Data Type**	**SQLLEN**
500/501	small integer	2	small integer	2
916/917	Not applicable	Not applicable	BLOB file reference variable.	267
920/921	Not applicable	Not applicable	CLOB file reference variable.	267
924/925	Not applicable	Not applicable	DBCLOB file reference variable.	267
960/961	Not applicable	Not applicable	BLOB locator	4
964/965	Not applicable	Not applicable	CLOB locator	4
968/969	Not applicable	Not applicable	DBCLOB locator	4

Note:

- The len.sqllonglen field in the secondary SQLVAR contains the length attribute of the column.
- The SQLTYPE has changed from the previous version for portability in DB2. The values from the previous version (see previous version SQL Reference) will continue to be supported.

Unrecognized and unsupported SQLTYPEs

The values that appear in the SQLTYPE field of the SQLDA are dependent on the level of data type support available at the sender as well as at the receiver of the data. This is particularly important as new data types are added to the product.

New data types may or may not be supported by the sender or receiver of the data and may or may not even be recognized by the sender or receiver of the data. Depending on the situation, the new data type may be returned, or a compatible data type agreed upon by both the sender and receiver of the data may be returned or an error may result.

When the sender and receiver agree to use a compatible data type, the following indicates the mapping that will take place. This mapping will take place when at least one of the sender or the receiver does not support the data type provided. The unsupported data type can be provided by either the application or the database manager.

Data Type	Compatible Data Type
BIGINT	DECIMAL(19, 0)
ROWID[1]	VARCHAR(40) FOR BIT DATA

[1] ROWID is supported by DB2 Universal Database for z/OS and OS/390 Version 6.

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 39. Values for Sign Indicator of a Packed Decimal Number

	Representation		
Sign	**Binary**	**Decimal**	**Hexadecimal**
Positive (+)	1100	12	C
Negative (-)	1101	13	D

In summary:
- To store any value, allocate *p*/2+1 bytes, where *p* is precision.
- Assign the nybbles from left to right to represent the value. If a number has an even precision, a leading zero nybble is added. This assignment includes leading (insignificant) and trailing (significant) zero digits.
- The sign nybble will be the second nybble of the last byte.

For example:

Column	Value	Nybbles in Hexadecimal Grouped by Bytes
DEC(8,3)	6574.23	00 65 74 23 0C
DEC(6,2)	-334.02	00 33 40 2D
DEC(7,5)	5.2323	05 23 23 0C
DEC(5,2)	-23.5	02 35 0D

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;
```

Related reference:
- "CHAR" on page 288

Appendix D. Catalog views

This appendix contains a description of each system catalog view, including column names and data types.

System catalog views

The database manager creates and maintains two sets of system catalog views that are defined on top of the base system catalog tables.

- SYSCAT views are read-only catalog views that are found in the SYSCAT schema. SELECT privilege on these views is granted to PUBLIC by default.
- SYSSTAT views are updatable catalog views that are found in the SYSSTAT schema. The updatable views contain statistical information that is used by the optimizer. The values in some columns in these views can be changed to test performance. (Before changing any statistics, it is recommended that the RUNSTATS command be invoked so that all the statistics reflect the current state.) Applications should be written to the SYSSTAT views rather than the base catalog tables.

All the system catalog views are created at database creation time. The catalog views cannot be explicitly created or dropped. The views are updated during normal operation in response to SQL data definition statements, environment routines, and certain utilities. Data in the system catalog views is available through normal SQL query facilities. The system catalog views (with the exception of some updatable catalog views) cannot be modified using normal SQL data manipulation statements.

An object (table, column, function, or index) will appear in a user's updatable catalog view only if that user created the object, holds CONTROL privilege on the object, or holds explicit DBADM authority.

The order of columns in the views may change from release to release. To prevent this from affecting programming logic, specify the columns in a select list explicitly, and avoid using `SELECT *`. Columns have consistent names based on the types of objects that they describe.

Described Object	Column Names
Table	TABSCHEMA, TABNAME
Index	INDSCHEMA, INDNAME
View	VIEWSCHEMA, VIEWNAME
Constraint	CONSTSCHEMA, CONSTNAME
Trigger	TRIGSCHEMA, TRIGNAME
Package	PKGSCHEMA, PKGNAME
Type	TYPESCHEMA, TYPENAME, TYPEID
Function	ROUTINESCHEMA, ROUTINENAME, ROUTINEID
Method	ROUTINESCHEMA, ROUTINENAME, ROUTINEID

Procedure	ROUTINESCHEMA, ROUTINENAME, ROUTINEID
Column	COLNAME
Schema	SCHEMANAME
Table Space	TBSPACE
Database partition group	NGNAME
Buffer pool	BPNAME
Event Monitor	EVMONNAME
Creation Timestamp	CREATE_TIME

Road map to the catalog views

Table 40. Road map to the read-only catalog views

Description	Catalog View
attributes of structured data types	"SYSCAT.ATTRIBUTES" on page 529
authorities on database	"SYSCAT.DBAUTH" on page 550
buffer pool configuration on database partition group	"SYSCAT.BUFFERPOOLS" on page 531
buffer pool size on database partition	"SYSCAT.BUFFERPOOLDBPARTITIONS" on page 530
cast functions	"SYSCAT.CASTFUNCTIONS" on page 532
check constraints	"SYSCAT.CHECKS" on page 533
column privileges	"SYSCAT.COLAUTH" on page 534
columns	"SYSCAT.COLUMNS" on page 542
columns referenced by check constraints	"SYSCAT.COLCHECKS" on page 535
columns used in dimensions	"SYSCAT.COLUSE" on page 546
columns used in keys	"SYSCAT.KEYCOLUSE" on page 574
constraint dependencies	"SYSCAT.CONSTDEP" on page 547
database partition group database partitions	"SYSCAT.DBPARTITIONGROUPDEF" on page 551
database partition group definitions	"SYSCAT.DBPARTITIONGROUPS" on page 552
data types	"SYSCAT.DATATYPES" on page 548
detailed column group statistics	"SYSCAT.COLGROUPDIST" on page 537
	"SYSCAT.COLGROUPDISTCOUNTS" on page 538
	"SYSCAT.COLGROUPS" on page 539
detailed column options	"SYSCAT.COLOPTIONS" on page 541
detailed column statistics	"SYSCAT.COLDIST" on page 536
event monitor definitions	"SYSCAT.EVENTMONITORS" on page 553
events currently monitored	"SYSCAT.EVENTS" on page 554
	"SYSCAT.EVENTTABLES" on page 555
function dependencies[1]	"SYSCAT.ROUTINEDEP" on page 591
function mapping	"SYSCAT.FUNCMAPPINGS" on page 559
function mapping options	"SYSCAT.FUNCMAPOPTIONS" on page 557
function parameter mapping options	"SYSCAT.FUNCMAPPARMOPTIONS" on page 558
function parameters[1]	"SYSCAT.ROUTINEPARMS" on page 592
functions[1]	"SYSCAT.ROUTINES" on page 594
hierarchies (types, tables, views)	"SYSCAT.HIERARCHIES" on page 560
	"SYSCAT.FULLHIERARCHIES" on page 556
| identity columns	"SYSCAT.COLIDENTATTRIBUTES" on page 540
index columns	"SYSCAT.INDEXCOLUSE" on page 562

Table 40. Road map to the read-only catalog views (continued)

Description	Catalog View
index dependencies	“SYSCAT.INDEXDEP” on page 563
index exploitation	“SYSCAT.INDEXEXPLOITRULES” on page 568
index extension dependencies	“SYSCAT.INDEXEXTENSIONDEP” on page 569
index extension parameters	“SYSCAT.INDEXEXTENSIONPARMS” on page 571
index extension search methods	“SYSCAT.INDEXEXTENSIONMETHODS” on page 570
index extensions	“SYSCAT.INDEXEXTENSIONS” on page 572
index options	“SYSCAT.INDEXOPTIONS” on page 573
index privileges	“SYSCAT.INDEXAUTH” on page 561
indexes	“SYSCAT.INDEXES” on page 564
method dependencies[1]	“SYSCAT.ROUTINEDEP” on page 591
method parameters[1]	“SYSCAT.ROUTINES” on page 594
methods[1]	“SYSCAT.ROUTINES” on page 594
object mapping	“SYSCAT.NAMEMAPPINGS” on page 575
package dependencies	“SYSCAT.PACKAGEDEP” on page 577
package privileges	“SYSCAT.PACKAGEAUTH” on page 576
packages	“SYSCAT.PACKAGES” on page 578
partitioning maps	“SYSCAT.PARTITIONMAPS” on page 582
pass-through privileges	“SYSCAT.PASSTHRUAUTH” on page 583
predicate specifications	“SYSCAT.PREDICATESPECS” on page 584
procedure options	“SYSCAT.PROCOPTIONS” on page 585
procedure parameter options	“SYSCAT.PROCPARMOPTIONS” on page 586
procedure parameters[1]	“SYSCAT.ROUTINEPARMS” on page 592
procedures[1]	“SYSCAT.ROUTINES” on page 594
provides DB2 Universal Database for z/OS and OS/390 compatibility	“SYSIBM.SYSDUMMY1” on page 528
referential constraints	“SYSCAT.REFERENCES” on page 587
remote table options	“SYSCAT.TABOPTIONS” on page 616
reverse data type mapping	“SYSCAT.REVTYPEMAPPINGS” on page 588
routine dependencies	“SYSCAT.ROUTINEDEP” on page 591
routine parameters[1]	“SYSCAT.ROUTINEPARMS” on page 592
routine privileges	“SYSCAT.ROUTINEAUTH” on page 590
routines[1]	“SYSCAT.ROUTINES” on page 594
schema privileges	“SYSCAT.SCHEMAAUTH” on page 600
schemas	“SYSCAT.SCHEMATA” on page 601
sequence privileges	“SYSCAT.SEQUENCEAUTH” on page 602
sequences	“SYSCAT.SEQUENCES” on page 603
server options	“SYSCAT.SERVEROPTIONS” on page 604

Table 40. Road map to the read-only catalog views (continued)

Description	Catalog View
server options values	"SYSCAT.USEROPTIONS" on page 623
statements in packages	"SYSCAT.STATEMENTS" on page 606
stored procedures	"SYSCAT.ROUTINES" on page 594
system servers	"SYSCAT.SERVERS" on page 605
table constraints	"SYSCAT.TABCONST" on page 609
table dependencies	"SYSCAT.TABDEP" on page 610
table privileges	"SYSCAT.TABAUTH" on page 607
table space use privileges	"SYSCAT.TBSPACEAUTH" on page 617
table spaces	"SYSCAT.TABLESPACES" on page 615
tables	"SYSCAT.TABLES" on page 611
transforms	"SYSCAT.TRANSFORMS" on page 618
trigger dependencies	"SYSCAT.TRIGDEP" on page 619
triggers	"SYSCAT.TRIGGERS" on page 620
type mapping	"SYSCAT.TYPEMAPPINGS" on page 621
user-defined functions	"SYSCAT.ROUTINES" on page 594
views	"SYSCAT.TABLES" on page 611
	"SYSCAT.VIEWS" on page 624
wrapper options	"SYSCAT.WRAPOPTIONS" on page 625
wrappers	"SYSCAT.WRAPPERS" on page 626

[1] The catalog views for functions, methods, and procedures from DB2 Version 7.1 and earlier still exist. These views, however, do not reflect any changes since DB2 Version 7.1. The views are:

```
Functions:  SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Methods:    SYSCAT.FUNCTIONS, SYSCAT.FUNCDEP, SYSCAT.FUNCPARMS
Procedures: SYSCAT.PROCEDURES, SYSCAT.PROCPARMS
```

Table 41. Road map to the updatable catalog views

Description	Catalog View
columns	"SYSSTAT.COLUMNS" on page 628
detailed column statistics	"SYSSTAT.COLDIST" on page 627
indexes	"SYSSTAT.INDEXES" on page 630
routines[1]	"SYSSTAT.ROUTINES" on page 634
tables	"SYSSTAT.TABLES" on page 636

[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.

SYSIBM.SYSDUMMY1

Contains one row. This view is available for applications that require compatibility with DB2 Universal Database for z/OS and OS/390.

Table 42. SYSCAT.DUMMY1 Catalog View

Column Name	Data Type	Nullable	Description
IBMREQD	CHAR(1)		Y

SYSCAT.ATTRIBUTES

Contains one row for each attribute (including inherited attributes where applicable) that is defined for a user-defined structured data type.

Table 43. SYSCAT.ATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR(128)		Qualified name of the structured data type that includes the attribute.
TYPENAME	VARCHAR(128)		
ATTR_NAME	VARCHAR(128)		Attribute name.
ATTR_TYPESCHEMA	VARCHAR(128)		Qualified name of the type of the attribute.
ATTR_TYPENAME	VARCHAR(128)		
TARGET_TYPESCHEMA	VARCHAR(128)	Yes	Qualified name of the target type, if the type of the attribute is REFERENCE. Null value if the type of the attribute is not REFERENCE.
TARGET_TYPENAME	VARCHAR(128)	Yes	
SOURCE_TYPESCHEMA	VARCHAR(128)		Qualified name of the data type in the data type hierarchy where the attribute was introduced. For non-inherited attributes, these columns are the same as TYPESCHEMA and TYPENAME.
SOURCE_TYPENAME	VARCHAR(128)		
ORDINAL	SMALLINT		Position of the attribute in the definition of the structured data type, starting with zero.
LENGTH	INTEGER		Maximum length of data; 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields.
SCALE	SMALLINT		Scale for DECIMAL fields; 0 if not DECIMAL.
CODEPAGE	SMALLINT		Code page of the attribute. For character-string attributes not defined with FOR BIT DATA, the value is the database code page. For graphic-string attributes, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0.
LOGGED	CHAR(1)		Applies only to attributes whose type is LOB or distinct based on LOB; otherwise blank. Y = Attribute is logged. N = Attribute is not logged.
COMPACT	CHAR(1)		Applies only to attributes whose type is LOB or distinct based on LOB; otherwise blank). Y = Attribute is compacted in storage. N = Attribute is not compacted.
DL_FEATURES	CHAR(10)		Applies to DATALINK type attributes only. Blank for REFERENCE type attributes; otherwise null. Encodes various DATALINK features such as linktype, control mode, recovery, and unlink properties.

SYSCAT.BUFFERPOOLDBPARTITIONS

Contains a row for each database partition in the buffer pool for which the size of the buffer pool on the database partition is different from the default size in SYSCAT.BUFFERPOOLS column NPAGES.

Table 44. SYSCAT.BUFFERPOOLDBPARTITIONS Catalog View

Column Name	Data Type	Nullable	Description
BUFFERPOOLID	INTEGER		Internal buffer pool identifier
DBPARTITIONNUM	SMALLINT		Database partition number
NPAGES	INTEGER		Number of pages in this buffer pool on this database partition

SYSCAT.BUFFERPOOLS

Contains a row for every buffer pool in every database partition group.

Table 45. SYSCAT.BUFFERPOOLS Catalog View

Column Name	Data Type	Nullable	Description
BPNAME	VARCHAR(128)		Name of the buffer pool
BUFFERPOOLID	INTEGER		Internal buffer pool identifier
NGNAME	VARCHAR(128)	Yes	Database partition group name (NULL if the buffer pool exists on all database partitions in the database)
NPAGES	INTEGER		Number of pages in the buffer pool
PAGESIZE	INTEGER		Page size for this buffer pool
ESTORE	CHAR(1)		N = This buffer pool does not use extended storage. Y = This buffer pool uses extended storage.

SYSCAT.CASTFUNCTIONS

Contains a row for each cast function. It does not include built-in cast functions.

Table 46. SYSCAT.CASTFUNCTIONS Catalog View

Column Name	Data Type	Nullable	Description
FROM_TYPESCHEMA	VARCHAR(128)		Qualified name of the data type of the parameter.
| FROM_TYPENAME	VARCHAR(128)		
TO_TYPESCHEMA	VARCHAR(128)		Qualified name of the data type of the result after casting.
| TO_TYPENAME	VARCHAR(128)		
FUNCSCHEMA	VARCHAR(128)		Qualified name of the function.
| FUNCNAME	VARCHAR(128)		
| SPECIFICNAME	VARCHAR(128)		The name of the function instance.
ASSIGN_FUNCTION	CHAR(1)		Y = Implicit assignment function N = Not an assignment function

SYSCAT.CHECKS

Contains one row for each CHECK constraint.

Table 47. SYSCAT.CHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the check constraint (unique within a table.)
DEFINER	VARCHAR(128)		Authorization ID under which the check constraint was defined.
TABSCHEMA	VARCHAR(128)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(128)		
CREATE_TIME	TIMESTAMP		The time at which the constraint was defined. Used in resolving functions that are used in this constraint. No functions will be chosen that were created after the definition of the constraint.
QUALIFIER	VARCHAR(128)		Value of the default schema at time of object definition. Used to complete any unqualified references.
TYPE	CHAR(1)		Type of check constraint: A = System generated check constraint for GENERATED ALWAYS column C = Check constraint F = Functional dependency O = Constraint is an object property
FUNC_PATH	VARCHAR(254)		The current SQL path that was used when the constraint was created.
TEXT	CLOB(64K)		The text of the CHECK clause. See Note 1.

Notes:

1. In the catalog view, the text of the CHECK clause 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.)

SYSCAT.COLAUTH

Contains one or more rows for each user or group who is granted a column level privilege, indicating the type of privilege and whether or not it is grantable.

Table 48. SYSCAT.COLAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
TABSCHEMA	VARCHAR(128)		Qualified name of the table or view.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Name of the column to which this privilege applies.
COLNO	SMALLINT		Number of this column in the table or view.

PRIVTYPE	CHAR(1)		Indicates the type of privilege held on the table or view:
			U = Update privilege
			R = Reference privilege

GRANTABLE	CHAR(1)		Indicates if the privilege is grantable.
			G = Grantable
			N = Not grantable

SYSCAT.COLCHECKS

Each row represents some column that is referenced by a CHECK constraint.

Table 49. SYSCAT.COLCHECKS Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the check constraint. (Unique within a table. May be system generated.)
TABSCHEMA	VARCHAR(128)		Qualified name of table containing referenced column.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Name of column.
USAGE	CHAR(1)		D = Functional dependency child column P = Functional dependency parent column R = Column is referenced in the check constraint. S = Column is a source column in the system generated check constraint that supports a generated column. T = Column is a target column in the system generated check constraint that supports a generated column.

SYSCAT.COLDIST

Contains detailed column statistics for use by the optimizer. Each row describes the Nth-most-frequent value of some column.

Table 50. SYSCAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of the table to which this entry applies.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Name of the column to which this entry applies.

TYPE	CHAR(1)		F = Frequency (most frequent value)
			Q = Quantile value

SEQNO	SMALLINT		• If TYPE = F, then N in this column identifies the Nth most frequent value. • If TYPE = Q, then N in this column identifies the Nth quantile value.
COLVALUE	VARCHAR(254)	Yes	The data value, as a character literal or a null value. See Note 1.
VALCOUNT	BIGINT		• If TYPE = F, then VALCOUNT is the number of occurrences of COLVALUE in the column. • If TYPE = Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE.
DISTCOUNT	BIGINT	Yes	If TYPE = Q, this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable).

Notes:

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.COLGROUPDIST

Contains a row for every value of a column in a column group that makes up the *n*th most frequent value of the column group or the *n*th quantile of the column group.

Table 51. SYSCAT.COLGROUPDIST Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Internal identifier of the column group.

TYPE	CHAR(1)		F = Frequency value
			Q = Quantile value

ORDINAL	SMALLINT		Ordinal number of the column in the group.
SEQNO	SMALLINT		Sequence number *n* representing the *n*th TYPE value.
COLVALUE	VARCHAR(254)	Yes	Data value as a character literal or a null value.

SYSCAT.COLGROUPDISTCOUNTS

Contains a row for the distribution statistics that apply to the *n*th most frequent value of a column group, or the *n*th quantile of a column group.

Table 52. SYSCAT.COLGROUPDISTCOUNTS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPID	INTEGER		Internal identifier of the column group.

TYPE	CHAR(1)		F = Frequency value
			Q = Quantile value

SEQNO	SMALLINT		Sequence number *n* representing the *n*th TYPE value.
VALCOUNT	BIGINT		If TYPE = F, VALCOUNT is the number of occurrences of COLVALUE for the column group with this SEQNO. If TYPE = Q, VALCOUNT is the number of rows whose value is less than or equal to COLVALUE for the column group with this SEQNO.
DISTCOUNT	BIGINT	Yes	If TYPE = Q, this column records the number of distinct values that are less than or equal to COLVALUE for the column group with this SEQNO (null if unavailable).

SYSCAT.COLGROUPS

Contains a row for every column group, and statistics that apply to the entire column group.

Table 53. SYSCAT.COLGROUPS Catalog View

Column Name	Data Type	Nullable	Description
COLGROUPSCHEMA	VARCHAR(128)		Qualified name of the column group.
COLGROUPNAME	VARCHAR(128)		
COLGROUPID	INTEGER		Internal identifier of 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

Contains one row for each identity column that is defined for a table.

Table 54. Columns in SYSCAT.COLIDENTATTRIBUTES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of the table or view that contains the column.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Column name.
START	DECIMAL(31,0)		Starting value.
INCREMENT	DECIMAL(31,0)		Increment value.
MINVALUE	DECIMAL(31,0)		Minimum value.
MAXVALUE	DECIMAL(31,0)		Maximum value.
CYCLE	CHAR(1)		Whether cycling will occur when a boundary is reached: Y - cycling will occur N - cycling will not occur
CACHE	INTEGER		Number of sequence values to preallocate in memory for faster access. 0 indicates that values are not preallocated.
SEQID	INTEGER		Internal ID of the sequence.

SYSCAT.COLOPTIONS

Each row contains column specific option values.

Table 55. SYSCAT.COLOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified nickname for the column.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Local column name.
OPTION	VARCHAR(128)		Name of the column option.
SETTING	VARCHAR(255)		Value for the column option.

SYSCAT.COLUMNS

Contains one row for each column (including inherited columns, where applicable) that is defined for a table or view. All of the catalog views have entries in the SYSCAT.COLUMNS table.

Table 56. SYSCAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of the table or view that contains the column.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Column name.
COLNO	SMALLINT		Numerical place of column in table or view, beginning at zero.
TYPESCHEMA	VARCHAR(128)		Contains the qualified name of the type, if the data type of the column is distinct. Otherwise TYPESCHEMA contains the value SYSIBM and TYPENAME contains the data type of the column (in long form, for example, CHARACTER). If FLOAT or FLOAT(*n*) with *n* greater than 24 is specified, TYPENAME is renamed to DOUBLE. If FLOAT(*n*) with *n* less than 25 is specified, TYPENAME is renamed to REAL. Also, NUMERIC is renamed to DECIMAL.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Maximum length of data. 0 for distinct types. The LENGTH column indicates precision for DECIMAL fields.
SCALE	SMALLINT		Scale for DECIMAL fields; 0 if not DECIMAL.
DEFAULT	VARCHAR(254)	Yes	Default value for the column of a table expressed as a constant, special register, or cast-function appropriate for the data type of the column. May also be the keyword NULL. Values may be converted from what was specified as a default value. For example, date and time constants are presented in ISO format and cast-function names are qualified with schema name and the identifiers are delimited (see Note 3). Null value if a DEFAULT clause was not specified or the column is a view column.
NULLS	CHAR(1)		Y = Column is nullable. N = Column is not nullable. The value can be N for a view column that is derived from an expression or function. Nevertheless, such a column allows nulls when the statement using the view is processed with warnings for arithmetic errors. See Note 1.

Table 56. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
CODEPAGE	SMALLINT		Code page of the column. For character-string columns not defined with the FOR BIT DATA attribute, the value is the database code page. For graphic-string columns, the value is the DBCS code page implied by the (composite) database code page. Otherwise, the value is 0.
LOGGED	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise). Y=Column is logged. N=Column is not logged.
COMPACT	CHAR(1)		Applies only to columns whose type is LOB or distinct based on LOB (blank otherwise). Y = Column is compacted in storage. N = Column is not compacted.
COLCARD	BIGINT		Number of distinct values in the column; −1 if statistics are not gathered; −2 for inherited columns and columns of H-tables.
HIGH2KEY	VARCHAR(254)	Yes	Second highest value of the column. This field is empty if statistics are not gathered, and for inherited columns and columns of H-tables. See Notes 2 and 4.
LOW2KEY	VARCHAR(254)	Yes	Second lowest value of the column. This field is empty if statistics are not gathered, and for inherited columns and columns of H-tables. See Notes 2 and 4.
AVGCOLLEN	INTEGER		Average space required for the column length. −1 if a long field or LOB, or statistics have not been collected; −2 for inherited columns and columns of H-tables.
KEYSEQ	SMALLINT	Yes	The column's numerical position within the table's primary key. This field is null for subtables and hierarchy tables.
PARTKEYSEQ	SMALLINT	Yes	The column's numerical position within the table's partitioning key. This field is null or 0 if the column is not part of the partitioning key. This field is also null for subtables and hierarchy tables.
NQUANTILES	SMALLINT		Number of quantile values recorded in SYSCAT.COLDIST for this column; −1 if no statistics; −2 for inherited columns and columns of H-tables.
NMOSTFREQ	SMALLINT		Number of most-frequent values recorded in SYSCAT.COLDIST for this column; −1 if no statistics; −2 for inherited columns and columns of H-tables.
NUMNULLS	BIGINT		Contains the number of nulls in a column. −1 if statistics are not gathered.
TARGET_TYPESCHEMA	VARCHAR(128)	Yes	Qualified name of the target type, if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE.
TARGET_TYPENAME	VARCHAR(18)	Yes	

Table 56. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
SCOPE_TABSCHEMA	VARCHAR(128)	Yes	Qualified name of the scope (target table), if the type of the column is REFERENCE. Null value if the type of the column is not REFERENCE or the scope is not defined.
SCOPE_TABNAME	VARCHAR(128)	Yes	
SOURCE_TABSCHEMA	VARCHAR(128)		Qualified name of the table or view in the respective hierarchy where the column was introduced. For non-inherited columns, the values are the same as TBCREATOR and TBNAME. Null for columns of non-typed tables and views
SOURCE_TABNAME	VARCHAR(128)		
DL_FEATURES	CHAR(10)	Yes	Applies to DATALINK type columns only. Null otherwise. Each character position is defined as follows: 1. Link type (U for URL) 2. Link control (F for file, N for no) 3. Integrity (A for all, N for none) 4. Read permission (F for file system, D for database) 5. Write permission (F for file system, B for blocked, A for admin requiring token for update, N for admin not requiring token for update) 6. Recovery (Y for yes, N for no) 7. On unlink (R for restore, D for delete, N for not applicable) Characters 8 through 10 are reserved for future use.
SPECIAL_PROPS	CHAR(8)	Yes	Applies to REFERENCE type columns only. Null otherwise. Each character position is defined as follows: Object identifier (OID) column (Y for yes, N for no) User generated or system generated (U for user, S for system)
HIDDEN	CHAR(1)		Type of hidden column S = System managed hidden column Blank if column is not hidden
INLINE_LENGTH	INTEGER		Length of structured type column that can be kept with base table row. 0 if no value explicitly set by ALTER/CREATE TABLE statement.
IDENTITY	CHAR(1)		'Y' indicates that the column is an identity column; 'N' indicates that the column is not an identity column.
GENERATED	CHAR(1)		Type of generated column A = Column value is always generated D = Column value is generated by default Blank if column is not generated
COMPRESS	CHAR(1)		S = Compress system default values O = Compress off

Table 56. SYSCAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TEXT	CLOB(64K)		Contains the text of the generated column, starting with the keyword AS.
REMARKS	VARCHAR(254)	Yes	User-supplied comment.
AVGDISTINCTPERPAGE	DOUBLE	Yes	For future use.
PAGEVARIANCERATIO	DOUBLE	Yes	For future use.
SUB_COUNT	SMALLINT		Average number of sub-elements. Only applicable for character columns. For example, consider the following string: ’database simulation analytical business intelligence’. In this example, SUB_COUNT = 5, because there are 5 sub-elements in the string.
SUB_DELIM_LENGTH	SMALLINT		Average length of each delimiter separating each sub-element. Only applicable for character columns. For example, consider the following string: ’database simulation analytical business intelligence’. In this example, SUB_DELIM_LENGTH = 1, because each delimiter is a single blank.

Notes:

1. Starting with Version 2, value D (indicating not null with a default) is no longer used. Instead, use of WITH DEFAULT is indicated by a non-null value in the DEFAULT column.
2. Starting with Version 2, representation of numeric data has been changed to character literals. The size has been enlarged from 16 to 33 bytes.
3. For Version 2.1.0, cast-function names were not delimited and may still appear this way in the DEFAULT column. Also, some view columns included default values which will still appear in the DEFAULT column.
4. 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.

SYSCAT.COLUSE

Contains a row for every column that participates in the DIMENSIONS clause of the CREATE TABLE statement.

Table 57. SYSCAT.COLUSE Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of the table containing the column
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Name of the column
DIMENSION	SMALLINT		Dimension number, based on the order of dimensions specified in the DIMENSIONS clause (initial position = 0). For a composite dimension, this value will be the same for each component of the dimension.
COLSEQ	SMALLINT		Numeric position of the column in the dimension to which it belongs (initial position = 0). The value is 0 for the single column in a noncomposite dimension.
TYPE	CHAR(1)		Type of dimension. C = clustering/multi-dimensional clustering (MDC)

SYSCAT.CONSTDEP

Contains a row for every dependency of a constraint on some other object.

Table 58. SYSCAT.CONSTDEP Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint.
TABSCHEMA	VARCHAR(128)		Qualified name of the table to which the constraint applies.
TABNAME	VARCHAR(128)		

BTYPE	CHAR(1)		Type of object that the constraint depends on. Possible values:
			F = Function instance
			I = Index instance
			R = Structured type

BSCHEMA	VARCHAR(128)		Qualified name of object that the constraint depends on.
BNAME	VARCHAR(18)		

SYSCAT.DATATYPES

Contains a row for every data type, including built-in and user-defined types.

Table 59. SYSCAT.DATATYPES Catalog View

Column Name	Data Type	Nullable	Description
TYPESCHEMA	VARCHAR(128)		Qualified name of the data type (for built-in types, TYPESCHEMA is SYSIBM).
TYPENAME	VARCHAR(18)		
DEFINER	VARCHAR(128)		Authorization ID under which type was created.
SOURCESCHEMA	VARCHAR(128)	Yes	Qualified name of the source type for distinct types. Qualified name of the builtin type used as the reference type that is used as the representation for references to structured types. Null for other types.
SOURCENAME	VARCHAR(18)	Yes	

METATYPE	CHAR(1)		S = System predefined type
			T = Distinct type
			R = Structured type

TYPEID	SMALLINT		The system generated internal identifier of the data type.
SOURCETYPEID	SMALLINT	Yes	Internal type ID of source type (null for built-in types). For user-defined structured types, this is the internal type ID of the reference representation type.
LENGTH	INTEGER		Maximum length of the type. 0 for system predefined parameterized types (for example, DECIMAL and VARCHAR). For user-defined structured types, this indicates the length of the reference representation type.
SCALE	SMALLINT		Scale for distinct types or reference representation types based on the system predefined DECIMAL type. 0 for all other types (including DECIMAL itself). For user-defined structured types, this indicates the length of the reference representation type.
CODEPAGE	SMALLINT		Code page for character and graphic distinct types or reference representation types; 0 otherwise.
CREATE_TIME	TIMESTAMP		Creation time of the data type.
ATTRCOUNT	SMALLINT		Number of attributes in data type.
INSTANTIABLE	CHAR(1)		Y = Type can be instantiated. N = Type can not be instantiated.
WITH_FUNC_ACCESS	CHAR(1)		Y = All the methods for this type can be invoked using function notation. N = Methods for this type can not be invoked using function notation.

Table 59. SYSCAT.DATATYPES Catalog View (continued)

Column Name	Data Type	Nullable	Description
FINAL	CHAR(1)		Y = User-defined type can not have subtypes. N = User-defined type can have subtypes.
INLINE_LENGTH	INTEGER		Length of structured type that can be kept with base table row. 0 if no value explicitly set by CREATE TYPE statement.
NATURAL_INLINE_LENGTH	INTEGER		System-calculated inline length of the structured type.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

SYSCAT.DBAUTH

Records the database authorities held by users.

Table 60. SYSCAT.DBAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		SYSIBM or authorization ID of the user who granted the privileges.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
DBADMAUTH	CHAR(1)		Whether grantee holds DBADM authority over the database: Y = Authority is held. N = Authority is not held.
CREATETABAUTH	CHAR(1)		Whether grantee can create tables in the database (CREATETAB): Y = Privilege is held. N = Privilege is not held.
BINDADDAUTH	CHAR(1)		Whether grantee can create new packages in the database (BINDADD): Y = Privilege is held. N = Privilege is not held.
CONNECTAUTH	CHAR(1)		Whether grantee can connect to the database (CONNECT): Y = Privilege is held. N = Privilege is not held.
NOFENCEAUTH	CHAR(1)		Whether grantee holds privilege to create non-fenced functions. Y = Privilege is held. N = Privilege is not held.
IMPLSCHEMAAUTH	CHAR(1)		Whether grantee can implicitly create schemas in the database (IMPLICIT_SCHEMA): Y = Privilege is held. N = Privilege is not held.
LOADAUTH	CHAR(1)		Whether grantee holds LOAD authority over the database: Y = Authority is held. N = Authority is not held.
EXTERNALROUTINEAUTH	CHAR(1)		Whether grantee can create external routines (CREATE_EXTERNAL_ROUTINE): Y = Authority is held. N = Authority is not held.
QUIESCECONNECTAUTH	CHAR(1)		Whether grantee can connect to a database (QUIESCE_CONNECT): Y = Authority is held. N = Authority is not held.

SYSCAT.DBPARTITIONGROUPDEF

Contains a row for each partition that is contained in a database partition group.

Table 61. SYSCAT.DBPARTITIONGROUPDEF Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR(18)		The name of the database partition group that contains the database partition.
DBPARTITIONNUM	SMALLINT		The partition number of a partition 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 partition is not in the partitioning map but the containers for the table spaces in the database partition group are created. The partition is added to the partitioning map when a redistribute database partition group operation is successfully completed. D = The partition will be dropped when a redistribute database partition group operation is completed. T = The newly added partition is not in the partitioning map and it was added using the WITHOUT TABLESPACES clause. Containers must be specifically added to the table spaces for the database partition group. Y = The partition is in the partitioning map.

SYSCAT.DBPARTITIONGROUPS

Contains a row for each database partition group.

Table 62. SYSCAT.DBPARTITIONGROUPS Catalog View

Column Name	Data Type	Nullable	Description
DBPGNAME	VARCHAR(18)		Name of the database partition group.
DEFINER	VARCHAR(128)		Authorization ID of the database partition group definer.
PMAP_ID	SMALLINT		Identifier of the partitioning map in SYSCAT.PARTITIONMAPS.
REDISTRIBUTE_PMAP_ID	SMALLINT		Identifier of the partitioning map currently being used for redistribution. Value is -1 if redistribution is currently not in progress.
CREATE_TIME	TIMESTAMP		Creation time of database partition group.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

SYSCAT.EVENTMONITORS

Contains a row for every event monitor that has been defined.

Table 63. SYSCAT.EVENTMONITORS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR(18)		Name of event monitor.
DEFINER	VARCHAR(128)		Authorization ID of definer of event monitor.
TARGET_TYPE	CHAR(1)		The type of target to which event data is written. Values: F = File; P = Pipe; T = Table
TARGET	VARCHAR(246)		Name of the target to which event data is written. Absolute pathname of file, or absolute name of pipe.
MAXFILES	INTEGER	Yes	Maximum number of event files that this event monitor permits in an event path. Null if there is no maximum, or if the target-type is not FILE.
MAXFILESIZE	INTEGER	Yes	Maximum size (in 4K pages) that each event file can reach before the event monitor creates a new file. Null if there is no maximum, or if the target-type is not FILE.
BUFFERSIZE	INTEGER	Yes	Size of buffers (in 4K pages) used by event monitors with file targets; otherwise null.
IO_MODE	CHAR(1)	Yes	Mode of file I/O. B = Blocked; N = Not blocked. Null if target-type is not FILE.
WRITE_MODE	CHAR(1)	Yes	Indicates how this monitor handles existing event data when the monitor is activated. Values: A = Append; R = Replace. Null if target-type is not FILE.
AUTOSTART	CHAR(1)		The event monitor will be activated automatically when the database starts. Y = Yes; N = No
DBPARTITIONNUM	SMALLINT		The number of the database partition on which the event monitor runs and logs events.
MONSCOPE	CHAR(1)		Monitoring scope: L = Local; G = Global; T = Per node where table space exists; A blank character, valid only for WRITE TO TABLE event monitors.
EVMON_ACTIVATES	INTEGER		The number of times this event monitor has been activated.
REMARKS	VARCHAR(254)	Yes	Reserved for future use.

SYSCAT.EVENTS

Contains a row for every event that is being monitored. An event monitor, in general, monitors multiple events.

Table 64. SYSCAT.EVENTS Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR(18)		Name of event monitor that is monitoring this event.
TYPE	VARCHAR(18)		Type of event being monitored. Possible values: DATABASE CONNECTIONS TABLES STATEMENTS TRANSACTIONS DEADLOCKS DETAILDEADLOCKS TABLESPACES
FILTER	CLOB(32K)	Yes	The full text of the WHERE-clause that applies to this event.

SYSCAT.EVENTTABLES

Contains a row for every target table of an event monitor that writes to SQL tables.

Table 65. SYSCAT.EVENTTABLES Catalog View

Column Name	Data Type	Nullable	Description
EVMONNAME	VARCHAR(128)		Name of event monitor.
LOGICAL_GROUP	VARCHAR(18)		Name of the logical data group. This can be one of: BUFFERPOOL, CONN, CONNHEADER, CONTROL, DB, DEADLOCK, DLCONN, DLLOCK, STMT, SUBSECTION, TABLE, TABLESPACE, XACT
TABSCHEMA	VARCHAR(128)		Qualified name of the target table.
TABNAME	VARCHAR(128)		
PCTDEACTIVATE	SMALLINT		A percent value that specifies how full a DMS table space must be before an event monitor automatically deactivates. Set to 100 for SMS table spaces.

SYSCAT.FULLHIERARCHIES

Each row represents the relationship between a subtable and a supertable, a subtype and a supertype, or a subview and a superview. All hierarchical relationships, including immediate ones, are included in this view

Table 66. SYSCAT.FULLHIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description

METATYPE	CHAR(1)		Encodes the type of relationship:
			R = Between structured types
			U = Between typed tables
			W = Between typed views

SUB_SCHEMA	VARCHAR(128)		Qualified name of subtype, subtable or subview.
SUB_NAME	VARCHAR(128)		
SUPER_SCHEMA	VARCHAR(128)		Qualified name of supertype, supertable or superview.
SUPER_NAME	VARCHAR(128)		
ROOT_SCHEMA	VARCHAR(128)		Qualified name of the table, view or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR(128)		

SYSCAT.FUNCMAPOPTIONS

Each row contains function mapping option values.

Table 67. SYSCAT.FUNCMAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR(18)		Function mapping name.
OPTION	VARCHAR(128)		Name of the function mapping option.
SETTING	VARCHAR(255)		Value of the function mapping option.

SYSCAT.FUNCMAPPARMOPTIONS

Each row contains function mapping parameter option values.

Table 68. SYSCAT.FUNCMAPPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_MAPPING	VARCHAR(18)		Name of function mapping.
ORDINAL	SMALLINT		Position of parameter

LOCATION	CHAR(1)		L = Local
			R = Remote

OPTION	VARCHAR(128)		Name of the function mapping parameter option.
SETTING	VARCHAR(255)		Value of the function mapping parameter option.

SYSCAT.FUNCMAPPINGS

Each row contains function mappings.

Table 69. SYSCAT.FUNCMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
FUNCTION_ MAPPING	VARCHAR(128)		Name of function mapping (may be system generated).
FUNCSCHEMA	VARCHAR(128)	Yes	Function schema. Null if system built-in function.
FUNCNAME	VARCHAR(1024)	Yes	Name of the local function (built-in or user-defined).
FUNCID	INTEGER	Yes	Internally assigned identifier.
SPECIFICNAME	VARCHAR(128)	Yes	Name of the local function instance.
DEFINER	VARCHAR(128)		Authorization ID under which this mapping was created.
WRAPNAME	VARCHAR(128)	Yes	Wrapper name to which the mapping is applied.
SERVERNAME	VARCHAR(128)	Yes	Name of the data source.
SERVERTYPE	VARCHAR(30)	Yes	Type of data source to which mapping is applied.
SERVERVERSION	VARCHAR(18)	Yes	Version of the server type to which mapping is applied.
CREATE_TIME	TIMESTAMP	Yes	Time at which the mapping is created.
REMARKS	VARCHAR(254)	Yes	User supplied comment, or null.

SYSCAT.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 70. SYSCAT.HIERARCHIES Catalog View

Column Name	Data Type	Nullable	Description
METATYPE	CHAR(1)		Encodes the type of relationship: R = Between structured types; U = Between typed tables; W = Between typed views
SUB_SCHEMA	VARCHAR(128)		Qualified name of subtype, subtable, or subview.
SUB_NAME	VARCHAR(128)		
SUPER_SCHEMA	VARCHAR(128)		Qualified name of supertype, supertable, or superview.
SUPER_NAME	VARCHAR(128)		
ROOT_SCHEMA	VARCHAR(128)		Qualified name of the table, view or type that is at the root of the hierarchy.
ROOT_NAME	VARCHAR(128)		

SYSCAT.INDEXAUTH

Contains a row for every privilege held on an index.

Table 71. SYSCAT.INDEXAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privileges.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
INDSCHEMA	VARCHAR(128)		Qualified name of the index.
INDNAME	VARCHAR(18)		
CONTROLAUTH	CHAR(1)		Whether grantee holds CONTROL privilege over the index: Y = Privilege is held. N = Privilege is not held.

SYSCAT.INDEXCOLUSE

Lists all columns that participate in an index.

Table 72. SYSCAT.INDEXCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR(128)		Qualified name of the index.
INDNAME	VARCHAR(18)		
COLNAME	VARCHAR(128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the index (initial position = 1).

COLORDER	CHAR(1)		Order of the values in this column in the index. Values:
			A = Ascending
			D = Descending
			I = INCLUDE column(ordering ignored)

SYSCAT.INDEXDEP

Each row represents a dependency of an index on some other object.

Table 73. SYSCAT.INDEXDEP Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR(128)		Qualified name of the index that has dependencies on another object.
INDNAME	VARCHAR(18)		

BTYPE	CHAR(1)		Type of object on which the index depends.
			A = Alias
			F = Function instance
			O = Privilege dependency on all subtables or subviews in a table or view hierarchy
			R = Structured type
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view
			X = Index extension

BSCHEMA	VARCHAR(128)		Qualified name of the object on which the index has a dependency.
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE = O, S, T, U, V, or W, encodes the privileges on the table or view that are required by the dependent index; otherwise null.

SYSCAT.INDEXES

Contains one row for each index (including inherited indexes, where applicable) that is defined for a table.

Table 74. SYSCAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR(128)		Name of the index.
INDNAME	VARCHAR(18)		
DEFINER	VARCHAR(128)		User who created the index.
TABSCHEMA	VARCHAR(128)		Qualified name of the table or nickname on which the index is defined.
TABNAME	VARCHAR(128)		
COLNAMES	VARCHAR(640)		List of column names, each preceded by + or − to indicate ascending or descending order respectively. Warning: This column will be removed in the future. Use SYSCAT.INDEXCOLUSE for this information.

UNIQUERULE	CHAR(1)		Unique rule:
			D = Duplicates allowed
			P = Primary index
			U = Unique entries only allowed

MADE_UNIQUE	CHAR(1)		Y = Index was originally non-unique but was converted to a unique index to support a unique or primary key constraint. If the constraint is dropped, the index will revert to non-unique. N = Index remains as it was created.
COLCOUNT	SMALLINT		Number of columns in the key, plus the number of include columns, if any.
UNIQUE_COLCOUNT	SMALLINT		The number of columns required for a unique key. Always <=COLCOUNT. < COLCOUNT only if there are include columns. −1 if the index has no unique key (permits duplicates).

INDEXTYPE	CHAR(4)		Type of index.
			CLUS = Clustering
			REG = Regular
			DIM = Dimension block index
			BLOK = Block index

ENTRYTYPE	CHAR(1)		H = An index on a hierarchy table (H-table) L = Logical index on a typed table blank if an index on an untyped table
PCTFREE	SMALLINT		Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.
IID	SMALLINT		Internal index ID.

Table 74. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NLEAF	INTEGER		Number of leaf pages; −1 if statistics are not gathered.
NLEVELS	SMALLINT		Number of index levels; −1 if statistics are not gathered.
FIRSTKEYCARD	BIGINT		Number of distinct first key values; −1 if statistics are not gathered.
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index; −1 if no statistics, or if not applicable.
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index; −1 if no statistics, or if not applicable.
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index; −1 if no statistics, or if not applicable.
FULLKEYCARD	BIGINT		Number of distinct full key values; −1 if statistics are not gathered.
CLUSTERRATIO	SMALLINT		Degree of data clustering with the index; −1 if statistics are not gathered, or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR	DOUBLE		Finer measurement of degree of clustering, or −1 if detailed index statistics have not been gathered, or if the index is defined on a nickname.
SEQUENTIAL_PAGES	INTEGER		Number of leaf pages located on disk in index key order with few or no large gaps between them; −1 if no statistics are available.
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100; −1 if no statistics are available.)
USER_DEFINED	SMALLINT		1 if this index was defined by a user and has not been dropped; otherwise 0.

Table 74. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SYSTEM_REQUIRED	SMALLINT		Valid values are: 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		Time when the index was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this index. Null if no statistics available.
PAGE_FETCH_PAIRS	VARCHAR(254)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
MINPCTUSED	SMALLINT		If not zero, online index defragmentation is enabled, and the value is the threshold of minimum used space before merging pages.
REVERSE_SCANS	CHAR(1)		Y = Index supports reverse scans N = Index does not support reverse scans
INTERNAL_FORMAT	SMALLINT		Valid values are: 1 if the index does not have backward pointers >= 2 if the index has backward pointers 6 if the index is a composite block index
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.
IESCHEMA	VARCHAR(128)	Yes	Qualified name of index extension. Null for ordinary indexes.
IENAME	VARCHAR(18)	Yes	
IEARGUMENTS	CLOB(32K)	Yes	External information of the parameter specified when the index is created. Null for ordinary indexes.
INDEX_OBJECTID	INTEGER		Index object identifier for the table.

Table 74. SYSCAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NUMRIDS	BIGINT		Total number of row identifiers (RIDs) in the index.
NUMRIDS_DELETED	BIGINT		Total number of row identifiers in the index that are marked as deleted, excluding those row identifiers on leaf pages on which all row identifiers are as marked deleted.
NUM_EMPTY_LEAFS	BIGINT		Total number of index leaf pages that have all of their row identifiers marked as deleted.
AVERAGE_RANDOM_ FETCH_PAGES	DOUBLE		Average number of random table pages between sequential page accesses when fetching using the index; −1 if it is not known. See Notes 1 and 2.
AVERAGE_RANDOM_ PAGES	DOUBLE		Average number of random index pages between sequential index page accesses; −1 if it is not known. See Note 2.
AVERAGE_SEQUENCE_ GAP	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 it is not known. See Note 2.
AVERAGE_SEQUENCE_ FETCH_GAP	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 it is not known. See Notes 1 and 2.
AVERAGE_SEQUENCE_ PAGES	DOUBLE		Average number of index pages accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); −1 if it is not known. See Note 2.
AVERAGE_SEQUENCE_ FETCH_PAGES	DOUBLE		Average number of table pages 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 it is not known. See Notes 1 and 2.
TBSPACEID	INTEGER		Internal identifier for the index table space.

Notes:

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.

Related reference:

- "SYSCAT.INDEXCOLUSE" on page 562

SYSCAT.INDEXEXPLOITRULES

Each row represents an index exploitation.

Table 75. SYSCAT.INDEXEXPLOITRULES Catalog View

Column Name	Data Type	Nullable	Description
FUNCID	INTEGER		Function ID.
SPECID	SMALLINT		Number of the predicate specification in the CREATE FUNCTION statement.
IESCHEMA	VARCHAR(128)		Qualified name of the index extension.
IENAME	VARCHAR(18)		
RULEID	SMALLINT		Unique exploitation rule ID.
SEARCHMETHODID	SMALLINT		The search method ID in the specific index extension.
SEARCHKEY	VARCHAR(320)		Key used to exploit index.
SEARCHARGUMENT	VARCHAR(1800)		Search arguments used in the index exploitation.
EXACT	CHAR(1)		Indicates whether or not index look-up is exact in terms of predicate evaluation. Possible values are: Y = Index look-up is exact. N = Index look-up is not exact.

SYSCAT.INDEXEXTENSIONDEP

Contains a row for each dependency that index extensions have on various database objects.

Table 76. SYSCAT.INDEXEXTENSIONDEP Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR(128)		Qualified name of the index extension that has dependencies on another object.
IENAME	VARCHAR(18)		

BTYPE	CHAR(1)		Type of object on which the index extension is dependent:
			A = Alias
			F = Function instance or method instance
			J = Server definition
			O = "Outer" dependency on hierarchic SELECT privilege
			R = Structured type
			S = Materialized query table
			T = Table (not typed)
			U = Typed table
			V = View (not typed)
			W = Typed view
			X = Index extension

BSCHEMA	VARCHAR(128)		Qualified name of the object on which the index extension depends. (If BTYPE='F', this is the specific name of a function.)
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE='O', 'T', 'U', 'V', or 'W', encodes the privileges on the table (or view) that are required by a dependent trigger; otherwise null.

SYSCAT.INDEXEXTENSIONMETHODS

Each row represents a search method. One index extension may include multiple search methods.

Table 77. SYSCAT.INDEXEXTENSIONMETHODS Catalog View

Column Name	Data Type	Nullable	Description
METHODNAME	VARCHAR(18)		Name of search method.
METHODID	SMALLINT		Number of the method in the index extension.
IESCHEMA	VARCHAR(128)		Qualified name of index extension.
IENAME	VARCHAR(18)		
RANGEFUNCSCHEMA	VARCHAR(128)		Qualified name of range-through function.
RANGEFUNCNAME	VARCHAR(18)		
RANGESPECIFICNAME	VARCHAR(18)		Range-through function specific name.
FILTERFUNCSCHEMA	VARCHAR(128)		Qualified name of filter function.
FILTERFUNCNAME	VARCHAR(18)		
FILTERSPECIFICNAME	VARCHAR(18)		Function specific name of filter function.
REMARKS	VARCHAR(254)	Yes	User-supplied or null.

SYSCAT.INDEXEXTENSIONPARMS

Each row represents an index extension instance parameter or source key definition.

Table 78. SYSCAT.INDEXEXTENSIONPARMS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR(128)		Qualified name of index extension.
IENAME	VARCHAR(18)		
ORDINAL	SMALLINT		Sequence number of parameter or source key.
PARMNAME	VARCHAR(18)		Name of parameter or source key.
TYPESCHEMA	VARCHAR(128)		Qualified name of the instance parameter or source key data type.
TYPENAME	VARCHAR(18)		
LENGTH	INTEGER		Length of the instance parameter or source key data type.
SCALE	SMALLINT		Scale of the instance parameter or source key data type. Zero (0) when not applicable.

PARMTYPE	CHAR(1)		Type represented by the row:
			P = index extension parameter
			K = key column

CODEPAGE	SMALLINT		Code page of the index extension parameter. Zero if not a string type.

SYSCAT.INDEXEXTENSIONS

Contains a row for each index extension.

Table 79. SYSCAT.INDEXEXTENSIONS Catalog View

Column Name	Data Type	Nullable	Description
IESCHEMA	VARCHAR(128)		Qualified name of index extension.
IENAME	VARCHAR(18)		
DEFINER	VARCHAR(128)		Authorization ID under which the index extension was defined.
CREATE_TIME	TIMESTAMP		Time at which the index extension was defined.
KEYGENFUNCSCHEMA	VARCHAR(128)		Qualified name of key generation function.
KEYGENFUNCNAME	VARCHAR(18)		
KEYGENSPECIFICNAME	VARCHAR(18)		Key generation function specific name.
TEXT	CLOB(64K)		The full text of the CREATE INDEX EXTENSION statement.
REMARKS	VARCHAR(254)		User-supplied comment, or null.

SYSCAT.INDEXOPTIONS

Each row contains index specific option values.

Table 80. SYSCAT.INDEXOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
INDSCHEMA	VARCHAR(128)		Schema name of the index.
INDNAME	VARCHAR(18)		Local name of the index.
OPTION	VARCHAR(128)		Name of the index option.
SETTING	VARCHAR(255)		Value.

SYSCAT.KEYCOLUSE

Lists all columns that participate in a key (including inherited primary or unique keys where applicable) defined by a unique, primary key, or foreign key constraint.

Table 81. SYSCAT.KEYCOLUSE Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	VARCHAR(128)		Qualified name of the table containing the column.
TABNAME	VARCHAR(128)		
COLNAME	VARCHAR(128)		Name of the column.
COLSEQ	SMALLINT		Numeric position of the column in the key (initial position=1).

SYSCAT.NAMEMAPPINGS

Each row represents the mapping between logical objects and the corresponding implementation objects that implement the logical objects.

Table 82. SYSCAT.NAMEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description

TYPE	CHAR(1)		C = Column
			I = Index
			U = Typed table

LOGICAL_SCHEMA	VARCHAR(128)		Qualified name of the logical object.
LOGICAL_NAME	VARCHAR(128)		
LOGICAL_COLNAME	VARCHAR(128)	Yes	If TYPE = C, then the name of the logical column. Otherwise null.
IMPL_SCHEMA	VARCHAR(128)		Qualified name of the implementation object that implements the logical object.
IMPL_NAME	VARCHAR(128)		
IMPL_COLNAME	VARCHAR(128)	Yes	If TYPE = C, then the name of the implementation column. Otherwise null.

SYSCAT.PACKAGEAUTH

Contains a row for every privilege held on a package.

Table 83. SYSCAT.PACKAGEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privileges.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
PKGSCHEMA	VARCHAR(128)		Name of the package on which the privileges are held.
PKGNAME	CHAR(8)		
CONTROLAUTH	CHAR(1)		Indicates whether grantee holds CONTROL privilege on the package: Y = Privilege is held. N = Privilege is not held.
BINDAUTH	CHAR(1)		Indicates whether grantee holds BIND privilege on the package: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
EXECUTEAUTH	CHAR(1)		Indicates whether grantee holds EXECUTE privilege on the package: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.

SYSCAT.PACKAGEDEP

Contains a row for each dependency that packages have on indexes, tables, views, triggers, functions, aliases, types, and hierarchies.

Table 84. SYSCAT.PACKAGEDEP Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR(128)		Name of the package.
PKGNAME	CHAR(8)		
UNIQUEID	CHAR(8)		Internal date and time information indicating when the package was first created. Useful for identifying a specific package when multiple packages having the same name exist.
PKGVERSION	VARCHAR(64)		Version identifier of the package.
BINDER	VARCHAR(128)	Yes	Binder of the package.

BTYPE	CHAR(1)		Type of object BNAME:
			A = Alias
			B = Trigger
			D = Server definition
			F = Function instance
			I = Index
			M = Function mapping
			N = Nickname
			O = Privilege dependency on all subtables or subviews in a table or view hierarchy
			P = Page size
			R = Structured type
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view

BSCHEMA	VARCHAR(128)		Qualified name of an object on which the package depends.
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE is O, S, T, U, V or W then it encodes the privileges that are required by this package (Select, Insert, Delete, Update).

Note: 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

Contains a row for each package that has been created by binding an application program.

Table 85. SYSCAT.PACKAGES Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR(128)		Schema qualifier of the package name.
PKGNAME	CHAR(8)		Unqualified identifier of the package name
PKGVERSION	VARCHAR(64)		Version identifier of the package name.
BOUNDBY	VARCHAR(128)		Authorization ID (OWNER) of the binder of the package.
DEFINER	VARCHAR(128)		User ID under which the package was bound.
DEFAULT_SCHEMA	VARCHAR(128)		Default schema (QUALIFIER) name used for unqualified names in static SQL statements.

VALID	CHAR(1)		Y = Valid
			N = Not valid
			X = Package is inoperative because some function instance on which it depends has been dropped. Explicit rebind is needed. (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.)

UNIQUE_ID	CHAR(8)		Internal date and time information indicating when the package was first created. Useful for identifying a specific package when multiple packages having the same name exist.
TOTAL_SECT	SMALLINT		Total 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 date and time 2 = EUR date, EUR time 3 = ISO date, ISO time 4 = JIS date, JIS time 5 = LOCAL date, LOCAL time

ISOLATION	CHAR(2)	Yes	Isolation level:
			RR = Repeatable read
			RS = Read stability
			CS = Cursor stability
			UR = Uncommitted read

Table 85. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description

BLOCKING	CHAR(1)	Yes	Cursor blocking option:
			N = No blocking
			U = Block unambiguous cursors
			B = Block all cursors

INSERT_BUF	CHAR(1)		Insert option used during bind: Y = Inserts are buffered N = Inserts are not buffered
REOPTVAR	CHAR(1)		Indicates whether the access path is reoptimized at execution time using input variable values: A = Access path is reoptimized at every OPEN or EXECUTE request. N = Access path is not reoptimized at execute time. O = Access path is reoptimized only at first OPEN or EXECUTE request. It is subsequently cached.
OS_PTR_SIZE	INTEGER		Indicates the 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.
LANG_LEVEL	CHAR(1)	Yes	LANGLEVEL value used during BIND: 0 = SAA1 1 = MIA 2 = SQL92E
FUNC_PATH	VARCHAR(254)		The SQL path used by the last BIND command for this package. This is used as the default path for REBIND. SYSIBM for pre-Version 2 packages.
QUERYOPT	INTEGER		Optimization class under which this package was bound. Used for REBIND. The classes are: 0, 1, 3, 5 and 9.
EXPLAIN_LEVEL	CHAR(1)		Indicates whether Explain was requested using the EXPLAIN or EXPLSNAP bind option. P = Plan Selection level Blank if ’No’ Explain requested

EXPLAIN_MODE	CHAR(1)		Value of EXPLAIN bind option:
			Y = Yes (static)
			N = No
			A = All (static and dynamic)

EXPLAIN_SNAPSHOT	CHAR(1)		Value of EXPLSNAP bind option:
			Y = Yes (static)
			N = No
			A = All (static and dynamic)

SQLWARN	CHAR(1)		Are positive SQLCODEs resulting from dynamic SQL statements returned to the application? Y = Yes N = No, they are suppressed.

Table 85. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
SQLMATHWARN	CHAR(1)		Value of the database configuration parameter DFT_SQLMATHWARN at the time of bind. Are arithmetic errors and retrieval conversion errors in static SQL statements handled as nulls with a warning? Y = Yes N = No, they are suppressed.
EXPLICIT_BIND_TIME	TIMESTAMP		The time at which this package was last explicitly bound or rebound. When the package is implicitly rebound, no function instance will be selected that was created later than this time.
LAST_BIND_TIME	TIMESTAMP		Time at which the package last explicitly or implicitly bound or rebound.
CODEPAGE	SMALLINT		Application code page at bind time (-1 if not known).
DEGREE	CHAR(5)		Indicates the limit on intra-partition parallelism (as a bind option) when package was bound. 1 = No intra-partition parallelism. 2 - 32 767 = Degree of intra-partition parallelism. ANY = Degree was determined by the database manager.
MULTINODE_PLANS	CHAR(1)		Y = Package was bound in a multiple partition environment. N =Package was bound in a single partition environment.
INTRA_PARALLEL	CHAR(1)		Indicates the use of intra-partition parallelism by static SQL statements within the package. Y = one or more static SQL statement in package uses intra-partition parallelism. N = no static SQL statement in package uses intra-partition parallelism. F = one or more static SQL statement in package can use intra-partition parallelism; this parallelism has been disabled for use on a system that is not configured for intra-partition parallelism.
VALIDATE	CHAR(1)		B = All checking must be performed during BIND R = Reserved

Table 85. SYSCAT.PACKAGES Catalog View (continued)

Column Name	Data Type	Nullable	Description
DYNAMICRULES	CHAR(1)		B = BIND. Dynamic SQL statements are executed with bind behavior. D = DEFINEBIND. 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 = DEFINERUN. 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)		Indicates SQLERROR option on the most recent subcommand that bound or rebound the package.
			C = Reserved
			N = No package

REFRESHAGE	DECIMAL (20,6)		Timestamp duration indicating the maximum length of time between when a REFRESH TABLE statement is run for a materialized query table and when the materialized query table is used in place of a base table.
TRANSFORMGROUP	CHAR(1024)	Yes	String containing the transform group bind option.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

SYSCAT.PARTITIONMAPS

Contains a row for each partitioning map that is used to distribute table rows among the partitions in a database partition group, based on hashing the table's partitioning key.

Table 86. SYSCAT.PARTITIONMAPS Catalog View

Column Name	Data Type	Nullable	Description
PMAP_ID	SMALLINT		Identifier of the partitioning map.
PARTITIONMAP	LONG VARCHAR FOR BIT DATA		The actual partitioning map, a vector of 4 096 two-byte integers for a multiple 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

This catalog view contains information about authorizations to query data sources in pass-through sessions. A constraint on the base table requires that the values in SERVER correspond to the values in the SERVER column of SYSCAT.SERVERS. None of the fields in SYSCAT.PASSTHRUAUTH are nullable.

Table 87. Columns in SYSCAT.PASSTHRUAUTHCatalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privilege.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privilege.
GRANTEETYPE	CHAR(1)		A letter that specifies the type of grantee: U = Grantee is an individual user. G = Grantee is a group.
SERVERNAME	VARCHAR(128)		Name of the data source that the user or group is being granted authorization to.

SYSCAT.PREDICATESPECS

Each row represents a predicate specification.

Table 88. SYSCAT.PREDICATESPECS Catalog View

Column Name	Data Type	Nullable	Description
FUNCSCHEMA	VARCHAR(128)		Qualified name of function.
FUNCNAME	VARCHAR(128)		
SPECIFICNAME	VARCHAR(128)		The name of the function instance.
FUNCID	INTEGER		Function ID.
SPECID	SMALLINT		ID of this predicate specification.
CONTEXTOP	CHAR(8)		Comparison operator is one of the built-in relational operators (=, <, >=, and so on).
CONTEXTEXP	CLOB(32K)		Constant, or an SQL expression.
FILTERTEXT	CLOB(32K)	Yes	Text of data filter expression.

SYSCAT.PROCOPTIONS

Each row contains procedure specific option values.

Table 89. SYSCAT.PROCOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
PROCSCHEMA	VARCHAR(128)		Qualifier for the stored procedure name or nickname.
PROCNAME	VARCHAR(128)		Name or nickname of the stored procedure.
OPTION	VARCHAR(128)		Name of the stored procedure option.
SETTING	VARCHAR(255)		Value of the stored procedure option.

SYSCAT.PROCPARMOPTIONS

Each row contains procedure parameter specific option values.

Table 90. SYSCAT.PROCPARMOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
PROCSCHEMA	VARCHAR(128)		Qualified procedure name or nickname.
PROCNAME	VARCHAR(128)		
ORDINAL	SMALLINT		The parameter's numerical position within the procedure signature.
OPTION	VARCHAR(128)		Name of the stored procedure parameter option.
SETTING	VARCHAR(255)		Value of the stored procedure parameter option.

SYSCAT.REFERENCES

Contains a row for each defined referential constraint.

Table 91. SYSCAT.REFERENCES Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint.
TABSCHEMA	VARCHAR(128)		Qualified name of the table.
TABNAME	VARCHAR(128)		
DEFINER	VARCHAR(128)		User who created the constraint.
REFKEYNAME	VARCHAR(18)		Name of parent key.
REFTABSCHEMA	VARCHAR(128)		Qualified name of the parent table.
REFTABNAME	VARCHAR(128)		
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		The timestamp when the referential constraint was defined.
FK_COLNAMES	VARCHAR (640)		List of foreign key column names. Warning: This column will be removed in the future. Use SYSCAT.KEYCOLUSE for this information.
PK_COLNAMES	VARCHAR (640)		List of parent key column names. Warning: This column will be removed in the future. Use SYSCAT.KEYCOLUSE for this information.

Note: The SYSCAT.REFERENCES view is based on the SYSIBM.SYSRELS table from Version 1.

Related reference:

- “SYSCAT.KEYCOLUSE” on page 574

SYSCAT.REVTYPEMAPPINGS

Each row contains reverse data type mappings (mappings from data types defined locally to data source data types). No data in this version. Defined for possible future use with data type mappings.

Table 92. SYSCAT.REVTYPEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE_MAPPING	VARCHAR(18)		Name of the reverse type mapping (may be system-generated).
TYPESCHEMA	VARCHAR(128)	Yes	Schema name of the type. Null for system built-in types.
TYPENAME	VARCHAR(18)		Name of the local type in a reverse type mapping.
TYPEID	SMALLINT		Type identifier.
SOURCETYPEID	SMALLINT		Source type identifier.
DEFINER	VARCHAR(128)		Authorization ID under which this type mapping was created.
LOWER_LEN	INTEGER	Yes	Lower bound of the length/precision of the local type.
UPPER_LEN	INTEGER	Yes	Upper bound of the length/precision of the local type. If null then the system determines the best length/precision attribute.
LOWER_SCALE	SMALLINT	Yes	Lower bound of the scale for local decimal data types.
UPPER_SCALE	SMALLINT	Yes	Upper bound of the scale for local decimal data types. If null, then the system determines the best scale attribute.
S_OPR_P	CHAR(2)	Yes	Relationship between local scale and local precision. Basic comparison operators can be used. A null indicates that no specific relationship is required.
BIT_DATA	CHAR(1)	Yes	Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type or that the system determines the bit data attribute.
WRAPNAME	VARCHAR(128)	Yes	Mapping applies to this data access protocol.
SERVERNAME	VARCHAR(128)	Yes	Name of the data source.
SERVERTYPE	VARCHAR(30)	Yes	Mapping applies to this type of data source.
SERVERVERSION	VARCHAR(18)	Yes	Mapping applies to this version of SERVERTYPE.
REMOTE_TYPESCHEMA	VARCHAR(128)	Yes	Schema name of the remote type.
REMOTE_TYPENAME	VARCHAR(128)		Name of the data type as defined on the data source(s).
REMOTE_META_TYPE	CHAR(1)	Yes	S = Remote type is a system built-in type. T = Remote type is a distinct type.
REMOTE_LENGTH	INTEGER	Yes	Maximum number of digits for remote decimal type, and maximum number of characters for remote character type. Otherwise null.

Table 92. SYSCAT.REVTYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_SCALE	SMALLINT	Yes	Maximum number of digits allowed to the right of the decimal point (for remote decimal types). Otherwise null.
REMOTE_BIT_DATA	CHAR(1)	Yes	Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type or that the system determines the bit data attribute.
USER_DEFINED	CHAR(1)		Defined by user.
CREATE_TIME	TIMESTAMP		Time when this mapping was created.
REMARKS	VARCHAR(254)	Yes	User supplied comments, or null.

SYSCAT.ROUTINEAUTH

Contains one or more rows for each user or group who is granted EXECUTE privilege on a particular routine in the database.

Table 93. SYSCAT.ROUTINEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privilege or SYSIBM.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privilege.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
SCHEMA	VARCHAR(128)		Qualifier of the routine.
SPECIFICNAME	VARCHAR(128)	Yes	Specific name of the routine. If SPECIFICNAME is null and ROUTINETYPE is not M, the privilege applies to all routines in the schema of the type specified in ROUTINETYPE. If SPECIFICNAME is null and ROUTINETYPE is M, the privilege applies to all methods in the schema of subject type TYPENAME.
TYPESCHEMA	VARCHAR(128)	Yes	Qualifier of the type name for the method. If ROUTINETYPE is not M, TYPESCHEMA is null.
TYPENAME	VARCHAR(18)	Yes	Type name for the method. If ROUTINETYPE is not M, TYPENAME is null. If TYPENAME is null and ROUTINETYPE is M, the privilege applies to subject types in the schema TYPESCHEMA.

ROUTINETYPE	CHAR(1)		Type of routine:
			F = Function
			M = Method
			P = Procedure

EXECUTEAUTH	CHAR(1)		Indicates whether grantee holds EXECUTE privilege on the function or method: Y = Privilege is held. G = Privilege is held and grantable. N = Privilege is not held.
GRANT_TIME	TIMESTAMP		Time at which the EXECUTE privilege is granted.

SYSCAT.ROUTINEDEP

Each row represents a dependency of a routine on some other object. (This catalog view supercedes SYSCAT.FUNCDEP. The other view exists, but will remain as it was in DB2 Version 7.1.)

Table 94. SYSCAT.FUNCDEP Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR(128)		Qualified name of the routine that has dependencies on another object.
ROUTINENAME	VARCHAR(128)		

BTYPE	CHAR(1)		Type of object on which the routine depends.
			A = Alias
			F = Routine instance
			O = Privilege dependency on all subtables or subviews in a table or view hierarchy
			R = Structured type
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view
			X = Index extension

BSCHEMA	VARCHAR(128)		Qualified name of the object on which the function or method depends (if BTYPE = F, this is the specific name of a routine).
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE = O, S, T, U, V or W, it encodes the privileges on the table or view that are required by the dependent routine. Otherwise null.

SYSCAT.ROUTINEPARMS

Contains a row for every parameter or result of a routine defined in SYSCAT.ROUTINES. (This catalog view supercedes SYSCAT.FUNCPARMS and SYSCAT.PROCPARMS. The other views exist, but will remain as they were in DB2 Version 7.1.)

Table 95. SYSCAT.ROUTINEPARMS Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR(128)		Qualified routine name.
ROUTINENAME	VARCHAR(128)		
SPECIFICNAME	VARCHAR(128)		The name of the routine instance (may be system-generated).
PARMNAME	VARCHAR(128)	Yes	Name of parameter or result column, or null 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, the parameter's numerical position within the routine signature. If ROWTYPE = R, and the routine is a table function, the column's numerical position within the result table. Otherwise 0.
TYPESCHEMA	VARCHAR(128)		Qualified name of data type of parameter or result.
TYPENAME	VARCHAR(128)		
LOCATOR	CHAR(1)		Y = Parameter or result is passed in the form of a locator. N = Parameter or result is not passed in the form of a locator.
LENGTH	INTEGER		Length of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
SCALE	SMALLINT		Scale of parameter or result. 0 if parameter or result is a distinct type. See Note 1.
CODEPAGE	SMALLINT		Code page of parameter or result. 0 denotes either not applicable, or a parameter or result for character data declared with the FOR BIT DATA attribute.
CAST_FUNCSCHEMA	VARCHAR(128)	Yes	Qualified name of the function used to cast an argument or a result. Applies to sourced and external functions; null otherwise.
CAST_FUNCSPECIFIC	VARCHAR(128)	Yes	
TARGET_TYPESCHEMA	VARCHAR(128)	Yes	Qualified name of the target type, if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE.
TARGET_TYPENAME	VARCHAR(128)	Yes	
SCOPE_TABSCHEMA	VARCHAR(128)	Yes	Qualified name of the scope (target table), if the type of the parameter or result is REFERENCE. Null value if the type of the parameter or result is not REFERENCE, or the scope is not defined.
SCOPE_TABNAME	VARCHAR(128)	Yes	

Table 95. SYSCAT.ROUTINEPARMS Catalog View (continued)

Column Name	Data Type	Nullable	Description
TRANSFORMGRPNAME	VARCHAR(128)	Yes	Name of transform group for a structured type parameter or result.
REMARKS	VARCHAR(254)	Yes	Parameter remarks.

Notes:

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

Contains a row for each user-defined function (scalar, table, or source), system-generated method, user-defined method, or procedure. Does not include built-in functions. (This catalog view supercedes SYSCAT.FUNCTIONS and SYSCAT.PROCEDURES. The other views exist, but will remain as they were in DB2 Version 7.1.)

Table 96. SYSCAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Description
ROUTINESCHEMA	VARCHAR(128)		Qualified routine name.
| ROUTINENAME	VARCHAR(128)		

ROUTINETYPE	CHAR(1)		F = Function
			M = Method
			P = Procedure.

DEFINER	VARCHAR(128)		Authorization ID of routine definer.
| SPECIFICNAME	VARCHAR(128)		The name of the routine instance (may be system-generated).
ROUTINEID	INTEGER		Internally-assigned routine ID.
RETURN_TYPESCHEMA	VARCHAR(128)	Yes	Qualified name of the return type for a scalar function or method.
RETURN_TYPENAME	VARCHAR(128)	Yes	

ORIGIN	CHAR(1)		B = Built-in
			E = User-defined, external
			M = Template
			Q = SQL-bodied
			U = User-defined, based on a source
			S = System-generated
			T = System-generated transform

FUNCTIONTYPE	CHAR(1)		C = Column function
			R = Row function
			S = Scalar function or method
			T = Table function
			Blank = Procedure

PARM_COUNT	SMALLINT		Number of parameters.
LANGUAGE	CHAR(8)		Implementation language of routine body. Possible values are C, COBOL, JAVA, OLE, OLEDB, or SQL. Blank if ORIGIN is not E or Q.
SOURCESCHEMA	VARCHAR(128)	Yes	If ORIGIN = U and the routine is a user-defined function, contains the qualified name of the source function. If ORIGIN = U and the source function is built-in, SOURCESCHEMA is 'SYSIBM' and SOURCESPECIFIC is 'N/A for built-in'. Null if ORIGIN is not U.
| SOURCESPECIFIC	VARCHAR(128)	Yes	

Table 96. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description

DETERMINISTIC	CHAR(1)		Y = Deterministic (results are consistent)
			N = Non-deterministic (results may differ)
			Blank if ORIGIN is not E or Q.

EXTERNAL_ACTION	CHAR(1)		E = Function has external side-effects (number of invocations is important) N = No side-effects Blank if ORIGIN is not E or Q.
NULLCALL	CHAR(1)		Y = CALLED ON NULL INPUT N = RETURNS NULL ON NULL INPUT (result is implicitly null if operand(s) are null). Blank if ORIGIN is not E or Q.
CAST_FUNCTION	CHAR(1)		Y = This is a cast function N = This is not a cast function
ASSIGN_FUNCTION	CHAR(1)		Y = Implicit assignment function N = Not an assignment function
SCRATCHPAD	CHAR(1)		Y = This routine has a scratch pad N = This routine does not have a scratch pad Blank if ORIGIN is not E or ROUTINETYPE is P.
SCRATCHPAD_LENGTH	SMALLINT		*n* = Length of the scratch pad in bytes 0 = SCRATCHPAD is N -1 = LANGUAGE is OLEDB
FINALCALL	CHAR(1)		Y = Final call is made to this function at runtime end-of-statement. N = No final call is made. Blank if ORIGIN is not E.
PARALLEL	CHAR(1)		Y = Function can be executed in parallel. N = Function cannot be executed in parallel. Blank if ORIGIN is not E.
PARAMETER_STYLE	CHAR(8)		Indicates the parameter style declared when the routine was created. Values: DB2SQL SQL DB2GENRL GENERAL JAVA DB2DARI GNRLNULL Blank if ORIGIN is not E.

Table 96. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description

FENCED	CHAR(1)		Y = Fenced
			N = Not fenced
			Blank if ORIGIN is not E.

SQL_DATA_ACCESS	CHAR(1)		C = CONTAINS SQL: only SQL that does not read or modify SQL data is allowed.
			M = MODIFIES SQL DATA: all SQL allowed in routines is allowed.
			N = NO SQL: SQL is not allowed.
			R = READS SQL DATA: only SQL that reads SQL data is allowed.

DBINFO	CHAR(1)		Y = DBINFO is passed.
			N = DBINFO is not passed.

PROGRAMTYPE	CHAR(1)		M = Main
			S = Subroutine

COMMIT_ON_RETURN	CHAR(1)		N = Changes are not committed after the procedure completes. Blank if ROUTINETYPE is not P.
RESULT_SETS	SMALLINT		Estimated upper limit of returned result sets.
SPEC_REG	CHAR(1)		I = INHERIT SPECIAL REGISTERS: special registers start with their values from the invoking statement. Blank if ORIGIN is not E or Q.
| FEDERATED	CHAR(1)		Not used.
THREADSAFE	CHAR(1)		Y = Routine can run in the same process as other routines. N = Routine must be run in a separate process from other routines. Blank if ORIGIN is not E.

VALID	CHAR(1)		Y = SQL procedure is valid.
			N = SQL procedure is invalid.
			X = SQL procedure is inoperative because some object it requires has been dropped. The SQL procedure must be explicitly dropped and recreated.
			Blank if ORIGIN is not Q.

METHODIMPLEMENTED	CHAR(1)		Y = Method is implemented. N = Method specification without an implementation. Blank if ROUTINETYPE is not M.

METHODEFFECT	CHAR(2)		MU = Mutator method
			OB = Observer method
			CN = Constructor method
			Blanks if FUNCTIONTYPE is not T.

Table 96. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
TYPE_PRESERVING	CHAR(1)		Y = Return type is governed by a "type-preserving" parameter. All system-generated mutator methods are type-preserving. N = Return type is the declared return type of the method. Blank if ROUTINETYPE is not M.
WITH_FUNC_ACCESS	CHAR(1)		Y = This method can be invoked by using functional notation. N = This method cannot be invoked by using functional notation. Blank if ROUTINETYPE is not M.
OVERRIDEN_METHODID	INTEGER	Yes	Reserved for future use.
SUBJECT_TYPESCHEMA	VARCHAR(128)	Yes	Subject type for method.
| SUBJECT_TYPENAME	VARCHAR(128)	Yes	
CLASS	VARCHAR(128)	Yes	If LANGUAGE = JAVA, identifies the class that implements this routine. Null otherwise.
JAR_ID	VARCHAR(128)	Yes	If LANGUAGE = JAVA, identifies the jar file that implements this routine. Null otherwise.
JARSCHEMA	VARCHAR(128)	Yes	If LANGUAGE = JAVA, identifies the schema of the jar file that implements this routine. Null otherwise.
| JAR_SIGNATURE	VARCHAR(1024)	Yes	If LANGUAGE = JAVA, identifies the signature of the Java method that implements this routine. Null otherwise.
CREATE_TIME	TIMESTAMP		Timestamp of routine creation. Set to 0 for Version 1 functions.
ALTER_TIME	TIMESTAMP		Timestamp of most recent routine alteration. If the routine has not been altered, set to CREATE_TIME.
FUNC_PATH	VARCHAR(254)	Yes	SQL path at the time the routine was defined.
QUALIFIER	VARCHAR(128)		Value of default schema at object definition time.
IOS_PER_INVOC	DOUBLE		Estimated number of I/Os per invocation; -1 if not known (0 default).
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; -1 if not known (450 default).
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/Os per input argument byte; -1 if not known (0 default).
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; -1 if not known (0 default).
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the routine will actually read; -1 if not known (100 default).
INITIAL_IOS	DOUBLE		Estimated number of I/Os performed the first/last time the routine is invoked; -1 if not known (0 default).

Table 96. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first/last time the routine is invoked; -1 if not known (0 default).
CARDINALITY	BIGINT		The predicted cardinality of a table function; -1 if not known, or if the routine is not a table function.
SELECTIVITY	DOUBLE		Used for user-defined predicates; -1 if there are no user-defined predicates. See Note 1.
RESULT_COLS	SMALLINT		For a table function (ROUTINETYPE = F and TYPE = T) contains the number of columns in the result table. For other functions and methods (ROUTINETYPE = F or M), contains 1. For procedures (ROUTINETYPE = P), contains 0.
IMPLEMENTATION	VARCHAR(254)	Yes	If ORIGIN = E, identifies the path/module/function that implements this function. If ORIGIN = U and the source function is built-in, this column contains the name and signature of the source function. Null otherwise.
LIB_ID	INTEGER	Yes	Reserved for future use.
TEXT_BODY_OFFSET	INTEGER		If LANGUAGE = SQL, the offset to the start of the SQL procedure body in the full text of the CREATE statement; -1 if LANGUAGE is not SQL.
TEXT	CLOB(2M)	Yes	If LANGUAGE = SQL, the text of the CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement.
NEWSAVEPOINTLEVEL	CHAR(1)		Indicates whether the routine initiates a new savepoint level when it is invoked. Y = A new savepoint level is initiated when the routine is invoked. N = A new savepoint level is not initiated when the routine is invoked. The routine uses the existing savepoint level. Blank if ORIGIN is not E or Q.
DEBUG_MODE	CHAR(3)		OFF = Debugging is off for this routine. ON = Debugging is on for this routine.
TRACE_LEVEL	CHAR(1)		Reserved for future use.
DIAGNOSTIC_LEVEL	CHAR(1)		Reserved for future use.
CHECKOUT_USERID	VARCHAR(128)	Yes	User ID of the user who performed a checkout of the object. Null if not checked out.
PRECOMPILE_OPTIONS	VARCHAR(1024)	Yes	Precompile options specified for the routine.
COMPILE_OPTIONS	VARCHAR(1024)	Yes	Compile options specified for the routine.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

Table 96. SYSCAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description

Notes:

1. This column will be set to -1 during migration 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.

SYSCAT.SCHEMAAUTH

Contains one or more rows for each user or group who is granted a privilege on a particular schema in the database. All schema privileges for a single schema granted by a specific grantor to a specific grantee appear in a single row.

Table 97. SYSCAT.SCHEMAAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
SCHEMANAME	VARCHAR(128)		Name of the schema.
ALTERINAUTH	CHAR(1)		Indicates whether grantee holds ALTERIN privilege on the schema: Y = Privilege is held. G = Privilege is held and grantable. N = Privilege is not held.
CREATEINAUTH	CHAR(1)		Indicates whether grantee holds CREATEIN privilege on the schema: Y = Privilege is held. G = Privilege is held and grantable. N = Privilege is not held.
DROPINAUTH	CHAR(1)		Indicates whether grantee holds DROPIN privilege on the schema: Y = Privilege is held. G = Privilege is held and grantable. N = Privilege is not held.

SYSCAT.SCHEMATA

Contains a row for each schema.

Table 98. SYSCAT.SCHEMATA Catalog View

Column Name	Data Type	Nullable	Description
SCHEMANAME	VARCHAR(128)		Name of the schema.
OWNER	VARCHAR(128)		Authorization id of the schema. The value for implicitly created schemas is SYSIBM.
DEFINER	VARCHAR(128)		User who created the schema.
CREATE_TIME	TIMESTAMP		Timestamp indicating when the object was created.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

SYSCAT.SEQUENCEAUTH

Contains a row for each authorization ID that can be used to use or to alter a sequence.

Table 99. SYSCAT.SEQUENCEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		SYSIBM or authorization ID that granted the privilege.
GRANTEE	VARCHAR(128)		Authorization ID that holds the privilege.
GRANTEETYPE	CHAR(1)		Type of authorization ID that holds the privilege. U = grantee is an individual user
SEQSCHEMA	VARCHAR(128)		Qualified name of the sequence.
SEQNAME	VARCHAR(128)		
USAGEAUTH	CHAR(1)		Y = privilege is held N = privilege is not held G = privilege is held and is grantable
ALTERAUTH	CHAR(1)		Y = privilege is held N = privilege is not held G = privilege is held and is grantable

SYSCAT.SEQUENCES

| Contains a row for each sequence or identity column defined in the database.

Table 100. Columns in SYSCAT.SEQUENCES Catalog View

Column Name	Data Type	Nullable	Description
SEQSCHEMA	VARCHAR(128)		Qualified name of the sequence (generated by DB2 for an identity column).
SEQNAME	VARCHAR(128)		
DEFINER	VARCHAR(128)		Definer of the sequence.
OWNER	VARCHAR(128)		Owner of the sequence.
SEQID	INTEGER		Internal ID of the sequence.

SEQTYPE	CHAR(1)		Sequence type
			S = Regular sequence
			I = Identity sequence

INCREMENT	DECIMAL(31,0)		Increment value.
START	DECIMAL(31,0)		Starting value.
MAXVALUE	DECIMAL(31,0)		Maximal value.
MINVALUE	DECIMAL(31,0)		Minimum value.
CYCLE	CHAR(1)		Whether cycling will occur when a boundary is reached: Y - cycling will occur N - cycling will not occur
CACHE	INTEGER		Number of sequence values to preallocate in memory for faster access. 0 indicates that values are not preallocated.
ORDER	CHAR(1)		Whether or not the sequence numbers must be generated in order of request: Y - sequence numbers must be generated in order of request N - sequence numbers are not required to be generated in order of request
DATATYPEID	INTEGER		For built-in types, the internal ID of the built-in type. For distinct types, the internal ID of the distinct type.
SOURCETYPEID	INTEGER		For a built-in type, this has a value of 0. For a distinct type, this is the internal ID of the built-in type that is the source type for the distinct type.
CREATE_TIME	TIMESTAMP		Time when the sequence was created.
ALTER_TIME	TIMESTAMP		Time when the last ALTER SEQUENCE statement was executed for this sequence.
PRECISION	SMALLINT		The precision of the data type of the sequence. Values are: 5 for a SMALLINT, 10 for INTEGER, and 19 for BIGINT. For DECIMAL, it is the precision of the specified DECIMAL data type.
ORIGIN	CHAR(1)		Sequence Origin U - User generated sequence S - System generated sequence
REMARKS	VARCHAR(254)	Yes	User supplied comments, or null.

SYSCAT.SERVEROPTIONS

Each row contains configuration options at the server level.

Table 101. Columns in SYSCAT.SERVEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR(128)	Yes	Wrapper name.
SERVERNAME	VARCHAR(128)	Yes	Name of the server.
SERVERTYPE	VARCHAR(30)	Yes	Server type.
SERVERVERSION	VARCHAR(18)	Yes	Server version.
CREATE_TIME	TIMESTAMP		Time when entry is created.
OPTION	VARCHAR(128)		Name of the server option.
SETTING	VARCHAR(2048)		Value of the server option.
SERVEROPTIONKEY	VARCHAR(18)		Uniquely identifies a row.
REMARKS	VARCHAR(254)	Yes	User supplied comments, or null.

SYSCAT.SERVERS

Each row represents a data source. Catalog entries are not necessary for tables that are stored in the same instance that contains this catalog table.

Table 102. Columns in SYSCAT.SERVERS Catalog View

Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR(128)		Wrapper name.
SERVERNAME	VARCHAR(128)		Name of data source as it is known to the system.
SERVERTYPE	VARCHAR(30)	Yes	Type of data source (always uppercase).
SERVERVERSION	VARCHAR(18)	Yes	Version of data source.
REMARKS	VARCHAR(254)	Yes	User supplied comments, or null.

SYSCAT.STATEMENTS

Contains one or more rows for each SQL statement in each package in the database.

Table 103. SYSCAT.STATEMENTS Catalog View

Column Name	Data Type	Nullable	Description
PKGSCHEMA	VARCHAR(128)		Name of the package.
PKGNAME	CHAR(8)		
UNIQUEID	CHAR(8)		Internal date and time information indicating when the package was first created. Useful for identifying a specific package when multiple packages having the same name exist.
PKGVERSION	VARCHAR(64)		Version identifier 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	SMALLINT		Always 1.
TEXT	CLOB (64K)		Text of the SQL statement.

SYSCAT.TABAUTH

Contains one or more rows for each user or group who is granted a privilege on a particular table or view in the database. All the table privileges for a single table or view granted by a specific grantor to a specific grantee appear in a single row.

Table 104. SYSCAT.TABAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	VARCHAR(128)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	VARCHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
TABSCHEMA	VARCHAR(128)		Qualified name of the table or view.
TABNAME	VARCHAR(128)		
CONTROLAUTH	CHAR(1)		Indicates whether grantee holds CONTROL privilege on the table or view: Y = Privilege is held. N = Privilege is not held.
ALTERAUTH	CHAR(1)		Indicates whether grantee holds ALTER privilege on the table: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
DELETEAUTH	CHAR(1)		Indicates whether grantee holds DELETE privilege on the table or view: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
INDEXAUTH	CHAR(1)		Indicates whether grantee holds INDEX privilege on the table: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
INSERTAUTH	CHAR(1)		Indicates whether grantee holds INSERT privilege on the table or view: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
SELECTAUTH	CHAR(1)		Indicates whether grantee holds SELECT privilege on the table or view: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.

Table 104. SYSCAT.TABAUTH Catalog View (continued)

Column Name	Data Type	Nullable	Description
REFAUTH	CHAR(1)		Indicates whether grantee holds REFERENCE privilege on the table or view: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.
UPDATEAUTH	CHAR(1)		Indicates whether grantee holds UPDATE privilege on the table or view: Y = Privilege is held. N = Privilege is not held. G = Privilege is held and grantable.

SYSCAT.TABCONST

Each row represents a table constraint of type CHECK, UNIQUE, PRIMARY KEY, or FOREIGN KEY.

Table 105. SYSCAT.TABCONST Catalog View

Column Name	Data Type	Nullable	Description
CONSTNAME	VARCHAR(18)		Name of the constraint (unique within a table).
TABSCHEMA	VARCHAR(128)		Qualified name of the table to which this constraint applies.
TABNAME	VARCHAR(128)		
DEFINER	VARCHAR(128)		Authorization ID under which the constraint was defined.

TYPE	CHAR(1)		Indicates the constraint type:
			F = Foreign key
			I = Functional dependency
			K = Check
			P = Primary key
			U = Unique

REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.
ENFORCED	CHAR(1)		Y = Enforce constraint N = Do not enforce constraint
CHECKEXISTINGDATA	CHAR(1)		D = Defer checking of existing data I = Immediately check existing data N = Never check existing data
ENABLEQUERYOPT	CHAR(1)		Y = Query optimization is enabled N = Query optimization is disabled

SYSCAT.TABDEP

Contains a row for every dependency of a view or a materialized query table on some other object. Also encodes how privileges on this view depend on privileges on underlying tables and views. (The VIEWDEP catalog view is still available, but only at the Version 7.1 level.)

Table 106. SYSCAT.TABDEP Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Name of the view or materialized query table with dependencies on a base table.
TABNAME	VARCHAR(128)		

DTYPE	CHAR(1)		S = Materialized query table
			V = View (untyped)
			W = Typed view

DEFINER	VARCHAR(128)	Yes	Authorization ID of the creator of the view.

BTYPE	CHAR(1)		Type of object BNAME:
			A = Alias
			F = Function instance
			N = Nickname
			O = Privilege dependency on all subtables or subviews in a table or view hierarchy
			I = Index if recording dependency on a base table
			R = Structured type
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view

BSCHEMA	VARCHAR(128)		Qualified name of the object on which the view depends.
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE = N, O, S, T, U, V, W, encodes the privileges on the underlying table or view on which this table depends. Otherwise null.

SYSCAT.TABLES

Contains one row for each table, view, nickname or alias that is created. All of the catalog tables and views have entries in the SYSCAT.TABLES catalog view.

Table 107. SYSCAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of the table, view, nickname, or alias.
TABNAME	VARCHAR(128)		
DEFINER	VARCHAR(128)		User who created the table, view, nickname or alias.

TYPE	CHAR(1)		The type of object:
			A = Alias
			H = Hierarchy table
			N = Nickname
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view

STATUS	CHAR(1)		The check pending status of the object: N = Normal table, view, alias or nickname C = Check pending on table or nickname X = Inoperative view or nickname
DROPRULE	CHAR(1)		N = No rule R = Restrict rule applies on drop
BASE_TABSCHEMA	VARCHAR(128)	Yes	If TYPE = A, these columns identify the table, view, alias, or nickname that is referenced by this alias; otherwise they are null.
BASE_TABNAME	VARCHAR(128)	Yes	
ROWTYPESCHEMA	VARCHAR(128)	Yes	Contains the qualified name of the rowtype of this table, where applicable. Null otherwise.
ROWTYPENAME	VARCHAR(18)		
CREATE_TIME	TIMESTAMP		The timestamp indicating when the object was created.
STATS_TIME	TIMESTAMP	Yes	Last time when any change was made to recorded statistics for this table. Null if no statistics available.
COLCOUNT	SMALLINT		Number of columns in the table.
TABLEID	SMALLINT		Internal table identifier.
TBSPACEID	SMALLINT		Internal identifier of primary table space for this table.
CARD	BIGINT		Total number of rows in the table. For tables in a table hierarchy, the number of rows at the given level of the hierarchy; −1 if statistics are not gathered, or the row describes a view or alias; −2 for hierarchy tables (H-tables).

Table 107. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; −1 if statistics are not gathered, or the row describes a view or alias; −2 for subtables or H-tables.
FPAGES	INTEGER		Total number of pages; −1 if statistics are not gathered, or the row describes a view or alias; −2 for subtables or H-tables.
OVERFLOW	INTEGER		Total number of overflow records in the table; −1 if statistics are not gathered, or the row describes a view or alias; −2 for subtables or H-tables.
TBSPACE	VARCHAR(18)	Yes	Name of primary table space for the table. If no other table space is specified, all parts of the table are stored in this table space. Null for aliases and views.
INDEX_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all indexes created on this table. Null for aliases and views, or if the INDEX IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
LONG_TBSPACE	VARCHAR(18)	Yes	Name of table space that holds all long data (LONG or LOB column types) for this table. Null for aliases and views, or if the LONG IN clause was omitted or specified with the same value as the IN clause of the CREATE TABLE statement.
PARENTS	SMALLINT	Yes	Number of parent tables of this table (the number of referential constraints in which this table is a dependent).
CHILDREN	SMALLINT	Yes	Number of dependent tables of this table (the number of referential constraints in which this table is a parent).
SELFREFS	SMALLINT	Yes	Number of self-referencing referential constraints for this table (the number of referential constraints in which this table is both a parent and a dependent).
KEYCOLUMNS	SMALLINT	Yes	Number of columns in the primary key of the table.
KEYINDEXID	SMALLINT	Yes	Index ID of the primary index. This field is null or 0 if there is no primary key.
KEYUNIQUE	SMALLINT		Number of unique constraints (other than primary key) defined on this table.
CHECKCOUNT	SMALLINT		Number of check constraints defined on this table.
DATACAPTURE	CHAR(1)		Y = Table participates in data replication N = Does not participate L = Table participates in data replication, including replication of LONG VARCHAR and LONG VARGRAPHIC columns

Table 107. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description

CONST_CHECKED	CHAR(32)		Byte 1 represents foreign key constraints. Byte 2 represents check constraints. Byte 5 represents materialized query table. Byte 6 represents generated columns. Byte 7 represents staging table. Other bytes are reserved. Encodes constraint information on checking. Values:
			Y = Checked by system
			U = Checked by user
			N = Not checked (pending)
			W = Was in a 'U' state when the table was placed in check pending (pending)
			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.

PMAP_ID	SMALLINT	Yes	Identifier of the partitioning map used by this table. Null for aliases and views.
PARTITION_MODE	CHAR(1)		Mode used for tables in a partitioned database. H = Hash on the partitioning key R = Table replicated across database partitions Blank for aliases, views and tables in single partition database partition groups with no partitioning key defined. Also blank for nicknames.
LOG_ATTRIBUTE	CHAR(1)		0 = Default logging 1 = Table created not logged initially
PCTFREE	SMALLINT		Percentage of each page to be reserved for future inserts. Can be changed by ALTER TABLE.
APPEND_MODE	CHAR(1)		Controls how rows are inserted on pages: N = New rows are inserted into existing spaces if available Y = New rows are appended at end of data Initial value is N.

REFRESH	CHAR(1)		Refresh mode:
			D = Deferred
			I = Immediate
			O = Once
			Blank if not a materialized query table

REFRESH_TIME	TIMESTAMP	Yes	For REFRESH = D or O, timestamp of the REFRESH TABLE statement that last refreshed the data. Otherwise null.

Table 107. SYSCAT.TABLES Catalog View (continued)

Column Name	Data Type	Nullable	Description
LOCKSIZE	CHAR(1)		Indicates preferred lock granularity for tables when accessed by DML statements. Only applies to tables. Possible values are: R = Row T = Table Blank if not applicable Initial value is R.
VOLATILE	CHAR(1)		C = Cardinality of the table is volatile Blank if not applicable
| ROW_FORMAT	CHAR(1)		Not used.
PROPERTY	VARCHAR(32)		Properties for the table. A single blank indicates that the table has no properties.
STATISTICS_PROFILE	CLOB(32K)	Yes	RUNSTATS command used to register a statistical profile of the table.
COMPRESSION	CHAR(1)		V = Value compression is activated, and a row format that supports compression is used N = No compression. A row format that does not support compression is used
ACCESS_MODE	CHAR(1)		Access mode of the object. This access mode is used in conjunction with the STATUS field to represent one of four states. Possible values are: N = No access (corresponds to a status value of C) R = Read-only (corresponds to a status value of C) D = No data movement (corresponds to a status value of N) F = Full access (corresponds to a status value of N)
CLUSTERED	CHAR(1)	Yes	Y = multi-dimensional clustering (MDC) table Null for a non-MDC table
ACTIVE_BLOCKS	INTEGER	Yes	Total number of in-use blocks in an MDC table; –1 if statistics are not gathered.
| MAXFREESPACESEARCH	SMALLINT		Reserved for future use.
REMARKS	VARCHAR(254)	Yes	User-provided comment.

SYSCAT.TABLESPACES

Contains a row for each table space.

Table 108. SYSCAT.TABLESPACES Catalog View

Column Name	Data Type	Nullable	Description
TBSPACE	VARCHAR(18)		Name of the table space.
DEFINER	VARCHAR(128)		Authorization ID of the table space definer.
CREATE_TIME	TIMESTAMP		Creation time of the table space.
TBSPACEID	INTEGER		Internal table space identifier.

TBSPACETYPE	CHAR(1)		The type of table space:
			S = System managed space
			D = Database managed space

DATATYPE	CHAR(1)		The type of data that can be stored: A = All types of permanent data L = Large data - long data or index data T = System temporary tables only U = Declared temporary tables only
EXTENTSIZE	INTEGER		Size of extent, in pages of size PAGESIZE. This many pages are written to one container in the table space before switching to the next container.
PREFETCHSIZE	INTEGER		Number of pages of size PAGESIZE to be read when prefetch is performed; -1 if prefetch size is AUTOMATIC.
OVERHEAD	DOUBLE		Controller overhead and disk seek and latency time, in milliseconds.
TRANSFERRATE	DOUBLE		Time to read one page of size PAGESIZE into the buffer.
PAGESIZE	INTEGER		Size (in bytes) of pages in the table space.
DBPGNAME	VARCHAR(18)		Name of the database partition group for the table space.
BUFFERPOOLID	INTEGER		ID of buffer pool used by this table space; 1 indicates the default buffer pool.
DROP_RECOVERY	CHAR(1)		N = table is not recoverable after a DROP TABLE statement Y = table is recoverable after a DROP TABLE statement
REMARKS	VARCHAR(254)	Yes	User-provided comment.

SYSCAT.TABOPTIONS

Each row contains option associated with a remote table.

Table 109. SYSCAT.TABOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
TABSCHEMA	VARCHAR(128)		Qualified name of table, view, alias or nickname.
TABNAME	VARCHAR(128)		
OPTION	VARCHAR(128)		Name of the table, view, alias or nickname option.
SETTING	VARCHAR(255)		Value.

SYSCAT.TBSPACEAUTH

Contains one row for each user or group who is granted USE privilege on a particular table space in the database.

Table 110. SYSCAT.TBSPACEAUTH Catalog View

Column Name	Data Type	Nullable	Description
GRANTOR	CHAR(128)		Authorization ID of the user who granted the privileges or SYSIBM.
GRANTEE	CHAR(128)		Authorization ID of the user or group who holds the privileges.
GRANTEETYPE	CHAR(1)		U = Grantee is an individual user. G = Grantee is a group.
TBSPACE	VARCHAR(18)		Name of the table space.
USEAUTH	CHAR(1)		Indicates whether grantee holds USE privilege on the table space: G = Privilege is held and grantable. N = Privilege is not held. Y = Privilege is held.

SYSCAT.TRANSFORMS

Contains a row for each transform function type within a user-defined type contained in a named transform group.

Table 111. SYSCAT.TRANSFORMS Catalog View

Column Name	Data Type	Nullable	Description
TYPEID	SMALLINT		Internal type ID as defined in SYSCAT.DATATYPES
TYPESCHEMA	VARCHAR(128)		Qualified name of the given user-defined structured type.
TYPENAME	VARCHAR(128)		
GROUPNAME	VARCHAR(128)		Transform group name.
FUNCID	INTEGER	Yes	Internal routine ID for the associated transform function, as defined in SYSCAT.ROUTINES. Null only for internal system functions.
FUNCSCHEMA	VARCHAR(128)		Qualified name of the associated transform functions.
FUNCNAME	VARCHAR(128)		
SPECIFICNAME	VARCHAR(128)		Function specific (instance) name.
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)		'U' = User defined
MAXLENGTH	INTEGER	Yes	Maximum length (in bytes) of output from the FROM SQL transform. Null for TO SQL transforms.
ORIGIN	CHAR(1)		'O' = Original transform group (user- or system-defined) 'R' = Redefined
REMARKS	VARCHAR(254)	Yes	User-supplied comment or null.

SYSCAT.TRIGDEP

Contains a row for every dependency of a trigger on some other object.

Table 112. SYSCAT.TRIGDEP Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR(128)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		

BTYPE	CHAR(1)		Type of object BNAME:
			A = Alias
			B = Trigger
			F = Function instance
			N = Nickname
			O = Privilege dependency on all subtables or subviews in a table or view hierarchy
			R = Structured type
			S = Materialized query table
			T = Table
			U = Typed table
			V = View
			W = Typed view
			X = Index extension

BSCHEMA	VARCHAR(128)		Qualified name of object depended on by a trigger.
BNAME	VARCHAR(128)		
TABAUTH	SMALLINT	Yes	If BTYPE= O, S, T, U, V or W encodes the privileges on the table or view that are required by this trigger; otherwise null.

SYSCAT.TRIGGERS

Contains one row for each trigger. For table hierarchies, each trigger is recorded only at the level of the hierarchy where it was created.

Table 113. SYSCAT.TRIGGERS Catalog View

Column Name	Data Type	Nullable	Description
TRIGSCHEMA	VARCHAR(128)		Qualified name of the trigger.
TRIGNAME	VARCHAR(18)		
DEFINER	VARCHAR(128)		Authorization ID under which the trigger was defined.
TABSCHEMA	VARCHAR(128)		Qualified name of the table or view to which this trigger applies.
TABNAME	VARCHAR(128)		
TRIGTIME	CHAR(1)		Time when triggered actions are applied to the base table, relative to the event that fired the trigger: A = Trigger applied after event B = Trigger applied before event I = Trigger applied instead of event
TRIGEVENT	CHAR(1)		Event that fires the trigger. I = Insert D = Delete U = Update
GRANULARITY	CHAR(1)		Trigger is executed once per: S = Statement R = Row
VALID	CHAR(1)		Y = Trigger is valid X = Trigger is inoperative; must be re-created.
CREATE_TIME	TIMESTAMP		Time at which the trigger was defined. Used in resolving functions and types.
QUALIFIER	VARCHAR(128)		Contains value of the default schema at the time of object definition.
FUNC_PATH	VARCHAR(254)		Function path at the time the trigger was defined. Used in resolving functions and types.
TEXT	CLOB(64K)		The full text of the CREATE TRIGGER statement, exactly as typed.
REMARKS	VARCHAR(254)	Yes	User-supplied comment, or null.

SYSCAT.TYPEMAPPINGS

Each row contains a user-defined mapping of a remote built-in data type to a local built-in data type.

Table 114. SYSCAT.TYPEMAPPINGS Catalog View

Column Name	Data Type	Nullable	Description
TYPE_MAPPING	VARCHAR(18)		Name of the type mapping (may be system-generated).
TYPESCHEMA	VARCHAR(128)	Yes	Schema name of the type. Null for system built-in types.
TYPENAME	VARCHAR(18)		Name of the local type in a data type mapping.
TYPEID	SMALLINT		Type identifier.
SOURCETYPEID	SMALLINT		Source type identifier.
DEFINER	VARCHAR(128)		Authorization ID under which this type mapping was created.
LENGTH	INTEGER	Yes	Maximum length or precision of the data type. If null, the system determines the best length/precision.
SCALE	SMALLINT	Yes	Scale for DECIMAL fields. If null, the system determines the best scale attribute.
BIT_DATA	CHAR(1)	Yes	Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type or that the system determines the bit data attribute.
WRAPNAME	VARCHAR(128)	Yes	Mapping applies to this wrapper.
SERVERNAME	VARCHAR(128)	Yes	Name of the data source.
SERVERTYPE	VARCHAR(30)	Yes	Mapping applies to this type of data source.
SERVERVERSION	VARCHAR(18)	Yes	Mapping applies to this version of data source with type specified in SERVERTYPE.
REMOTE_TYPESCHEMA	VARCHAR(128)	Yes	Schema name of the remote type.
REMOTE_TYPENAME	VARCHAR(128)		Name of the data type as defined on the data source(s).
REMOTE_META_TYPE	CHAR(1)	Yes	S = Remote type is a system built-in type. T = Remote type is a distinct type.
REMOTE_LOWER_LEN	INTEGER	Yes	Lower bound of the length or precision of the remote decimal type. For character data types, this field indicates the number of characters. 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_UPPER_LEN	INTEGER	Yes	Upper bound of the length or precision of the remote decimal type. For character data types, this field indicates the number of characters. 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	Yes	Lower bound of the scale of the remote type.

Table 114. SYSCAT.TYPEMAPPINGS Catalog View (continued)

Column Name	Data Type	Nullable	Description
REMOTE_UPPER_SCALE	SMALLINT	Yes	Upper bound of the scale of the remote type.
REMOTE_S_OPR_P	CHAR(2)	Yes	Relationship between remote scale and remote precision. Basic comparison operators can be used. A null value indicates that no specific relationship is required.
REMOTE_BIT_DATA	CHAR(1)	Yes	Y = Type is for bit data. N = Type is not for bit data. NULL = This is not a character data type, or the system will determine the bit data attribute.
USER_DEFINED	CHAR(1)		Definition supplied by user.
CREATE_TIME	TIMESTAMP		Time at which this mapping was created.
REMARKS	VARCHAR(254)	Yes	User supplied comments, or null.

SYSCAT.USEROPTIONS

Each row contains server specific option values.

Table 115. SYSCAT.USEROPTIONS Catalog View

Column Name	Data Type	Nullable	Description
AUTHID	VARCHAR(128)		Local authorization ID (always uppercase)
SERVERNAME	VARCHAR(128)		Name of the server for which the user is defined.
OPTION	VARCHAR(128)		Name of the user options.
SETTING	VARCHAR(255)		Value.

SYSCAT.VIEWS

Contains one or more rows for each view that is created.

Table 116. SYSCAT.VIEWS Catalog View

Column Name	Data Type	Nullable	Description
VIEWSCHEMA	VARCHAR(128)		Qualified name of a view or the qualified name of a table that is used to define a materialized query table or a staging table.
VIEWNAME	VARCHAR(128)		
DEFINER	VARCHAR(128)		Authorization ID of the creator of the view.
SEQNO	SMALLINT		Always 1.

VIEWCHECK	CHAR(1)		States the type of view checking:
			N = No check option
			L = Local check option
			C = Cascaded check option

READONLY	CHAR(1)		Y = View is read-only because of its definition. N = View is not read-only.
VALID	CHAR(1)		Y = View or materialized query table definition is valid. X = View or materialized query table definition is inoperative; must be re-created.
QUALIFIER	VARCHAR(128)		Contains value of the default schema at the time of object definition.
FUNC_PATH	VARCHAR(254)		The SQL path of the view creator at the time the view was defined. When the view is used in data manipulation statements, this path must be used to resolve function calls in the view. SYSIBM for views created before Version 2.
TEXT	CLOB(64k)		Text of the CREATE VIEW statement.

SYSCAT.WRAPOPTIONS

Each row contains wrapper specific options.

Table 117. SYSCAT.WRAPOPTIONS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR(128)		Wrapper name.
OPTION	VARCHAR(128)		Name of wrapper option.
SETTING	VARCHAR(255)		Value.

SYSCAT.WRAPPERS

Each row contains information on the registered wrapper.

Table 118. SYSCAT.WRAPPERS Catalog View

Column Name	Data Type	Nullable	Description
WRAPNAME	VARCHAR(128)		Wrapper name.

WRAPTYPE	CHAR(1)		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 associated with this wrapper.
REMARKS	VARCHAR(254)	Yes	User supplied comment, or null.

SYSSTAT.COLDIST

Each row describes the Nth-most-frequent value or Nth quantile value of some column. Statistics are not recorded for inherited columns of typed tables.

Table 119. SYSSTAT.COLDIST Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	VARCHAR(128)		Qualified name of the table to which this entry applies.	
TABNAME	VARCHAR(128)			
COLNAME	VARCHAR(128)		Name of the column to which this entry applies.	
TYPE	CHAR(1)		Type of statistic collected: F = Frequency (most frequent value) Q = Quantile value	
SEQNO	SMALLINT		If TYPE = F, then N in this column identifies the Nth most frequent value. If TYPE = Q, then N in this column identifies the Nth quantile value.	
COLVALUE	VARCHAR(254)	Yes	The data value, as a character literal or a null value. See Note 1. This column can be updated with a valid representation of the value appropriate to the column that the statistic is associated with. If null is the required frequency value, the column should be set to NULL.	Yes
VALCOUNT	BIGINT		If TYPE = F, then VALCOUNT is the number of occurrences of COLVALUE in the column. If TYPE = Q, then VALCOUNT is the number of rows whose value is less than or equal to COLVALUE. This column can be only updated with the following values: • >= 0 (zero)	Yes
DISTCOUNT	BIGINT		If TYPE = q, this column records the number of distinct values that are less than or equal to COLVALUE (null if unavailable.) the number of rows whose value is less than or equal to COLVALUE.	Yes

Notes:

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.

   If the value is changed using an UPDATE statement, only characters in the database code page should be used. Otherwise, substitution characters will occur and will be used when the statistics are applied during query optimization.

SYSSTAT.COLUMNS

Contains one row for each column for which statistics can be updated. Statistics are not recorded for inherited columns of typed tables.

Table 120. SYSSTAT.COLUMNS Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	VARCHAR(128)		Qualified name of the table that contains the column.	
TABNAME	VARCHAR(128)			
COLNAME	VARCHAR(128)		Column name.	
COLCARD	BIGINT		Number of distinct values in the column; −1 if statistics are not gathered; −2 for inherited columns and columns of H-tables. For any column, COLCARD cannot have a value higher than the cardinality of the table containing that column. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
HIGH2KEY	VARCHAR(33)	Yes	Second highest value of the column. This field is empty if statistics are not gathered; for inherited columns; and for columns of H-tables. This column can be updated with a valid representation of a value that is appropriate to the column with which the statistic is associated. See Notes 1 and 2.	Yes
LOW2KEY	VARCHAR(33)	Yes	Second lowest value of the column. This field is empty if statistics are not gathered; for inherited columns; and for columns of H-tables. This column can be updated with a valid representation of a value that is appropriate to the column with which the statistic is associated. See Notes 1 and 2.	Yes
AVGCOLLEN	INTEGER		Average column length. −1 if a long field or LOB, or statistics have not been collected; −2 for inherited columns and columns of H-tables. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
NUMNULLS	BIGINT		Contains the number of nulls in a column. −1 if statistics are not gathered. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
SUB_COUNT	SMALLINT		Average number of sub-elements. Only applicable to character columns. For example, consider the following string: 'database simulation analytical business intelligence'. In this example, SUB_COUNT = 5, because there are five sub-elements in the string.	Yes

Table 120. SYSSTAT.COLUMNS Catalog View (continued)

Column Name	Data Type	Nullable	Description	Updatable
SUB_DELIM_ LENGTH	SMALLINT		Average length of each delimiter separating each sub-element. Only applicable for character columns. For example, consider the following string: 'database simulation analytical business intelligence'. In this example, SUB_DELIM_LENGTH = 1, because each delimiter is a single blank.	Yes

Notes:

1. HIGH2KEY and LOW2KEY each represent an ASCII version of the second highest and the second lowest value in a column, respectively. An ASCII representation of an integer value can be between 1 and 11 characters long, including the sign. The HIGH2KEY and LOW2KEY values are stored in such a way that they can be used on the SET clause of an UPDATE statement. For character strings, this means that single quotation marks are added at the beginning and at the end of the string, and an extra single quotation mark is inserted before every single quotation mark that is already in the string. Only the first 33 characters of the column's value will be considered.

   If you intend to manually populate HIGH2KEY and LOW2KEY values, ensure that:

   - HIGH2KEY and LOW2KEY are valid values for the data type of the corresponding user column.
   - The length of the HIGH2KEY and the LOW2KEY values must be the smaller of: the maximum length for the target column's data type, or 33, not including additional single quotation marks, which can bring the length of the string up to 68.
   - HIGH2KEY must be greater than LOW2KEY whenever there are three or more distinct values in the corresponding column. If there are fewer than three distinct values in the column, HIGH2KEY can be equal to LOW2KEY.
2. 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.

   If the values are changed using an UPDATE statement, only characters in the database code page should be used. Otherwise, substitution characters will occur and will be used when the statistics are applied during query optimization.

SYSSTAT.INDEXES

Contains one row for each index that is defined for a table.

Table 121. SYSSTAT.INDEXES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
INDSCHEMA	VARCHAR(128)		Qualified name of the index.	
INDNAME	VARCHAR(18)			
TABSCHEMA	VARCHAR(128)		Qualifier of the table name.	
TABNAME	VARCHAR(128)		Name of the table or nickname on which the index is defined.	
COLNAMES	CLOB(1M)		List of column names with + or − prefixes.	
NLEAF	INTEGER		Number of leaf pages; −1 if statistics are not gathered. This column can only be updated with the following values: • −1 or > 0 (zero)	Yes
NLEVELS	SMALLINT		Number of index levels; −1 if statistics are not gathered. This column can only be updated with the following values: • −1 or > 0 (zero)	Yes
FIRSTKEYCARD	BIGINT		Number of distinct first key values; −1 if statistics are not gathered. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
FIRST2KEYCARD	BIGINT		Number of distinct keys using the first two columns of the index (−1 if no statistics, or not applicable). This column can only be updated with the following values; • −1 or >= 0 (zero)	Yes
FIRST3KEYCARD	BIGINT		Number of distinct keys using the first three columns of the index (−1 if no statistics, or not applicable). This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
FIRST4KEYCARD	BIGINT		Number of distinct keys using the first four columns of the index (−1 if no statistics, or not applicable). This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes

Table 121. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description	Updatable
FULLKEYCARD	BIGINT		Number of distinct full key values; –1 if statistics are not gathered. This column can only be updated with the following values: • –1 or >= 0 (zero)	Yes
CLUSTERRATIO	SMALLINT		This column is used by the optimizer. It indicates the degree of data clustering with the index; –1 if statistics are not gathered, or if detailed index statistics have been gathered. This column can only be updated with the following values: • –1 or between 0 and 100	Yes
CLUSTERFACTOR	DOUBLE		This column is used by the optimizer. It is a finer measurement of degree of clustering, or –1 if detailed index statistics have not been gathered. This column can only be updated with the following values: • –1 or between 0 and 1	Yes
SEQUENTIAL_PAGES	INTEGER		Number of leaf pages located on disk in index key order with few or no large gaps between them; –1 if no statistics are available. This column can only be updated with the following values: • –1 or >= 0 (zero)	Yes
DENSITY	INTEGER		Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100; –1 if no statistics are available.) This column can only be updated with the following values: • –1 or between 0 and 100	Yes

Table 121. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description	Updatable
PAGE_FETCH_PAIRS	VARCHAR(254)		A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the index using that hypothetical buffer. (Zero-length string if no data available.) This column can be updated with the following input values: • The pair delimiter and pair separator characters are the only non-numeric characters accepted. • Blanks are the only characters recognized as a pair delimiter and pair separator. • Each number entry must have an accompanying partner number entry with the two being separated by the pair separator character. • Each pair must be separated from any other pairs by the pair delimiter character. • Each expected number entry must between 0-9 (positive values only).	Yes
NUMRIDS	BIGINT		Number of RIDs in the index; −1 if statistics are not gathered.	Yes
NUMRIDS_DELETED	BIGINT		Number of RIDs in the index that are marked deleted, excluding the ones on pages on which all RIDs are marked deleted; −1 if statistics are not gathered.	Yes
NUM_EMPTY_LEAFS	BIGINT		Number of leaf pages in the index on which all RIDs are marked deleted; −1 if statistics are not gathered.	Yes
AVERAGE_RANDOM_ FETCH_PAGES	DOUBLE		Average number of random table pages between sequential page accesses when fetching using the index; −1 if it is not known. See Notes 1 and 2.	
AVERAGE_RANDOM_ PAGES	DOUBLE		Average number of random index pages between sequential index page accesses; −1 if it is not known. See Note 2.	
AVERAGE_SEQUENCE_ GAP	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 it is not known. See Note 2.	

Table 121. SYSSTAT.INDEXES Catalog View (continued)

Column Name	Data Type	Nullable	Description	Updatable
AVERAGE_SEQUENCE_ FETCH_GAP	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 it is not known. See Notes 1 and 2.	
AVERAGE_SEQUENCE_ PAGES	DOUBLE		Average number of index pages accessible in sequence (that is, the number of index pages that the prefetchers would detect as being in sequence); –1 if it is not known. See Note 2.	
AVERAGE_SEQUENCE_ FETCH_PAGES	DOUBLE		Average number of table pages 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 it is not known. See Notes 1 and 2.	

Notes:

1. When using DMS table spaces, this statistic cannot be computed.
2. Prefetch statistics are not gathered during a LOAD...STATISTICS YES, or a CREATE INDEX...COLLECT STATISTICS operation, or when the database configuration parameter *seqdetect* is turned off.

SYSSTAT.ROUTINES

Contains a row for each user-defined function (scalar, table, or source), system-generated method, user-defined method, or procedure. Does not include built-in functions. (This catalog view supercedes SYSSTAT.FUNCTIONS. The other view exists, but will remain as it was in DB2 Version 7.1.)

Table 122. SYSSTAT.ROUTINES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
ROUTINESCHEMA	VARCHAR(128)		Qualified routine name.	
ROUTINENAME	VARCHAR(128)			

ROUTINETYPE	CHAR(1)		F = Function
			M = Method
			P = Procedure.	

SPECIFICNAME	VARCHAR(128)		The name of the routine instance (may be system-generated).	
IOS_PER_INVOC	DOUBLE		Estimated number of I/Os per invocation; −1 if not known (0 default). This column can only be updated with −1 or >= 0 (zero).	Yes
INSTS_PER_INVOC	DOUBLE		Estimated number of instructions per invocation; −1 if not known (450 default). This column can only be updated with −1 or >= 0 (zero).	Yes
IOS_PER_ARGBYTE	DOUBLE		Estimated number of I/Os per input argument byte; −1 if not known (0 default). This column can only be updated with −1 or >= 0 (zero).	Yes
INSTS_PER_ARGBYTE	DOUBLE		Estimated number of instructions per input argument byte; −1 if not known (0 default). This column can only be updated with −1 or >= 0 (zero).	Yes
PERCENT_ARGBYTES	SMALLINT		Estimated average percent of input argument bytes that the routine will actually read; −1 if not known (100 default). This column can only be updated with −1 or a number between 0 (zero) and 100.	Yes
INITIAL_IOS	DOUBLE		Estimated number of I/Os performed the first/last time the routine is invoked; −1 if not known (0 default). This column can only be updated with −1 or >= 0 (zero).	Yes
INITIAL_INSTS	DOUBLE		Estimated number of instructions executed the first/last time the routine is invoked; −1 if not known (0 default). This column can only be updated with −1 or >= 0 (zero).	Yes
CARDINALITY	BIGINT		The predicted cardinality of a table function; −1 if not known, or if the routine is not a table function. This column can only be updated with −1 or >= 0 (zero).	Yes
SELECTIVITY	DOUBLE		Used for user-defined predicates; −1 if there are no user-defined predicates. See Note 1.	

Table 122. SYSSTAT.ROUTINES Catalog View (continued)

Column Name	Data Type	Nullable	Description	Updatable

Notes:

1. This column will be set to −1 during migration from DB2 Version 5.2 to 8.1 in the system catalogs for all user-defined functions. 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.

SYSSTAT.TABLES

Contains one row for each *base* table. Views or aliases are, therefore, not included. For typed tables, only the root table of a table hierarchy is included in this view. Statistics are not recorded for inherited columns of typed tables. The CARD value applies to the root table only while the other statistics apply to the entire table hierarchy.

Table 123. SYSSTAT.TABLES Catalog View

Column Name	Data Type	Nullable	Description	Updatable
TABSCHEMA	VARCHAR(128)		Qualified name of the table.	
TABNAME	VARCHAR(128)			
CARD	BIGINT		Total number of rows in the table; −1 if statistics are not gathered. An update to CARD for a table should not attempt to assign it a value less than the COLCARD value of any of the columns in that table. A value of −2 cannot be changed and a column value cannot be directly set to −2. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
NPAGES	INTEGER		Total number of pages on which the rows of the table exist; −1 if statistics are not gathered, and −2 for subtables and H-tables. A value of −2 cannot be changed and a column value cannot be directly set to −2. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
FPAGES	INTEGER		Total number of pages in the file; −1 if statistics are not gathered, and −2 for subtables and H-tables. A value of −2 cannot be changed and a column value cannot be directly set to −2. This column can only be updated with the following values: • −1 or > 0 (zero)	Yes
OVERFLOW	INTEGER		Total number of overflow records in the table; −1 if statistics are not gathered, and −2 for subtables and H-tables. A value of −2 cannot be changed and a column value cannot be directly set to −2. This column can only be updated with the following values: • −1 or >= 0 (zero)	Yes
CLUSTERED	CHAR(1)	Yes	Y = Table is multidimensionally clustered (even if only by one dimension). Null = Table is not clustered.	
ACTIVE_ BLOCKS	INTEGER		Total number of in-use blocks in a multi-dimensional clustering (MDC) table; −1 if statistics are not gathered.	Yes

Appendix E. Federated systems

Valid server types in SQL statements

Server types indicate what kind of data source that the server definition represents. Server types vary by vendor, purpose, and operating system. Supported values depend on the wrapper being used.

For most data sources, you must specify a valid server type in the CREATE SERVER statement.

BioRS wrapper

BioRS data sources.

Server Type	Data Source
Not required in the CREATE SERVER statement.	BioRS

BLAST wrapper

BLAST data sources supported by the BLAST daemon.

Server Type	Data Source
BLASTN	BLAST searches in which a nucleotide sequence is compared with the contents of a nucleotide sequence database to find sequences with regions homologous to regions of the original sequence.
BLASTP	BLAST searches in which an amino acid sequence is compared with the contents of an amino acid sequence database to find sequences with regions homologous to regions of the original sequence.
BLASTX	BLAST searches in which a nucleotide sequence is compared with the contents of an amino acid sequence database to find sequences with regions homologous to regions of the original sequence.
TBLASTN	BLAST searches in which an amino acid sequence is compared with the contents of a nucleotide sequence database to find sequences with regions homologous to regions of the original sequence.
TBLASTX	BLAST searches in which a nucleotide sequence is compared with the contents of a nucleotide sequence database to find sequences with regions homologous to regions of the original sequence.

CTLIB wrapper

Sybase data sources supported by the CTLIB client software.

Server Type	Data Source
SYBASE	Sybase

Documentum wrapper

Documentum data sources supported by the Documentum Client API/Library.

Server Type	Data Source
DCTM	Documentum

DRDA wrapper

DB2 Family data sources

Table 124. DB2 for Linux, UNIX, and Windows

Server Type	Data Source
DB2/UDB	IBM DB2 Universal Database
DB2/6000	IBM DB2 for AIX
DB2/AIX	IBM DB2 for AIX
DB2/HPUX	IBM DB2 for HP-UX
DB2/HP	IBM DB2 for HP-UX
DB2/NT	IBM DB2 for Windows NT
DB2/EEE	IBM DB2 Enterprise-Extended Edition
DB2/SUN	IBM DB2 for Solaris
DB2/PE	IBM DB2 for Personal Edition
DB2/2	IBM DB2 for OS/2
DB2/LINUX	IBM DB2 for Linux
DB2/PTX	IBM DB2 for NUMA-Q
DB2/SCO	IBM DB2 for SCO Unixware

Table 125. DB2 for iSeries (and AS/400)

Server Type	Data Source
DB2/400	IBM DB2 for iSeries and AS/400

Table 126. DB2 for z/OS and OS/390

Server Type	Data Source
DB2/ZOS	IBM DB2 for z/OS
DB2/390	IBM DB2 for OS/390
DB2/MVS	IBM DB2 for MVS

Table 127. DB2 Server for VM and VSE

Server Type	Data Source
DB2/VM	IBM DB2 for VM
DB2/VSE	IBM DB2 for VSE
SQL/DS	IBM SQL/DS

Entrez wrapper

Entrez data sources.

Server Type	Data Source
NUCLEOTIDE	Entrez
PUBMED	Entrez

Excel wrapper

Excel data sources supported by Microsoft Excel 97, 2000, and 2002.

Server Type	Data Source
Not required in the CREATE SERVER statement.	Microsoft Excel

Extended Search wrapper

Extended Search data sources supported by the Extended Search Client Library.

Server Type	Data Source
Not required in the CREATE SERVER statement.	IBM Lotus Extended Search

HMMER wrapper

HMMER data sources supported by the HMMER daemon.

Server Type	Data Source
PFAM	HMMER
SEARCH	HMMER

Informix wrapper

Informix data sources supported by Informix Client SDK software.

Server Type	Data Source
INFORMIX	Informix

MSSQLODBC3 wrapper

Microsoft SQL Server data sources supported by the DataDirect Connect ODBC 3.6 driver or the ODBC 3.0 (or later) driver

Server Type	Data Source
MSSQLSERVER	Microsoft SQL Server

NET8 wrapper

Oracle data sources supported by Oracle NET8 client software.

Server Type	Data Source
ORACLE	Oracle Version 8.0. or later

ODBC wrapper

ODBC data sources supported by the ODBC 3.x driver.

Server Type	Data Source
ODBC	ODBC

OLE DB wrapper

OLE DB providers compliant with Microsoft OLE DB 2.0 or later.

Server Type	Data Source
Not required in the CREATE SERVER statement.	Any OLE DB provider

Table-structured files wrapper

Table-structured file data sources.

Server Type	Data Source
Not required in the CREATE SERVER statement.	Table-structured files

Teradata wrapper

Teradata data sources supported by the Teradata V2R3, V2R4, and V2R5 client software.

Server Type	Data Source
TERADATA	Teradata

Web services wrapper

Web services data sources.

Server Type	Data Source
Not required in the CREATE SERVER statement.	Any Web services data source.

WebSphere Business Integration wrapper

Business application data sources supported by the WeSphere Business Integration wrapper.

Server Type	Data Source
WBI	WebSphere Business Integration 2.2 or 2.3

XML wrapper

XML data sources.

Server Type	Data Source
Not required in the CREATE SERVER statement.	XML

Nickname column options for federated systems

You can specify column information in the CREATE NICKNAME or ALTER NICKNAME statements using parameters called nickname column options.

The following table lists the nickname column options for each data source.

Table 128. Available nickname column options

Data source	ALL_VALUES	DEFAULT	DELIMITER	DOCUMENT	ESCAPE_INPUT	FOREIGN_KEY	INDEX	IS_REPEATING	NUMERIC_STRING	PRIMARY_KEY	REMOTE_NAME	SOAPACTIONCOLUMN	TEMPLATE	URLCOLUMN	VARCHAR_NO_TRAILING_BLANKS	XPATH
BLAST		X	X				X									
DB2 Universal Database for iSeries									X							
DB2 Universal Database for z/OS and OS/390									X							
DB2 Universal Database for VM and VSE									X							
DB2 Universal Database for Linux, UNIX, and Windows									X							
Documentum	X		X					X			X					

Table 128. Available nickname column options (continued)

Data source	ALL_VALUES	DEFAULT	DELIMITER	DOCUMENT	ESCAPE_INPUT	FOREIGN_KEY	INDEX	IS_REPEATING	NUMERIC_STRING	PRIMARY_KEY	REMOTE_NAME	SOAPACTIONCOLUMN	TEMPLATE	URLCOLUMN	VARCHAR_NO_TRAILING_BLANKS	XPATH
Informix									X							
Microsoft SQL Server									X							
ODBC									X							
OLE DB																
Oracle									X						X	
Sybase									X							
Table-structured files				X												
Teradata									X							
WebSphere Business Integration					X	X				X			X			X
Web services					X	X				X		X	X	X		X
XML				X		X				X						X

Table 129. Column options and their settings

Option	Description and valid settings	Default setting
ALL_VALUES	Specifies that all values of a repeating attribute will be returned, separated by the specified delimiter. If this option is missing or is N, then only the last value of a repeating attribute is returned. The ALL_VALUES option can be specified only for VARCHAR columns for which the IS_REPEATING option is 'Y' (and is invalid when IS_REG_TABLE = 'Y').	

Table 129. Column options and their settings (continued)

Option	Description and valid settings	Default setting
DEFAULT	Specifies a new default value for the following input fixed columns: • E_value • QueryStrands • GapAlign • NMisMatchPenalty • NMatchReward • Matrix • FilterSequence • NumberOfAlignments • GapCost • ExtendedGapCost • WordSize • ThresholdEx This new value overrides the preset default values. The new default value must be of the same type as the indicated value for a given column.	
DELIMITER	For Documentum: Specifies the delimiter string to be used when multiple values of a repeating attribute are concatenated. The delimiter can be one or more characters. This option is valid only for attributes of objects with data type VARCHAR where the IS_REPEATING option is set to Y. For BLAST: The delimiter characters to be used to determine the end point of the definition line information for the column on which this option appears. If more than one character appears in this option's value, then the first occurrence of any one of the characters signals the end of this field's information. The default is end of line. This option is required, unless you want the last specified column to contain the remainder of the definition line.	For Documentum: The default delimiter is a comma. For BLAST: The default delimiter is end of line.

Table 129. Column options and their settings (continued)

Option	Description and valid settings	Default setting
DOCUMENT	For table-structured files: Specifies the kind of table-structured file. This wrapper supports only the value FILE for this option. Only one column can be specified with the DOCUMENT option per nickname. The column that is associated with the DOCUMENT option must be a data type of VARCHAR or CHAR. Using the DOCUMENT nickname column option instead of the FILE_PATH nickname option implies that the file that corresponds to this nickname will be supplied when the query runs. If the DOCUMENT option has the FILE value, the value that is supplied when the query runs is the full path of the file whose schema matches the nickname definition for this nickname. For XML: Specifies that this column is a DOCUMENT column. The value of the DOCUMENT column indicates the type of XML source data that is supplied to the nickname when the query runs. This option is accepted only for columns of the root nickname (the nickname that identifies the elements at the top level of the XML document). Only one column can be specified with the DOCUMENT option per nickname. The column that is associated with the DOCUMENT option must be a VARCHAR data type. If you use a DOCUMENT column option instead of the FILE_PATH or DIRECTORY_PATH nickname option the document that corresponds to this nickname is supplied when the query runs. The valid values for the DOCUMENT option are: **FILE** Specifies that the value of the nickname column is bound to the path name of a file. The data from this file is supplied when the query runs. **DIRECTORY** Specifies that the value of the nickname column is bound to the path name of a directory that contains multiple XML data files. The XML data from multiple files is supplied when the query runs. The data is in XML files in the specified directory path. The XML wrapper uses only the files with an .xml extension that are located in the directory that you specify. The XML wrapper ignores all other files in this directory. **URI** Specifies that the value of the nickname column is bound to the path name of a remote XML file to which a URI refers. The URI address indicates the remote location of this XML file on the Web. **COLUMN** Specifies that the XML document is stored in a relational column.	

Table 129. Column options and their settings (continued)

Option	Description and valid settings	Default setting
ELEMENT_NAME	Specifies the BioRS element name. The case sensitivity of this name depends on the case sensitivity of the BioRS server and on the value of the CASE_SENSITIVE server option. You must specify the BioRS element name only if it is different from the column name.	
ESCAPE_INPUT	Specifies whether XML special characters are replaced in XML input values or not. Use this option to include XML fragments as input, such as XML fragments with repeating elements. The TEMPLATE column option must be defined on columns that use the ESCAPE_INPUT column option. The column data type must be VARCHAR or CHAR. Valid values are: Y If the XML input contains special characters these are replaced with their counterpart characters that XML uses to represent the input characters. N Input characters are preserved exactly as they appear.	Y
FOREIGN_KEY	Indicates that this nickname is a child nickname and specifies the name of the corresponding parent nickname. A nickname can have at most one FOREIGN_KEY column option. The value for this option is case sensitive. The table that is designated with this option holds a key that is generated by the wrapper. The XPATH option must not be specified for this column. The column can be used only to join parent nicknames and child nicknames. A CREATE NICKNAME statement with a FOREIGN_KEY option will fail if the parent nickname has a different schema name. Unless the nickname that is referred to in a FOREIGN_KEY clause was explicitly defined as lowercase or mixed case by enclosing it in quotation marks in the corresponding CREATE NICKNAME statement, then when you refer to this nickname in the FOREIGN_KEY clause, you must specify the nickname in uppercase. When this option is set on a column, no other option can be set on the column.	
INDEX	The ordinal number of the column on which this option appears in the group of definition line columns. This option is required.	
IS_INDEXED	Indicates whether the corresponding column is indexed (whether the column can be referenced in a predicate). The valid values are Y and N. The value Y can be specified only for columns whose corresponding element is indexed by the BioRS server.	When a nickname is created, this option is automatically added with the value Y to any columns that correspond to a BioRS indexed element.

Table 129. Column options and their settings (continued)

Option	Description and valid settings	Default setting
IS_REPEATING	Indicates whether the column is multi-valued. Valid values are Y and N. Only the last value is returned for: • Non-VARCHAR repeating attributes • VARCHAR columns when ALL_VALUES 'N' is specified To overcome this limitation, you can create a dual definition for the repeating attribute column.	N
NUMERIC_STRING	Specifies whether a column contains strings of numeric characters. **Y** This column contains strings of numeric characters '0', '1', '2', ... '9'. It does not contain blanks. If this column contains only numeric strings followed by trailing blanks, do not specify Y. When you set NUMERIC_STRING to Y for a column, you are informing the optimizer that this column contains no blanks that could interfere with sorting of the column's data. Use this option when the collating sequence of a data source is different from the collating sequence that the federated server uses. Columns that use this option are not excluded from remote evaluation because of a different collating sequence. **N** This column is either not a numeric string column or is a numeric string column that contains blanks.	N
PRIMARY_KEY	Indicates that this nickname is a parent nickname. The column data type must be VARCHAR(16). A nickname can have at most one PRIMARY_KEY column option. YES is the only valid value. The column that is designated with this option holds a key that is generated by the wrapper. The XPATH option must not be specified for this column. The column can be used only to join parent nicknames and child nicknames. When this option is set on a column, no other option can be set on the column.	
REFERENCED_OBJECT	This option is valid only for columns whose BioRS data type is Reference. This option specifies the name of the BioRS databank that is referenced by the current column. The case sensitivity of this name depends on the case sensitivity of the BioRS server and on the value of the CASE_SENSITIVE server option.	
REMOTE_NAME	Specifies the name of the corresponding Documentum attribute or column. This option maps remote attribute or column names to local DB2 UDB column names.	The DB2 UDB column name.
SOAPACTIONCOLUMN	A column to dynamically specify the URI SOAPACTION attribute from the Web Service Description Language (WSDL) format. This option is specified on only the root nickname. When this option is set on a column, no other option can be set on the column.	

Table 129. Column options and their settings (continued)

Option	Description and valid settings	Default setting
TEMPLATE	The column template fragment to use to construct the XML input document. The fragment must conform to the specified template syntax.	
URLCOLUMN	A column to dynamically specify the URL for the Web service endpoint when you run a query. This option is specified on only the root nickname. When this option is set on a column, no other option can be set on the column.	
VARCHAR_NO_ TRAILING_BLANKS	This option applies to data sources that have variable character data types that do not pad the length with trailing blanks during comparison. Some data sources, such as Oracle, do not have blank-padded character comparison semantics that return the same results as the DB2 UDB for Linux, UNIX, and Windows comparison semantics. Set this option when you want it to apply only to a specific VARCHAR or VARCHAR2 column in a data source object. **Y** Trailing blanks are absent from these VARCHAR columns, or the data source has blank-padded character comparison semantics that are similar to the semantics on the federated server. The federated server sends character comparison operations to the data source for processing. **N** Trailing blanks are present in these VARCHAR columns, and the data source has blank-padded character comparison semantics that are different than the federated server. The federated server processes character comparison operations if it is not possible to compensate for equivalent semantics. For example, rewriting the predicate.	N for affected data sources
XPATH	Specifies the XPath expression in the XML document that contains the data that corresponds to this column. The wrapper evaluates the XPath expression after the CREATE NICKNAME statement applies this XPath expression from this XPATH nickname option.	

Related concepts:
- "Pushdown analysis" in the *Federated Systems Guide*

Related tasks:
- "Global optimization" in the *Federated Systems Guide*

Function mapping options for federated systems

DB2 Information Integrator supplies default mappings between existing built-in data source functions and built-in DB2 functions. For most data sources, the default function mappings are in the wrappers. To use a data source function that the federated server does not recognize, you must create a function mapping between a data source function and a counterpart function at the federated database.

The primary purpose of function mapping options, is to provide information about the potential cost of executing a data source function at the data source. Pushdown analysis determines if a function at the data source is able to execute a function in a query. The query optimizer decides if pushing down the function processing to the data source is the least cost alternative.

The statistical information provided in the function mapping definition helps the query optimizer compare the estimated cost of executing the data source function with the estimated cost of executing the DB2 function.

Table 130. Function mapping options and their settings

Option	Valid settings	Default setting
DISABLE	Disable a default function mapping. Valid values are 'Y' and 'N'.	'N'
INITIAL_INSTS	Estimated number of instructions processed the first and last time that the data source function is invoked.	'0'
INITIAL_IOS	Estimated number of I/Os performed the first and last time that the data source function is invoked.	'0'
IOS_PER_ARGBYTE	Estimated number of I/Os expended for each byte of the argument set that's passed to the data source function.	'0'
IOS_PER_INVOC	Estimated number of I/Os per invocation of a data source function.	'0'
INSTS_PER_ARGBYTE	Estimated number of instructions processed for each byte of the argument set that's passed to the data source function.	'0'
INSTS_PER_INVOC	Estimated number of instructions processed per invocation of the data source function.	'450'
PERCENT_ARGBYTES	Estimated average percent of input argument bytes that the data source function will actually read.	'100'
REMOTE_NAME	Name of the data source function.	local name

Server options for federated systems

Server options are used to describe a data source server. Server options specify data integrity, location, security, and performance information. Some server options are available for all data sources, and other server options are data source specific.

The common federated server options for relational data sources are:

- Compatibility options. COLLATING_SEQUENCE, IGNORE_UDT

- Data integrity options. IUD_APP_SVPT_ENFORCE
- Data and time options. DATEFORMAT, TIMEFORMAT, TIMESTAMPFORMAT
- Location options. CONNECTSTRING, DBNAME, IFILE
- Security options. FOLD_ID, FOLD_PW, INFORMIX_LOCK_MODE
- Performance options. COMM_RATE, CPU_RATIO, DB2_MAXIMAL_PUSHDOWN, IO_RATIO, LOGIN_TIMEOUT, PACKET_SIZE, PLAN_HINTS, PUSHDOWN, TIMEOUT, VARCHAR_NO_TRAILING_BLANKS

The following table lists the server definition server options applicable for each relational data source.

Table 131. Server options for relational data sources

Data Source	**CODEPAGE**	**COLLATING_SEQUENCE**	**COMM_RATE**	**CONNECTSTRING**	**CPU_RATIO**	**DATEFORMAT**	**DB2_MAXIMAL_PUSHDOWN**	**DBNAME**	**FOLD_ID**	**FOLD_PW**	**IFILE**	**INFORMIX_LOCK_MODE**	**IO_RATIO**	**IUD_APP_SVPT_ENFORCE**	**LOGIN_TIMEOUT**	**NODE**	**PACKET_SIZE**	**PASSWORD**	**PLAN_HINTS**	**PUSHDOWN**	**TIMEOUT**	**TIMEFORMAT**	**TIMESTAMPFORMAT**	**VARCHAR_NO_TRAILING_BLANKS**
DB2 UDB for iSeries		X	X		X		X	X	X	X			X	X				X		X				X
DB2 UDB for z/OS and OS/390		X	X		X		X	X	X	X			X	X				X		X				X
DB2 for VM and VSE		X	X		X		X	X	X	X			X	X				X		X				X
DB2 UDB for Linux, UNIX, and Windows		X	X		X		X	X	X	X			X	X				X		X				X
Informix		X	X		X		X	X	X	X		X	X	X		X		X		X				
Microsoft SQL Server	X	X	X		X		X	X	X	X			X	X		X		X		X				
ODBC	X	X	X		X	X	X	X	X	X			X	X		X		X		X		X	X	X
OLE DB		X		X																				
Oracle		X	X		X		X		X	X			X			X		X	X	X				X
Sybase		X	X		X		X	X	X	X	X		X		X	X	X	X	X	X	X			
Teradata		X	X		X		X						X	X		X				X				

The following table lists the server definition server options applicable for each nonrelational data source, except WebSphere Business Integration. The server definition server options for WebSphere Business Integration are listed in Table 133 on page 650.

Table 132. Server options for nonrelational data sources.

Data Source	CASE_SENSITIVE	CONTENT_DIR	DAEMON_PORT	ES_HOST	ES_PORT	ES_TRACING	ES_TRACELEVEL	ES_TRACEFILENAME	HMMPFAM_OPTIONS	HMMSEARCH_OPTIONS	MAX_ROWS	NODE	OS_TYPE	PORT	PROCESSORS	PROXU_AUTHID	PROXY_PASSWORD	PROXY_SERVER_NAME	PROXY_SERVER_PORT	PROXY_TYPE	RDBMS_TYPE	SOCKET_TIMEOUT	TIMEOUT	TRANSACTIONS	USE_CLOB_SEQUENCE
BioRS	X											X		X									X		
BLAST			X									X													X
Documentum		X										X	X								X			X	
Entrez											X					X	X	X	X	X		X			
Excel																									
Extended Search				X	X	X	X	X																	
HMMER			X						X	X		X			X										X
Table-structured files																									
Web services																									
XML																X	X	X	X	X		X			

The following table lists the server definition server options applicable for WebSphere Business Integration data sources.

Table 133. Server options for WebSphere Business Integration data sources.

Data Source	APP_TYPE	FAULT_QUEUE	MQ_CONN_NAME	MQ_MANAGER	MQ_RESPONSE_TIMEOUT	MQ_SVRCONN_CHANNELNAME	REQUEST_QUEUE	RESPONSE_QUEUE
WebSphere Business Integration	X	X	X	X	X	X	X	X

The following table describes each server option and lists the valid and default settings.

Table 134. Server options and their settings

Option	Description and valid settings	Default setting
APP_TYPE	The type of remote application. Valid values are 'PSOFT', 'SAP', and 'SIEBEL'. This option is required.	None.
CASE_SENSITIVE	Specifies whether the BioRS server treats names in a case sensitive manner. Valid values are Y or N. 'Y' The BioRS server treats names in a case sensitive manner. 'N' The BioRS server does not treat names in a case sensitive manner In the BioRS product, a configuration parameter controls the case sensitivity of the data that is stored on the BioRS server. The CASE_SENSITIVE option is the DB2 Information Integrator counterpart to that BioRS system configuration parameter. You must synchronize the BioRS server case sensitivity configuration settings in your BioRS system and in DB2 Information Integrator. If you do not keep the case sensitivity configuration settings synchronized between BioRS and DB2 Information Integrator, errors will occur when you attempt to access BioRS data through DB2 Information Integrator. You cannot change or delete the CASE_SENSITIVE option after you create a new BioRS server in DB2 Information Integrator. If you need to change the CASE_SENSITIVE option, you must drop and then create the entire server again. If you drop the BioRS server, you must also create all of the corresponding BioRS nicknames again. DB2 Information Integrator automatically drops all nicknames that correspond to a dropped server.	Y
CODEPAGE	Specifies the DB2 code page identifier corresponding to the coded character set of the data source client configuration. You must specify the client's code page if the client's code page and the federated database code page do not match. For data sources that support Unicode, the CODEPAGE option can be set to the DB2 code page identifier corresponding to the supported Unicode encoding of the data source client.	On UNIX or Windows systems with a non-Unicode federated database: The federated database code page. On UNIX systems with a Unicode federated database: 1208 On Windows systems with a Unicode federated database: 1202

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
COLLATING_ SEQUENCE	Specifies whether the data source uses the same default collating sequence as the federated database, based on the NLS code set and the country/region information. 'Y' The data source has the same collating sequence as the DB2 federated database. 'N' The data source has a different collating sequence than the DB2 federated database collating sequence. 'I' The data source has a different collating sequence than the DB2 federated database collating sequence, and the data source collating sequence is insensitive to case (for example, 'STEWART' and 'StewART' are considered equal).	'N'
COMM_RATE	Specifies the communication rate between the federated server and the data source server. Expressed in megabytes per second. Valid values are greater than 0 and less than $1x10^{23}$. Values can be expressed in any valid REAL notation.	'2'
CONTENT_DIR	Specifies the name of the locally-accessible root directory for storing content files retrieved by the GET_FILE, GET_FILE_DEL, GET_RENDITION, and GET_RENDITION_DEL pseudo columns. It must be writable by all users who can use these pseudo columns.	On UNIX systems: '/tmp' On Windows systems: 'C:\temp'
CONNECTSTRING	Specifies initialization properties needed to connect to an OLE DB provider.	None.
CPU_RATIO	Indicates how much faster or slower a data source CPU runs than the federated server CPU. Valid values are greater than 0 and less than $1x10^{23}$. Values can be expressed in any valid REAL notation. A setting of 1 indicates that the DB2 federated CPU speed and the data source CPU speed have the same CPU speed, a 1:1 ratio. A setting of .5 indicates that the DB2 federated CPU speed is 50% slower than the data source CPUO speed. A setting of 2 indicates that the DB2 federated CPU speed is twice as fast as the data source CPU speed.	'1.0'
DATEFORMAT	The date format used by the data source. Enter the format using 'DD', 'MM', and 'YY' or 'YYYY' to represent the numeric form of the date. You should also specify the delimiter such as a space or comma. For example, to represent the date format for '2003-01-01', use 'YYYY-MM-DD'. This field is nullable.	None.

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
DAEMON_PORT	Specifies the port number on which the daemon will listen for BLAST or HMMER job requests. The port number must be the same number specified in the DAEMON_PORT option of the daemon configuration file.	BLAST: ’4007’; HMMER: ’4098’
DB2_MAXIMAL_ PUSHDOWN	Specifies the primary criteria that the query optimizer uses when choosing an access plan. The query optimizer can choose access plans based on cost or based on the user requirement that as much query processing as possible be performed by the remote data sources. ’Y’ The query optimizer chooses an access plan that pushes down more query operations to the data source than other plans. When several access plans provide the same amount of pushdown, the query optimizer then chooses the plan with the lowest cost. If a materialized query table (MQT) on the federated server can process part or all of the query, then an access plan that includes the materialized query table is might be used. The federated database does not push down queries that result in a Cartesian product. ’N’ The query optimizer chooses an access plan based on cost.	’N’
DBNAME	Name of the data source database that you want the federated server to access. For DB2 database, this value corresponds to a specific database for the initial remote DB2 database connection. This specific database is the database alias for the remote DB2 database that is cataloged at the federated server using the CATALOG DATABASE command or the DB2 Configuration Assistant. Does not apply to Oracle data sources because Oracle instances contain only one database.	None.
ES_HOST	Specifies the fully qualified host name or IP address of the Extended Search server that you want to search. This option is required.	None.
ES_PORT	Specifies the port number where this Extended Search server listens for requests. This option is optional.	’6001’
ES_TRACING	Specifies whether tracing should be enabled for error messages, warning messages, and informational messages that are produced by the remote Extended Search server. Valid values are: ’OFF’ No trace messages will be logged. ’ON’ Trace messages will be logged. This option is optional.	’OFF’

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting

ES_TRACELEVEL	If tracing is enabled, this option specifies the types of messages that will be written to the log file. You can enable and disable the following trace levels independently:
	'C' Critical error messages.
	'N' Noncritical messages.
	'W' Warning messages.
	'I' Informational messages.
	For example:
	`ES_TRACELEVEL 'W'`
	`ES_TRACELEVEL 'CN'`
	This option is optional.
		'C'

ES_TRACEFILENAME	If tracing is enabled, this option specifies the name of a directory and file where messages will be written. This option is optional.	For UNIX operating systems: $INSTHOME/sqllib/log/ ESWrapper.log. For Windows operating systems: %DB2TEMPDIR%\ ESWrapper.log.
FAULT_QUEUE	The name of the fault queue that delivers error messages from the adapter to the wrapper. The name must conform to the specifications for queue names for WebSphere MQ. This is a required option.	None.
FOLD_ID (See notes 1 and 4 at the end of this table.)	Applies to user IDs that the federated server sends to the data source server for authentication. Valid values are: 'U' The federated server folds the user ID to uppercase before sending it to the data source. This is a logical choice for DB2 family and Oracle data sources (See note 2 at end of this table.) 'N' The federated server does nothing to the user ID before sending it to the data source. (See note 2 at end of this table.) 'L' The federated server folds the user ID to lowercase before sending it to the data source. If none of these settings are used, the federated server tries to send the user ID to the data source in uppercase. If the user ID fails, the server tries sending it in lowercase.	None.

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
FOLD_PW (See notes 1, 3 and 4 at the end of this table.)	Applies to passwords that the federated server sends to data sources for authentication. Valid values are: 'U' The federated server folds the password to uppercase before sending it to the data source. This is a logical choice for DB2 family and Oracle data sources. 'N' The federated server does nothing to the password before sending it to the data source. 'L' The federated server folds the password to lowercase before sending it to the data source. If none of these settings are used, the federated server tries to send the password to the data source in uppercase. If the password fails, the server tries sending it in lowercase.	None.
HMMPFAM_OPTIONS	Specifies hmmpfam options such as --null2, --pvm, and --xnu that have no corresponding column name in a reference table that maps options to column names. For example: `HMMPFAM_OPTIONS '--xnu --pvm'` In this example, the daemon runs the HMMPFAM program with options from the WHERE clause of the query, plus the additional options --xnu --pvm.	
HMMSEARCH_ OPTIONS	Allows the user to provide additional command line options to the hmmsearch command. Only valid with type SEARCH. See the HMMER User's Guide for more information.	None.
IFILE	Specifies the path and name of the Sybase Open Client interfaces file. On Windows NT federated servers, the default is %DB2PATH%\interfaces. On UNIX federated servers, the default path and name value is $DB2INSTANCE/sqllib/interfaces.	None.

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
INFORMIX_LOCK_ MODE	Specifies the lock mode to be set for an Informix data source. The Informix wrapper issues the ’SET LOCK MODE’ command immediately after establishing the connection to an Informix data source. Valid values are: **’W’** Sets the Informix lock mode to WAIT. If the wrapper tries to access a locked table or row, Informix waits until the lock is released. **’N’** Sets the Informix lock mode to NOWAIT. If the wrapper tries to access a locked table or row, Informix returns an error. **’n’** Sets the Informix lock mode to WAIT *n* seconds. If the wrapper tries to access a locked table or row and the lock is not released within the specified number of seconds, Informix returns an error.	’W’
IO_RATIO	Denotes how much faster or slower a data source I/O system runs than the federated server I/O system. Valid values are greater than 0 and less than 1x10^{23} . Values can be expressed in any valid REAL notation. A setting of 1 indicates that the DB2 federated I/O speed and the data source I/O speed have the same I/O speed, a 1:1 ratio. A setting of .5 indicates that the DB2 federated I/O speed is 50% slower than the data source I/O speed. A setting of 2 indicates that the DB2 federated I/O speed is twice as fast as the data source I/O speed.	’1.0’
IUD_APP_SVPT_ ENFORCE	Specifies whether the DB2 federated system should enforce detecting or building of application savepoint statements. When set using the SET SERVER OPTION statement, this server option will have no effect with static SQL statements. ’Y’ The federated server rolls back insert, update, or delete transactions if an error occurs in an insert, update, or delete operation and the data source does not enforce application savepoint statements. SQL error code SQL1476N is returned. ’N’ The federated server will not roll back transactions when an error is encountered. Your application must handle the error recovery.	’Y’
LOGIN_TIMEOUT	Specifies the number of seconds for the DB2 federated server to wait for a response from Sybase Open Client to the login request. The default values are the same as for TIMEOUT.	’0’

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
MAX_ROWS	Specifies the number of rows that the federated server returns for a query that uses the Entrez wrapper. You can specify only positive numbers and zero. When you set the option to be zero, you enable queries to retrieve an unlimited number of rows from the NCBI Web site. However, setting the MAX_ROWS server option to zero or to a very high number can impact your query performance. The MAX_ROWS server option is not required.	Microsoft Windows operating systems: 2000 rows. UNIX-based operating systems: 5000 rows.
MQ_CONN_NAME	The hostname or network address of the computer where the Websphere MQ server is running. An example of a connection name is: 9.30.76.151(1420) where 1420 is the port number. If the port number is excluded a default value of 1414 will be used. This option is optional. If it is omitted, the MQSERVER environment variable (if sepcified in db2dj.ini file) is used to select the channel definition. If MQSERVER is not set, the client channel table is used.	The wrapper uses the MQSERVER environment variable, if specified in the db2dj.ini file, to select the channel definition. If the MQSERVER environment variable is not set, the wrapper uses the client channel table.
MQ_MANAGER	The name of the WebSphere MQ manager. Any valid WebSphere MQ manager name. This option is required.	None.
MQ_RESPONSE_ TIMEOUT	The amount of time that the wrapper should wait for a response message from the response queue. The value is in milliseconds. You can specify a special value of -1 to indicate that there is no timeout period. This option is optional.	10000
MQ_SVRCONN_ CHANNELNAME	The name of the server-connection channel on the Webspehere MQ Manager that the wrapper should try to connect to. This parameter can be specified only if the MQ_CONN_NAME server option is specified. The default server-connection channel, SYSTEM.DEF.SVRCONN, is used if this option is omitted.	SYSTEM.DEF.SVRCONN
NODE	Relational data sources: Name by which a data source is defined as an instance to its RDBMS. Documentum: Specifies the actual name of the Documentum Docbase. This option is required. BLAST: Specifies the host name of the system on which the BLAST daemon process is running. This option is required. HMMER: Specifies the host name of the server on which the HMMER daemon process runs. This option is required. BioRS: Specifies the host name of the system on which the BioRS query tool is available. This option is optional.	BioRS: *localhost*

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
OS_TYPE	Specifies the Docbase server's operating system. Valid values are AIX, SOLARIS, and WINDOWS. This option is required.	None.
PACKET_SIZE	Specifies the packet size of the Sybase interfaces file in bytes. If the data source does not support the specified packet size, the connection will fail. Increasing the packet size when each record is very large (for example, when inserting rows into large tables) significantly increases performance. The byte size is a numeric value.	
PASSWORD	Specifies whether passwords are sent to a data source. 'Y' Passwords are sent to the data source and validated. 'N' Passwords are not sent to the data source and not validated.	'Y'
PLAN_HINTS	Specifies whether *plan hints* are to be enabled. Plan hints are statement fragments that provide extra information for data source optimizers. This information can, for certain query types, improve query performance. The plan hints can help the data source optimizer decide whether to use an index, which index to use, or which table join sequence to use. 'Y' Plan hints are to be enabled at the data source if the data source supports plan hints. 'N' Plan hints are not to be enabled at the data source. This option is only available for Oracle and Sybase data sources.	'N'
PORT	Specifies the number of the port the wrapper uses to connect to the BioRS server. This option is optional.	'5014'
PROCESSORS	Specifies the number of processors that the HMMER program uses. This option is equivalent to the --cpu option of the `hmmpfam` command.	None.
PROXY_AUTHID	Specifies the user name to use when the value of PROXY_TYPE is 'SOCKS5'. This field is optional if the value of PROXY_TYPE is 'SOCKS5'. Contact your network administrator for the user name to use. This option is invalid if the PROXY_TYPE is not 'SOCKS5'.	None.
PROXY_PASSWORD	Specifies the password to use when the value of PROXY_TYPE is 'SOCKS5'. This field is optional if the value of PROXY_TYPE is 'SOCKS5'. Contact your network administrator for the password to use. This option is invalid if the PROXY_TYPE is not 'SOCKS5'.	None.

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
PROXY_SERVER_ NAME	Specifies the proxy server name or the IP address. This field is required if the value of PROXY_TYPE is ’HTTP’, ’SOCKS4’, or ’SOCKS5’. Contact your network administrator for the proxy server name or the IP address.	None.
PROXY_SERVER_ PORT	Specifies the proxy server port number. This field is required if the value of PROXY_TYPE is ’HTTP’, ’SOCKS4’, or ’SOCKS5’. Contact your network administrator for the proxy server port number that should be used.	None.
PROXY_TYPE	Specifies the proxy type that is used to access the Internet when behind a firewall. The valid values are ’NONE’, ’HTTP’, ’SOCKS4’, or ’SOCKS5’. The default value is ’NONE’. Contact your network administrator for the type of proxy that is used.	’NONE’
PUSHDOWN	’Y’ DB2 UDB will consider letting the data source evaluate operations. ’N’ DB2 UDB will send the data source SQL statements that include only SELECT with column names. Predicates (such as WHERE=) column and scalar functions (such as MAX and MIN), sorts (such as ORDER BY or GROUP BY), and joins will not be included in any SQL sent to the data source.	’Y’
RDBMS_TYPE	Specifies the RDBMS used by the Docbase. Valid values are DB2, INFORMIX, ORACLE, SQLSERVER or SYBASE. This option is required.	None.
RESPONSE_QUEUE	The name of the response queue that delivers query results from the adapter to the wrapper. The name must conform to the specifications for queue names for WebSphere MQ. This option is required.	None.
REQUEST_QUEUE	The name of the request queue that delivers query requests from the wrapper to the adapter. The name must conform to the specifications for queue names for WebSphere MQ. This option is required.	None.
SOCKET_TIMEOUT	Specifies the maximum time in minutes that the DB2 federated server will wait for results from the proxy server. A valid value is any number that is greater than or equal to zero. The default is zero ’0’. A value of zero denotes an unlimited amount of time to wait.	0
TIMEFORMAT	The time format used by the data source. Enter the format using ’hh12’, ’hh24’, ’mm’, ’ss’, ’AM’, or ’A.M’. For example, to represent the time format of ’16:00:00’, use ’hh24:mm:ss’. To represent the time format of ’8:00:00 AM’, use ’hh12:mm:ss AM’. This field is nullable.	None.

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
TIMESTAMPFORMAT	The timestamp format used by the data source. The format follows that for date and time, plus ’n’ for tenth of a second, ’nn’ for hundredth of a second, ’nnn’ for milliseconds, and so on, up to ’nnnnnn’ for microseconds. For example, to represent the timestamp format of ’2003-01-01-24:00:00.000000’, use ’YYYY-MM-DD-hh24:mm:ss.nnnnnn’. This field is nullable.	None.
TIMEOUT	Sybase: Specifies the number of seconds the DB2 federated server will wait for a response from Sybase Open Client for any SQL statement. The value of *seconds* is a positive whole number in DB2 Universal Database’s integer range. The timeout value that you specify depends on which wrapper you are using. The default behavior of the TIMEOUT option for the Sybase wrappers is 0, which causes DB2 UDB to wait indefinitely for a response. BioRS: Specifies the time, in minutes, that the BioRS wrapper should wait for a response from the BioRS server. The default value is 10. This option is optional.	’0’; BioRS: ’10’
TRANSACTIONS	Specifies the server transaction mode. The valid values are: ’NONE’ No transactions are enabled. ’QUERY’ Transactions are enabled only for Dctm_Query methods. **’ALL’** Transactions are enabled for the Dctm_Query method. ALL has the same function as QUERY in this release.	’QUERY’
USE_CLOB_ SEQUENCE	This option specifies the data type the federated server uses for the BlastSeq or HmmQSeq column. The values can be ’Y’ or ’N’. You can use the CREATE NICKNAME or ALTER NICKNAME statement. to override the default data type for the BlastSeq or HmmQSeq column.	’Y’

Table 134. Server options and their settings (continued)

Option	Description and valid settings	Default setting
VARCHAR_NO_ TRAILING_BLANKS	This option applies to data sources which have variable character data types that do not pad the length with trailing blanks during comparison. Some data sources, such as Oracle, do not have blank-padded character comparison semantics that return the same results as the DB2 for Linux, UNIX, and Windows comparison semantics. Set this option when you want it to apply to all the VARCHAR and VARCHAR2 columns in the data source objects that will be accessed from the designated server. This includes views. **Y** Trailing blanks are absent from these VARCHAR columns, or the data source has blank-padded character comparison semantics that are similar to the semantics on the federated server. The federated server pushes down character comparison operations to the data source for processing. **N** Trailing blanks are present in these VARCHAR columns and the data source has blank-padded character comparison semantics that are different than the federated server. The federated server processes character comparison operations if it is not possible to compensate for equivalent semantics. For example, rewriting the predicate.	N for affected data sources.

Notes on this table:

1. This field is applied regardless of the value specified for authentication.
2. Because DB2 UDB stores user IDs in uppercase, the values 'N' and 'U' are logically equivalent to each other.
3. The setting for FOLD_PW has no effect when the setting for password is 'N'. Because no password is sent, case cannot be a factor.
4. Avoid null settings for either of these options. A null setting can seem attractive because DB2 UDB will make multiple attempts to resolve user IDs and passwords; however, performance might suffer (it is possible that DB2 UDB will send a user ID and password four times before successfully passing data source authentication).

Related concepts:

- "Server characteristics affecting pushdown opportunities" in the *Federated Systems Guide*
- "Server characteristics affecting global optimization" in the *Federated Systems Guide*

Related reference:

- "DROP statement" in the *SQL Reference, Volume 2*
- "ALTER SERVER statement" in the *SQL Reference, Volume 2*
- "CREATE SERVER statement" in the *SQL Reference, Volume 2*

User mapping options for federated systems

These options are valid for all relational data sources. For nonrelational data sources, the REMOTE_AUTHID and REMOTE_PASSWORD options are valid for the following data sources: BioRS, Documentum, Extended Search, and Web services. The GUEST option is valid for the BioRS data source.

These options are used with the CREATE USER MAPPING and ALTER USER MAPPING statements.

Table 135. User mapping options and their settings

Option	Valid settings	Default setting
ACCOUNTING	DRDA: Used to specify a DRDA accounting string. Valid settings include any string of length 255 or less. This option is required only if accounting information needs to be passed. See the DB2 Connect Users Guide for more information.	None
GUEST	Specifies if the wrapper is to use the guest access mode to the BioRS server. **Y** The wrapper uses the guest access mode to the BioRS server. **N** The wrapper does not use the guest access mode to the BioRS server. When set to a value of Y, this option is mutually exclusive with the REMOTE_AUTHID option and the REMOTE_PASSWORD option.	N
REMOTE_AUTHID	Indicates the authorization ID used at the data source. Valid settings include any string of length 255 or less.	The authorization ID you use to connect to the DB2 Universal Database.
REMOTE_DOMAIN	Documentum: Indicates the Windows NT domain used to authenticate users connecting to a Documentum data source. Valid settings include any valid Windows NT domain name.	The default authentication domain for the Documentum database.
REMOTE_PASSWORD	Indicates the authorization password used at the data source. Valid settings include any string of length 32 or less. You do not need to set this option if the following conditions are met: • The database manager configuration parameter AUTHENTICATON is set to SERVER. • When you connected to the DB2 database, you specified an auth ID and password. If your server requires a password and you do not set this option, you must ensure both of the previous conditions are met or the connection will fail.	The password you use to connect to the DB2 Universal Database if both conditions listed in the valid settings column are met.

Related concepts:

- "DB2 Connect and DRDA" in the *DB2 Connect User's Guide*
- "DRDA and data access" in the *DB2 Connect User's Guide*

Wrapper options for federated systems

Wrapper options are used to configure the wrapper or to define how the federated server uses the wrapper. Wrapper options can be set when you create or alter the wrapper.

All relational and nonrelational data sources use the DB2_FENCED wrapper option. The ODBC data source uses the MODULE wrapper option. The Entrez data source uses the EMAIL wrapper option.

Table 136. Wrapper options and their settings

Option	Valid settings	Default setting
DB2_FENCED	Specifies whether the wrapper runs in fenced or trusted mode. Y The wrapper runs in fenced mode. N The wrapper runs in trusted mode.	Relational wrappers: N. Nonrelational wrappers from IBM: N. Nonrelational wrappers from third parties: Y.
EMAIL	Specifies an e-mail address when you register the Entrez wrapper. This e-mail address is included with all queries and allows NCBI to contact you if there are problems, such as too many queries overloading the NCBI servers. This option is required.	
MODULE	Specifies the full path of the library that contains the ODBC Driver Manager implementation or the SQL/CLI implementation. Required for the ODBC wrapper on UNIX federated servers.	On Windows, the default value is odbc32.dll

Related tasks:

- "Altering a wrapper" in the *Federated Systems Guide*

Default forward data type mappings

The two kinds of mappings between data source data types and federated database data types are forward type mappings and reverse type mappings. In a *forward type mapping*, the mapping is from a remote type to a comparable local type.

You can override a default type mapping, or create a new type mapping with the CREATE TYPE MAPPING statement.

These mappings are valid with all the supported versions, unless otherwise noted.

For all default forward data types mapping from a data source to DB2 for Linux, UNIX, and Windows, the DB2 federated schema is SYSIBM.

The following tables show the default forward mappings between DB2 for Linux, UNIX, and Windows data types and data source data types.

DB2 for z/OS and OS/390 data sources

Table 137. DB2 for z/OS and OS/390 forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	255	32672	-	-	-	-	VARCHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CHAR	255	32672	-	-	Y	-	VARCHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
ROWID	-	-	-	-	Y	-	VARCHAR	40	-	Y
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARG	1	16336	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	0	N

DB2 for iSeries data sources

Table 138. DB2 for iSeries forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	255	32672	-	-	-	-	VARCHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CHAR	255	32672	-	-	Y	-	VARCHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
GRAPHIC	128	16336	-	-	-	-	VARGRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
NUMERIC	-	-	-	-	-	-	DECIMAL	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARG	1	16336	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	0	N

DB2 Server for VM and VSE data sources

Table 139. DB2 Server for VM and VSE forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	1	254	-	-	-	-	CHAR	-	0	N
CHAR	1	254	-	-	Y	-	CHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBAHW	-	-	-	-	-	-	SMALLINT	-	0	-
DBAINT	-	-	-	-	-	-	INTEGER	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
FLOAT	4	-	-	-	-	-	REAL	-	-	-
FLOAT	8	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	0	N
VARCHAR	1	32672	-	-	Y	-	VARCHAR	-	0	Y
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPH	1	16336	-	-	-	-	VARGRAPHIC	-	0	N

DB2 for Linux, UNIX, and Windows data sources

Table 140. DB2 for Linux, UNIX, and Windows forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BIGINT	-	-	-	-	-	-	BIGINT	-	0	-
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHAR	-	-	-	-	-	-	CHAR	-	0	N
CHAR	-	-	-	-	Y	-	CHAR	-	0	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	0	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	-	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	0	N
INTEGER	-	-	-	-	-	-	INTEGER	-	0	-
LONGVAR	-	-	-	-	N	-	CLOB	-	-	-
LONGVAR	-	-	-	-	Y	-	BLOB	-	-	-
LONGVARG	-	-	-	-	-	-	DBCLOB	-	-	-
REAL	-	-	-	-	-	-	REAL	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	0	-
TIME	-	-	-	-	-	-	TIME	-	0	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
TIMESTMP	-	-	-	-	-	-	TIMESTAMP	-	0	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	0	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	0	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPHIC	-	0	N
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	0	N

Informix data sources

Table 141. Informix forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	2147483647	-	-
BOOLEAN	-	-	-	-	-	-	CHARACTER	1	-	-
BYTE	-	-	-	-	-	-	BLOB	2147483647	-	-
CHAR	1	254	-	-	-	-	CHARACTER	-	-	-
CHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
CLOB	-	-	-	-	-	-	CLOB	2147483647	-	-
DATE	-	-	-	-	-	-	DATE	4	-	-
DATETIME	0	4	0	4	-	-	DATE	4	-	-
DATETIME	6	10	6	10	-	-	TIME	3	-	-
DATETIME	0	4	6	15	-	-	TIMESTAMP	10	-	-
DATETIME	6	10	11	15	-	-	TIMESTAMP	10	-	-
DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
DECIMAL	32	130	-	-	-	-	DOUBLE	8	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	8	-	-
INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
INTERVAL	-	-	-	-	-	-	VARCHAR	25	-	-
INT8	-	-	-	-	-	-	BIGINT	19	0	-
LVARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
MONEY	1	31	0	31	-	-	DECIMAL	-	-	-
MONEY	32	32	-	-	-	-	DOUBLE	8	-	-
NCHAR	1	254	-	-	-	-	CHARACTER	-	-	-
NCHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
NVARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
REAL	-	-	-	-	-	-	REAL	4	-	-
SERIAL	-	-	-	-	-	-	INTEGER	4	-	-
SERIAL8	-	-	-	-	-	-	BIGINT	-	-	-
SMALLFLOAT	-	-	-	-	-	-	REAL	4	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
TEXT	-	-	-	-	-	-	CLOB	2147483647	-	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-

Table 141. Informix forward default data type mappings (Not all columns shown) (continued)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA

Notes:

- For the Informix DATETIME data type, the DB2 UNIX and Windows federated server uses the Informix high-level qualifer as the REMOTE_LENGTH and the Informix low-level qualifier as the REMOTE_SCALE.

  The Informix qualifiers are the ″TU_″ constants defined in the Informix Client SDK `datatime.h` file. The contstants are:

0 = YEAR	8 = MINUTE	13 = FRACTION(3)
2 = MONTH	10 = SECOND	14 = FRACTION(4)
4 = DAY	11 = FRACTION(1)	15 = FRACTION(5)
6 = HOUR	12 = FRACTION(2)	

Microsoft SQL Server data sources

Table 142. Microsoft SQL Server forward default data type mappings

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
bigint [4]	-	-	-	-	-	-	BIGINT	-	-	-
binary	1	254	-	-	-	-	CHARACTER	-	-	Y
binary	255	8000	-	-	-	-	VARCHAR	-	-	Y
bit	-	-	-	-	-	-	SMALLINT	2	-	-
char	1	254	-	-	-	-	CHAR	-	-	N
char	255	8000	-	-	-	-	VARCHAR	-	-	N
datetime	-	-	-	-	-	-	TIMESTAMP	10	-	-
datetimen	-	-	-	-	-	-	TIMESTAMP	10	-	-
decimal	1	31	0	31	-	-	DECIMAL	-	-	-
decimal	32	38	0	38	-	-	DOUBLE	-	-	-
decimaln	1	31	0	31	-	-	DECIMAL	-	-	-

Table 142. Microsoft SQL Server forward default data type mappings (continued)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
decimaln	32	38	0	38	-	-	DOUBLE	-	-	-
DUMMY65 [1]	1	38	-84	127	-	-	DOUBLE	-	-	-
DUMMY2000 [3]	1	38	-84	127	-	-	DOUBLE	-	-	-
float	-	8	-	-	-	-	DOUBLE	8	-	-
floatn	-	8	-	-	-	-	DOUBLE	8	-	-
float	-	4	-	-	-	-	REAL	4	-	-
floatn	-	4	-	-	-	-	REAL	4	-	-
image	-	-	-	-	-	-	BLOB	2147483647	-	Y
int	-	-	-	-	-	-	INTEGER	4	-	-
intn	-	-	-	-	-	-	INTEGER	4	-	-
money	-	-	-	-	-	-	DECIMAL	19	4	-
moneyn	-	-	-	-	-	-	DECIMAL	19	4	-
| nchar	1	127	-	-	-	-	CHAR	-	-	N
| nchar	128	4000	-	-	-	-	VARCHAR	-	-	N
numeric	1	31	0	31	-	-	DECIMAL	-	-	-
numeric	32	38	0	38	-	-	DOUBLE	8	-	-
numericn	32	38	0	38	-	-	DOUBLE	-	-	-
numericn	1	31	0	31	-	-	DECIMAL	-	-	-
| ntext [2]	-	-	-	-	-	-	CLOB	2147483647	-	Y
| nvarchar	1	4000	-	-	-	-	VARCHAR	-	-	N
real	-	-	-	-	-	-	REAL	4	-	-
smallint	-	-	-	-	-	-	SMALLINT	2	-	-
smalldatetime	-	-	-	-	-	-	TIMESTAMP	10	-	-
smallmoney	-	-	-	-	-	-	DECIMAL	10	4	-
smallmoneyn	-	-	-	-	-	-	DECIMAL	10	4	-
SQL_BIGINT	-	-	-	-	-	-	DECIMAL	-	-	-
| SQL_BIGINT [4]	-	-	-	-	-	-	BIGINT	-	-	-
SQL_BINARY	1	254	-	-	-	-	CHARACTER	-	-	Y
SQL_BINARY	255	8000	-	-	-	-	VARCHAR	-	-	Y
SQL_BIT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_CHAR	1	254	-	-	-	-	CHAR	-	-	N
SQL_CHAR	255	8000	-	-	-	-	VARCHAR	-	-	N

Table 142. Microsoft SQL Server forward default data type mappings (continued)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
SQL_DATE	-	-	-	-	-	-	DATE	4	-	-
SQL_DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_DECIMAL	32	38	0	38	-	-	DOUBLE	8	-	-
SQL_DECIMAL	32	32	0	31	-	-	DOUBLE	8	-	-
SQL_DOUBLE	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_FLOAT	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_GUID [2]	1	4000	-	-	Y	-	VARCHAR	16	-	Y
SQL_INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
SQL_LONGVARCHAR	-	-	-	-	-	-	CLOB	2147483647	-	N
SQL_LONGVARBINARY	-	-	-	-	-	-	BLOB	-	-	Y
SQL_NUMERIC	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_REAL	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_TIME	-	-	-	-	-	-	TIME	3	-	-
SQL_TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	10	-	-
SQL_TINYINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_VARBINARY	1	8000	-	-	-	-	VARCHAR	-	-	Y
SQL_VARCHAR	1	8000	-	-	-	-	VARCHAR	-	-	N
text	-	-	-	-	-	-	CLOB	-	-	N
timestamp	-	-	-	-	-	-	VARCHAR	8		Y
tinyint	-	-	-	-	-	-	SMALLINT	2	-	-
uniqueidentifier [2]	1	4000	-	-	Y	-	VARCHAR	16	-	Y
varbinary	1	8000	-	-	-	-	VARCHAR	-	-	Y
varchar	1	8000	-	-	-	-	VARCHAR	-	-	N

Notes:

1. This type mapping is valid only with Microsoft SQL Server Version 6.5.
2. This type mapping is valid only with Microsoft SQL Server Version 7 and Version 2000.
3. This type mapping is valid only with Windows 2000 operating systems.
4. This type mapping is valid only with Microsoft SQL Server Version 2000.

ODBC data sources

Table 143. ODBC forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
SQL_BIGINT	-	-	-	-	-	-	BIGINT	8	-	-
SQL_BINARY	1	254	-	-	-	-	CHARACTER	-	-	Y
SQL_BINARY	255	32672	-	-	-	-	VARCHAR	-	-	Y
SQL_BIT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_CHAR	1	254	-	-	-	-	CHAR	-	-	N
SQL_CHAR	255	32672	-	-	-	-	VARCHAR	-	-	N
SQL_DECIMAL	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_DECIMAL	32	38	0	38	-	-	DOUBLE	8	-	-
SQL_DOUBLE	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_FLOAT	-	-	-	-	-	-	DOUBLE	8	-	-
SQL_INTEGER	-	-	-	-	-	-	INTEGER	4	-	-
SQL_LONGVARCHAR	-	-	-	-	-	-	CLOB	2147483647	-	N
SQL_LONGVARBINARY	-	-	-	-	-	-	BLOB	-	-	Y
SQL_NUMERIC	1	31	0	31	-	-	DECIMAL	-	-	-
SQL_NUMERIC	32	32	0	31	-	-	DOUBLE	8	-	-
SQL_REAL	-	-	-	-	-	-	REAL	4	-	-
SQL_SMALLINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_TYPE_DATE	-	-	-	-	-	-	DATE	4	-	-
SQL_TYPE_TIME	-	-	-	-	-	-	TIME	3	-	-
SQL_TYPE_TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	10	-	-
SQL_TINYINT	-	-	-	-	-	-	SMALLINT	2	-	-
SQL_VARBINARY	1	32672	-	-	-	-	VARCHAR	-	-	Y
SQL_VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	N
SQL_WCHAR	1	127	-	-	-	-	CHAR	-	-	N
SQL_WCHAR	128	16336	-	-	-	-	VARCHAR	-	-	N
SQL_WVARCHAR	1	16336	-	-	-	-	VARCHAR	-	-	N
SQL_WLONGVARCHAR	-	1073741823	-	-	-	-	CLOB	2147483647	-	N

Oracle NET8 data sources

Table 144. Oracle NET8 forward default data type mappings

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BLOB	0	0	0	0	-	\0	BLOB	2147483647	0	Y
CHAR	1	254	0	0	-	\0	CHAR	0	0	N
CHAR	255	2000	0	0	-	\0	VARCHAR	0	0	N
CLOB	0	0	0	0	-	\0	CLOB	2147483647	0	N
DATE	0	0	0	0	-	\0	TIMESTAMP	0	0	N
FLOAT	1	126	0	0	-	\0	DOUBLE	0	0	N
LONG	0	0	0	0	-	\0	CLOB	2147483647	0	N
LONG RAW	0	0	0	0	-	\0	BLOB	2147483647	0	Y
MLSLABEL	0	0	0	0	-	\0	VARCHAR	255	0	N
NUMBER	1	38	-84	127	-	\0	DOUBLE	0	0	N
NUMBER	1	31	0	31	-	>=	DECIMAL	0	0	N
NUMBER	1	4	0	0	-	\0	SMALLINT	0	0	N
NUMBER	5	9	0	0	-	\0	INTEGER	0	0	N
NUMBER	-	10	0	0	-	\0	DECIMAL	0	0	N
RAW	1	2000	0	0	-	\0	VARCHAR	0	0	Y
ROWID	0	0	0	NULL	-	\0	CHAR	18	0	N
TIMESTAMP [1]	-	-	-	-	-	-	TIMESTAMP	10	-	-
VARCHAR2	1	4000	0	0	-	\0	VARCHAR	0	0	N

Notes:

1. This type mapping is valid only for Oracle 9i (or later) client and server configurations.

Sybase data sources

Table 145. Sybase CTLIB forward default data type mappings

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
binary	1	254	-	-	-	-	CHAR	-	-	Y
binary	255	16384	-	-	-	-	VARCHAR	-	-	Y
bit	-	-	-	-	-	-	SMALLINT	-	-	-
char	1	254	-	-	-	-	CHAR	-	-	N
char	255	16384	-	-	-	-	VARCHAR	-	-	N
char null (see varchar)										
datetime	-	-	-	-	-	-	TIMESTAMP	-	-	-
datetimn	-	-	-	-	-	-	TIMESTAMP	-	-	-
decimal	1	31	0	31	-	-	DECIMAL	-	-	-
decimal	32	38	0	38	-	-	DOUBLE	-	-	-
decimaln	1	31	0	31	-	-	DECIMAL	-	-	-
decimaln	32	38	0	38	-	-	DOUBLE	-	-	-
float	-	4	-	-	-	-	REAL	-	-	-
float	-	8	-	-	-	-	DOUBLE	-	-	-
floatn	-	4	-	-	-	-	REAL	-	-	-
floatn	-	8	-	-	-	-	DOUBLE	-	-	-
image	-	-	-	-	-	-	BLOB	-	-	-
int	-	-	-	-	-	-	INTEGER	-	-	-
intn	-	-	-	-	-	-	INTEGER	-	-	-
money	-	-	-	-	-	-	DECIMAL	19	4	-
moneyn	-	-	-	-	-	-	DECIMAL	19	4	-
nchar	1	254	-	-	-	-	CHAR	-	-	N
nchar	255	16384	-	-	-	-	VARCHAR	-	-	N
nchar null (see nvarchar)										
numeric	1	31	0	31	-	-	DECIMAL	-	-	-
numeric	32	38	0	38	-	-	DOUBLE	-	-	-
numericn	1	31	0	31	-	-	DECIMAL	-	-	-
numericn	32	38	0	38	-	-	DOUBLE	-	-	-
nvarchar	1	16384	-	-	-	-	VARCHAR	-	-	N
real	-	-	-	-	-	-	REAL	-	-	-
smalldatetime	-	-	-	-	-	-	TIMESTAMP	-	-	-

Table 145. Sybase CTLIB forward default data type mappings (continued)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
smallint	-	-	-	-	-	-	SMALLINT	-	-	-
smallmoney	-	-	-	-	-	-	DECIMAL	10	4	-
sysname	1	254	-	-	-	-	CHAR	-	-	N
text	-	-	-	-	-	-	CLOB	-	-	-
timestamp	-	-	-	-	-	-	VARCHAR	8	-	Y
tinyint	-	-	-	-	-	-	SMALLINT	-	-	-
unichar[1]	1	254	-	-	-	-	CHAR	-	-	N
unichar[1]	255	16384	-	-	-	-	VARCHAR	-	-	N
unichar null (see univarchar)										
univarchar[1]	1	16384	-	-	-	-	VARCHAR	-	-	N
varbinary	1	16384	-	-	-	-	VARCHAR	-	-	Y
varchar	1	16384	-	-	-	-	VARCHAR	-	-	N

Notes:

1. Valid for non-Unicode federated databases.

Teradata data sources

Table 146. Teradata forward default data type mappings (Not all columns shown)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
BYTE	1	254	-	-	-	-	CHAR	-	-	Y
BYTE	255	32672	-	-	-	-	VARCHAR	-	-	Y
BYTE	32673	64000	-	-	-	-	BLOB	-	-	-
BYTEINT	-	-	-	-	-	-	SMALLINT	-	-	-
CHAR	1	254	-	-	-	-	CHARACTER	-	-	-

Table 146. Teradata forward default data type mappings (Not all columns shown) (continued)

REMOTE_TYPENAME	REMOTE_LOWER_LEN	REMOTE_UPPER_LEN	REMOTE_LOWER_SCALE	REMOTE_UPPER_SCALE	REMOTE_BIT_DATA	REMOTE_DATA_OPERATORS	FEDERATED_TYPENAME	FEDERATED_LENGTH	FEDERATED_SCALE	FEDERATED_BIT_DATA
CHAR	255	32672	-	-	-	-	VARCHAR	-	-	-
CHAR	32673	64000	-	-	-	-	CLOB	-	-	-
DATE	-	-	-	-	-	-	DATE	-	-	-
DECIMAL	1	18	0	18	-	-	DECIMAL	-	-	-
DOUBLE PRECISION	-	-	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	-	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	1	127	-	-	-	-	GRAPHIC	-	-	-
GRAPHIC	128	16336	-	-	-	-	VARGRAPHIC	-	-	-
GRAPHIC	16337	32000	-	-	-	-	DBCLOB	-	-	-
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
INTERVAL	-	-	-	-	-	-	CHAR	-	-	-
NUMERIC	1	18	0	18	-	-	DECIMAL	-	-	-
REAL	-	-	-	-	-	-	DOUBLE	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	-	-
VARBYTE	1	32762	-	-	-	-	VARCHAR	-	-	Y
VARBYTE	32763	64000	-	-	-	-	BLOB	-	-	-
VARCHAR	1	32672	-	-	-	-	VARCHAR	-	-	-
VARCHAR	32673	64000	-	-	-	-	CLOB	-	-	-
VARGRAPHIC	1	16336	-	-	-	-	VARGRAPHIC	-	-	-
VARGRAPHIC	16337	32000	-	-	-	-	DBCLOB	-	-	-

Related concepts:

- "Forward and reverse data type mappings" in the *Federated Systems Guide*

Related reference:

- "Altering long data types to varchar data types" in the *Federated Systems Guide*
- "Unicode default forward data type mappings - NET8 wrapper" in the *Federated Systems Guide*
- "Unicode default forward data type mappings - Sybase wrapper" in the *Federated Systems Guide*
- "Unicode default forward data type mappings - ODBC wrapper" in the *Federated Systems Guide*

- “Unicode default forward data type mappings - Microsoft SQL Server wrapper” in the *Federated Systems Guide*

Default reverse data type mappings

The two kinds of mappings between data source data types and federated database data types are forward type mappings and reverse type mappings. In a *forward type mapping*, the mapping is from a remote type to a comparable local type. The other type of mapping is a *reverse type mapping*, which is used with transparent DDL to create or modify remote tables.

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

When you define a remote table or view to the DB2 federated database, the definition includes a reverse type mapping. The mapping is from a *local* DB2 for Linux, UNIX, and Windows data type for each column, and the corresponding *remote* data type. For example, there is a default reverse type mapping in which the local type REAL points to the Informix type SMALLFLOAT.

| DB2 for Linux, UNIX, and Windows federated servers do not support mappings
| for LONG VARCHAR, LONG VARGRAPHIC, DATALINK, and user-defined types.

When you use the CREATE TABLE statement to create a remote table, you specify the local data types you want to include in the remote table. These default reverse type mappings will assign corresponding remote types to these columns. For example, suppose that you use the CREATE TABLE statement to define an Informix table with a column C2. You specify BIGINT as the data type for C2 in the statement. The default reverse type mapping of BIGINT depends on which version of Informix you are creating the table on. The mapping for C2 in the Informix table will be to DECIMAL in Informix Version 8 and to INT8 in Informix Version 9.

| You can override a default reverse type mapping, or create a new reverse type
| mapping with the CREATE TYPE MAPPING statement.

The following tables show the default reverse mappings between DB2 for Linux, UNIX, and Windows local data types and remote data source data types.

These mappings are valid with all the supported versions, unless otherwise noted.

DB2 for z/OS and OS/390 data sources

Table 147. DB2 for z/OS and OS/390 reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	N
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	DOUBLE	-	-	–
FLOAT	-	8	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	N

DB2 for iSeries data sources

Table 148. DB2 for iSeries reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHARACTER	-	-	N
CHARACTER	-	-	-	-	Y	-	CHARACTER	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	NUMERIC	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	FLOAT	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	VARG	-	-	N

DB2 for VM and VSE data sources

Table 149. DB2 for VM and VSE reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	-
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	-
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPH	-	-	N

DB2 for Linux, UNIX, and Windows data sources

Table 150. DB2 for Linux, UNIX, and Windows reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	FEDERATED_BIT_DATA
BIGINT	-	8	-	-	-	-	BIGINT	-	-	-
BLOB	-	-	-	-	-	-	BLOB	-	-	-
CHARACTER	-	-	-	-	-	-	CHAR	-	-	N
CHARACTER	-	-	-	-	Y	-	CHAR	-	-	Y
CLOB	-	-	-	-	-	-	CLOB	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DBCLOB	-	-	-	-	-	-	DBCLOB	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	DOUBLE	-	-	-
FLOAT	-	8	-	-	-	-	DOUBLE	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	N
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
| REAL	-	-	-	-	-	-	REAL	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	10	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	N
VARCHAR	-	-	-	-	Y	-	VARCHAR	-	-	Y
VARGRAPH	-	-	-	-	-	-	VARGRAPHIC	-	-	N
| VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	-

Informix data sources

Table 151. Informix reverse default data type mappings

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BIGINT [1]	-	-	-	-	-	-	DECIMAL	19	-	-
BIGINT [2]	-	-	-	-	-	-	INT8	-	-	-
BLOB	1	2147483647	-	-	-	-	BYTE	-	-	-
CHARACTER	-	-	-	-	N	-	CHAR	-	-	-
CHARACTER	-	-	-	-	Y	-	BYTE	-	-	-
CLOB	1	2147483647	-	-	-	-	TEXT	-	-	-
DATE	-	4	-	-	-	-	DATE	-	-	-
DECIMAL	-	-	-	-	-	-	DECIMAL	-	-	-
DOUBLE	-	8	-	-	-	-	FLOAT	-	-	-
INTEGER	-	4	-	-	-	-	INTEGER	-	-	-
REAL	-	4	-	-	-	-	SMALLFLOAT	-	-	-
SMALLINT	-	2	-	-	-	-	SMALLINT	-	-	-
TIME	-	3	-	-	-	-	DATETIME	6	10	-
TIMESTAMP	-	10	-	-	-	-	DATETIME	0	15	-
VARCHAR	1	254	-	-	N	-	VARCHAR	-	-	-
VARCHAR	255	32672	-	-	N	-	TEXT	-	-	-
VARCHAR	-	-	-	-	Y	-	BYTE	-	-	-
VARCHAR [2]	255	2048	-	-	N	-	LVARCHAR	-	-	-
VARCHAR [2]	2049	32672	-	-	N	-	TEXT	-	-	-

Notes:

1. This type mapping is valid only with Informix server Version 8 (or lower).
2. This type mapping is valid only with Informix server Version 9.

For the Informix DATETIME data type, the DB2 UNIX and Windows federated server uses the Informix high-level qualifer as the REMOTE_LENGTH and the Informix low-level qualifier as the REMOTE_SCALE.

The Informix qualifiers are the ″TU_″ constants defined in the Informix Client SDK `datatime.h` file. The contstants are:

0 = YEAR	8 = MINUTE	13 = FRACTION(3)
2 = MONTH	10 = SECOND	14 = FRACTION(4)
4 = DAY	11 = FRACTION(1)	15 = FRACTION(5)
6 = HOUR	12 = FRACTION(2)	

Microsoft SQL Server data sources

Table 152. Microsoft SQL Server reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BIGINT [1]	-	-	-	-	-	-	bigint	-	-	-
BLOB	-	-	-	-	-	-	image	-	-	-
CHARACTER	-	-	-	-	Y	-	binary	-	-	-
CHARACTER	-	-	-	-	N	-	char	-	-	-
CLOB	-	-	-	-	-	-	text	-	-	-
DATE	-	4	-	-	-	-	datetime	-	-	-
DECIMAL	-	-	-	-	-	-	decimal	-	-	-
DOUBLE	-	8	-	-	-	-	float	-	-	-
INTEGER	-	-	-	-	-	-	int	-	-	-
SMALLINT	-	-	-	-	-	-	smallint	-	-	-
REAL	-	4	-	-	-	-	real	-	-	-
TIME	-	3	-	-	-	-	datetime	-	-	-
TIMESTAMP	-	10	-	-	-	-	datetime	-	-	-
VARCHAR	1	8000	-	-	N	-	varchar	-	-	-
VARCHAR	8001	32672	-	-	N	-	text	-	-	-
VARCHAR	1	8000	-	-	Y	-	varbinary	-	-	-
VARCHAR	8001	32672	-	-	Y	-	image	-	-	-

Notes:

1. This type mapping is valid only with Microsoft SQL Server Version 2000.

Oracle NET8 data sources

Table 153. Oracle NET8 reverse default data type mappings

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BLOB	0	2147483647	0	0	Y	\0	BLOB	0	0	Y
CHARACTER	1	254	0	0	N	\0	CHAR	0	0	N
CHARACTER	1	254	0	0	Y	\0	RAW	0	0	Y
CLOB	0	2147483647	0	0	N	\0	CLOB	0	0	N
DATE	0	4	0	0	N	\0	DATE	0	0	N
DECIMAL	0	0	0	0	N	\0	NUMBER	0	0	N
DOUBLE	0	8	0	0	N	\0	FLOAT	126	0	N
FLOAT	0	8	0	0	N	\0	FLOAT	126	0	N
INTEGER	0	4	0	0	N	\0	NUMBER	9	0	N
REAL	0	4	0	0	N	\0	FLOAT	63	0	N
SMALLINT	0	2	0	0	N	\0	NUMBER	4	0	N
TIME	0	3	0	0	N	\0	DATE	0	0	N
TIMESTAMP	0	10	0	0	N	\0	DATE	0	0	N
VARCHAR	1	4000	0	0	N	\0	VARCHAR2	0	0	N
VARCHAR	1	2000	0	0	Y	\0	RAW	0	0	Y

Note: The DB2 Universal Database for Linux, UNIX, and Windows BIGINT data type is not available for transparent DDL. You cannot specify the BIGINT data type in a CREATE TABLE statement when you create a remote Oracle table.

Sybase data sources

Table 154. Sybase CTLIB default reverse data type mappings

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	REMOTE_BIT_DATA
BIGINT	-	-	-	-	-	-	decimal	19	0	-
BLOB	-	-	-	-	-	-	image	-	-	-
CHARACTER	-	-	-	-	N	-	char	-	-	-
CHARACTER	-	-	-	-	Y	-	binary	-	-	-
CLOB	-	-	-	-	-	-	text	-	-	-
DATE	-	-	-	-	-	-	datetime	-	-	-
DECIMAL	-	-	-	-	-	-	decimal	-	-	-
DOUBLE	-	-	-	-	-	-	float	-	-	-
GRAPHIC	-	-	-	-	-	-	unichar	-	-	-
VARGRAPHIC	-	-	-	-	-	-	univarchar	-	-	-
INTEGER	-	-	-	-	-	-	integer	-	-	-
REAL	-	-	-	-	-	-	real	-	-	-
SMALLINT	-	-	-	-	-	-	smallint	-	-	-
TIME	-	-	-	-	-	-	datetime	-	-	-
TIMESTAMP	-	-	-	-	-	-	datetime	-	-	-
VARCHAR[1]	1	255	-	-	N	-	varchar	-	-	-
VARCHAR[1]	256	32672	-	-	N	-	text	-	-	-
VARCHAR [2]	1	16384	-	-	N	-	varchar	-	-	-
VARCHAR [2]	16385	32672	-	-	N	-	text	-	-	-
VARCHAR[1]	1	255	-	-	Y	-	varbinary	-	-	-
VARCHAR[1]	256	32672	-	-	Y	-	image	-	-	-
VARCHAR [2]	1	16384	-	-	Y	-	varbinary	-	-	-
VARCHAR [2]	16385	32672	-	-	Y	-	image	-	-	-

Notes:

1. This type mapping is valid only for CTLIB with Sybase server version 12.0 (or earlier).
2. This type mapping is valid only for CTLIB with Sybase server version 12.5 (or later).

Teradata data sources

Table 155. Teradata reverse default data type mappings (Not all columns shown)

FEDERATED_TYPENAME	FEDERATED_LOWER_LEN	FEDERATED_UPPER_LEN	FEDERATED_LOWER_SCALE	FEDERATED_UPPER_SCALE	FEDERATED_BIT_DATA	FEDERATED_DATA_OPERATORS	REMOTE_TYPENAME	REMOTE_LENGTH	REMOTE_SCALE	FEDERATED_BIT_DATA
BLOB [1]	1	64000	-	-	-	-	VARBYTE	-	-	-
CHARACTER	-	-	-	-	-	-	CHARACTER	-	-	-
CHARACTER	-	-	-	-	Y	-	BYTE	-	-	-
CLOB [2]	1	64000	-	-		-	VARCHAR	-	-	-
DATE	-	-	-	-	-	-	DATE	-	-	-
DBCLOB [3]	1	32000	-	-	-	-	VARGRAPHIC	-	-	-
DECIMAL	1	18	0	18	-	-	DECIMAL	-	-	-
DECIMAL	19	31	0	31	-	-	FLOAT	-	-	-
DOUBLE	-	-	-	-	-	-	FLOAT	-	-	-
GRAPHIC	-	-	-	-	-	-	GRAPHIC	-	-	-
INTEGER	-	-	-	-	-	-	INTEGER	-	-	-
REAL	-	-	-	-	-	-	FLOAT	-	-	-
SMALLINT	-	-	-	-	-	-	SMALLINT	-	-	-
TIME	-	-	-	-	-	-	TIME	-	-	-
TIMESTAMP	-	-	-	-	-	-	TIMESTAMP	-	-	-
VARCHAR	-	-	-	-	-	-	VARCHAR	-	-	-
VARCHAR	-	-	-	-	Y	-	VARBYTE	-	-	-
VARGRAPHIC	-	-	-	-	-	-	VARGRAPHIC	-	-	-

Notes:

1. The Teradata VARBYTE data type can contain only the specified length (1 to 64000) of a DB2 BLOB data type.
2. The Teradata VARCHAR data type can contain only the specified length (1 to 64000) of a DB2 CLOB data type.
3. The Teradata VARGRAPHIC data type can contain only the specified length (1 to 32000) of a DB2 DBCLOB data type.

Related concepts:

- "Forward and reverse data type mappings" in the *Federated Systems Guide*

Appendix F. The SAMPLE database

Many of the code examples in the DB2 documentation use the SAMPLE database. Following is a description of each of the tables in the SAMPLE database. Instructions for creating and dropping the database are also provided. Initial data values for each table are given; a dash (–) indicates a NULL value.

Creating the SAMPLE database

Use the DB2SAMPL command to create the SAMPLE database. To create a database you must have SYSADM authority.

- **When using UNIX-based platforms**

  If you are using the operating system command prompt, issue:

  ```
  sqllib/bin/db2sampl <path>
  ```

  from the home directory of the database manager instance owner, where *path* is an optional parameter specifying the path where the SAMPLE database is to be created. If the path parameter is not specified, the sample database is created in the default path specified by the DFTDBPATH parameter in the database manager configuration file. The schema for DB2SAMPL is the value of the CURRENT SCHEMA special register.
- **When using Windows platforms**

  If you are using the operating system command prompt, issue:

  ```
  db2sampl e
  ```

  where *e* is an optional parameter specifying the drive where the database is to be created. If the drive parameter is not specified, the sample database is created on the same drive as DB2.

Erasing the SAMPLE database

If you do not need to access the SAMPLE database, you can erase it by using the DROP DATABASE command:

```
db2 drop database sample
```

CL_SCHED table

Name:	**CLASS_CODE**	**DAY**	**STARTING**	**ENDING**
Type:	char(7)	smallint	time	time
Desc:	Class Code (room:teacher)	Day # of 4 day schedule	Class Start Time	Class End Time

DEPARTMENT table

Name:	**DEPTNO**	**DEPTNAME**	**MGRNO**	**ADMRDEPT**	**LOCATION**
Type:	char(3) not null	varchar(29) not null	char(6)	char(3) not null	char(16)

DEPARTMENT table

Name:	**DEPTNO**	**DEPTNAME**	**MGRNO**	**ADMRDEPT**	**LOCATION**
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) to which this department reports	Name of the remote location
Values:	A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-
	B01	PLANNING	000020	A00	-
	C01	INFORMATION CENTER	000030	A00	-
	D01	DEVELOPMENT CENTER	-	A00	-
	D11	MANUFACTURING SYSTEMS	000060	D01	-
	D21	ADMINISTRATION SYSTEMS	000070	D01	-
	E01	SUPPORT SERVICES	000050	A00	-
	E11	OPERATIONS	000090	E01	-
	E21	SOFTWARE SUPPORT	000100	E01	-

EMPLOYEE table

Names:	**EMPNO**	**FIRSTNME**	**MIDINIT**	**LASTNAME**	**WORKDEPT**	**PHONENO**	**HIREDATE**
Type:	char(6) not null	varchar(12) not null	char(1) not null	varchar(15) not null	char(3)	char(4)	date
Desc:	Employee number	First name	Middle initial	Last name	Department (DEPTNO) in which the employee works	Phone number	Date of hire

JOB	**EDLEVEL**	**SEX**	**BIRTHDATE**	**SALARY**	**BONUS**	**COMM**
char(8)	smallint not null	char(1)	date	dec(9,2)	dec(9,2)	dec(9,2)
Job	Number of years of formal education	Sex (M male, F female)	Date of birth	Yearly salary	Yearly bonus	Yearly commission

The following table contains the values in the EMPLOYEE table.

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	dec(9,2)	dec(9,2)	dec(9,2)
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01	PRES	18	F	1933-08-24	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10	MANAGER	18	M	1948-02-02	41250	800	3300
000030	SALLY	A	KWAN	C01	4738	1975-04-05	MANAGER	20	F	1941-05-11	38250	800	3060
000050	JOHN	B	GEYER	E01	6789	1949-08-17	MANAGER	16	M	1925-09-15	40175	800	3214
000060	IRVING	F	STERN	D11	6423	1973-09-14	MANAGER	16	M	1945-07-07	32250	500	2580
000070	EVA	D	PULASKI	D21	7831	1980-09-30	MANAGER	16	F	1953-05-26	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15	MANAGER	16	F	1941-05-15	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19	MANAGER	14	M	1956-12-18	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16	SALESREP	19	M	1929-11-05	46500	900	3720
000120	SEAN		O'CONNELL	A00	2167	1963-12-05	CLERK	14	M	1942-10-18	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28	ANALYST	16	F	1925-09-15	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15	ANALYST	18	F	1946-01-19	28420	600	2274
000150	BRUCE		ADAMSON	D11	4510	1972-02-12	DESIGNER	16	M	1947-05-17	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11	DESIGNER	17	F	1955-04-12	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15	DESIGNER	16	M	1951-01-05	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07	DESIGNER	17	F	1949-02-21	21340	500	1707
000190	JAMES	H	WALKER	D11	2986	1974-07-26	DESIGNER	16	M	1952-06-25	20450	400	1636
000200	DAVID		BROWN	D11	4501	1966-03-03	DESIGNER	16	M	1941-05-29	27740	600	2217
000210	WILLIAM	T	JONES	D11	0942	1979-04-11	DESIGNER	17	M	1953-02-23	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29	DESIGNER	18	F	1948-03-19	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21	CLERK	14	M	1935-05-30	22180	400	1774
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05	CLERK	17	M	1954-03-31	28760	600	2301
000250	DANIEL	S	SMITH	D21	0961	1969-10-30	CLERK	15	M	1939-11-12	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11	CLERK	16	F	1936-10-05	17250	300	1380
000270	MARIA	L	PEREZ	D21	9001	1980-09-30	CLERK	15	F	1953-05-26	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24	OPERATOR	17	F	1936-03-28	26250	500	2100
000290	JOHN	R	PARKER	E11	4502	1980-05-30	OPERATOR	12	M	1946-07-09	15340	300	1227
000300	PHILIP	X	SMITH	E11	2095	1972-06-19	OPERATOR	14	M	1936-10-27	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12	OPERATOR	12	F	1931-04-21	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07	FIELDREP	16	M	1932-08-11	19950	400	1596
000330	WING		LEE	E21	2103	1976-02-23	FIELDREP	14	M	1941-07-18	25370	500	2030
000340	JASON	R	GOUNOT	E21	5698	1947-05-05	FIELDREP	16	M	1926-05-17	23840	500	1907

EMP_ACT table

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
Type:	char(6) not null	char(6) not null	smallint not null	dec(5,2)	date	date
Desc:	Employee number	Project number	Activity number	Proportion of employee's time spent on project	Date activity starts	Date activity ends
Values:	000010	AD3100	10	.50	1982-01-01	1982-07-01
	000070	AD3110	10	1.00	1982-01-01	1983-02-01
	000230	AD3111	60	1.00	1982-01-01	1982-03-15
	000230	AD3111	60	.50	1982-03-15	1982-04-15
	000230	AD3111	70	.50	1982-03-15	1982-10-15
	000230	AD3111	80	.50	1982-04-15	1982-10-15
	000230	AD3111	180	1.00	1982-10-15	1983-01-01
	000240	AD3111	70	1.00	1982-02-15	1982-09-15
	000240	AD3111	80	1.00	1982-09-15	1983-01-01
	000250	AD3112	60	1.00	1982-01-01	1982-02-01
	000250	AD3112	60	.50	1982-02-01	1982-03-15
	000250	AD3112	60	.50	1982-12-01	1983-01-01
	000250	AD3112	60	1.00	1983-01-01	1983-02-01
	000250	AD3112	70	.50	1982-02-01	1982-03-15
	000250	AD3112	70	1.00	1982-03-15	1982-08-15
	000250	AD3112	70	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.25	1982-08-15	1982-10-15
	000250	AD3112	80	.50	1982-10-15	1982-12-01
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	.50	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	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

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	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
	000130	xwd	db200130.xwd
	000140	bitmap	db200140.bmp
	000140	gif	db200140.gif

EMP_PHOTO table

Name:	EMPNO	PHOTO_FORMAT	PICTURE
	000140	xwd	db200140.xwd
	000150	bitmap	db200150.bmp
	000150	gif	db200150.gif
	000150	xwd	db200150.xwd
	000190	bitmap	db200190.bmp
	000190	gif	db200190.gif
	000190	xwd	db200190.xwd

EMP_RESUME table

Name:	EMPNO	RESUME_FORMAT	RESUME
Type:	char(6) not null	varchar(10) not null	clob(5k)
Desc:	Employee number	Resume Format	Resume of employee
Values:	000130	ascii	db200130.asc
	000130	script	db200130.scr
	000140	ascii	db200140.asc
	000140	script	db200140.scr
	000150	ascii	db200150.asc
	000150	script	db200150.scr
	000190	ascii	db200190.asc
	000190	script	db200190.scr

IN_TRAY table

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Type:	timestamp	char(8)	char(64)	varchar(3000)
Desc:	Date and Time received	User id of person sending note	Brief description	The note

ORG table

Name:	DEPTNUMB	DEPTNAME	MANAGER	DIVISION	LOCATION
Type:	smallint not null	varchar(14)	smallint	varchar(10)	varchar(13)
Desc:	Department number	Department name	Manager number	Division of corporation	City
Values:	10	Head Office	160	Corporate	New York
	15	New England	50	Eastern	Boston
	20	Mid Atlantic	10	Eastern	Washington
	38	South Atlantic	30	Eastern	Atlanta
	42	Great Lakes	100	Midwest	Chicago
	51	Plains	140	Midwest	Dallas
	66	Pacific	270	Western	San Francisco
	84	Mountain	290	Western	Denver

PROJECT table

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
Type:	char(6) not null	varchar(24) not null	char(3) not null	char(6) not null	dec(5,2)	date	date	char(6)
Desc:	Project number	Project name	Department responsible	Employee responsible	Estimated mean staffing	Estimated start date	Estimated end date	Major project, for a subproject
Values:	AD3100	ADMIN SERVICES	D01	000010	6.5	1982-01-01	1983-02-01	-
	AD3110	GENERAL ADMIN SYSTEMS	D21	000070	6	1982-01-01	1983-02-01	AD3100
	AD3111	PAYROLL PROGRAMMING	D21	000230	2	1982-01-01	1983-02-01	AD3110
	AD3112	PERSONNEL PROGRAMMING	D21	000250	1	1982-01-01	1983-02-01	AD3110
	AD3113	ACCOUNT PROGRAMMING	D21	000270	2	1982-01-01	1983-02-01	AD3110
	IF1000	QUERY SERVICES	C01	000030	2	1982-01-01	1983-02-01	-
	IF2000	USER EDUCATION	C01	000030	1	1982-01-01	1983-02-01	-
	MA2100	WELD LINE AUTOMATION	D01	000010	12	1982-01-01	1983-02-01	-
	MA2110	W L PROGRAMMING	D11	000060	9	1982-01-01	1983-02-01	MA2100
	MA2111	W L PROGRAM DESIGN	D11	000220	2	1982-01-01	1982-12-01	MA2110
	MA2112	W L ROBOT DESIGN	D11	000150	3	1982-01-01	1982-12-01	MA2110
	MA2113	W L PROD CONT PROGS	D11	000160	3	1982-02-15	1982-12-01	MA2110
	OP1000	OPERATION SUPPORT	E01	000050	6	1982-01-01	1983-02-01	-
	OP1010	OPERATION	E11	000090	5	1982-01-01	1983-02-01	OP1000
	OP2000	GEN SYSTEMS SERVICES	E01	000050	5	1982-01-01	1983-02-01	-
	OP2010	SYSTEMS SUPPORT	E21	000100	4	1982-01-01	1983-02-01	OP2000
	OP2011	SCP SYSTEMS SUPPORT	E21	000320	1	1982-01-01	1983-02-01	OP2010
	OP2012	APPLICATIONS SUPPORT	E21	000330	1	1982-01-01	1983-02-01	OP2010
	OP2013	DB/DC SUPPORT	E21	000340	1	1982-01-01	1983-02-01	OP2010
	PL2100	WELD LINE PLANNING	B01	000020	1	1982-01-01	1982-09-15	MA2100

SALES table

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
Type:	date	varchar(15)	varchar(15)	int
Desc:	Date of sales	Employee's last name	Region of sales	Number of sales
Values:	12/31/1995	LUCCHESSI	Ontario-South	1
	12/31/1995	LEE	Ontario-South	3
	12/31/1995	LEE	Quebec	1
	12/31/1995	LEE	Manitoba	2

SALES table

Name:	SALES_DATE	SALES_PERSON	REGION	SALES
	12/31/1995	GOUNOT	Quebec	1
	03/29/1996	LUCCHESSI	Ontario-South	3
	03/29/1996	LUCCHESSI	Quebec	1
	03/29/1996	LEE	Ontario-South	2
	03/29/1996	LEE	Ontario-North	2
	03/29/1996	LEE	Quebec	3
	03/29/1996	LEE	Manitoba	5
	03/29/1996	GOUNOT	Ontario-South	3
	03/29/1996	GOUNOT	Quebec	1
	03/29/1996	GOUNOT	Manitoba	7
	03/30/1996	LUCCHESSI	Ontario-South	1
	03/30/1996	LUCCHESSI	Quebec	2
	03/30/1996	LUCCHESSI	Manitoba	1
	03/30/1996	LEE	Ontario-South	7
	03/30/1996	LEE	Ontario-North	3
	03/30/1996	LEE	Quebec	7
	03/30/1996	LEE	Manitoba	4
	03/30/1996	GOUNOT	Ontario-South	2
	03/30/1996	GOUNOT	Quebec	18
	03/30/1996	GOUNOT	Manitoba	1
	03/31/1996	LUCCHESSI	Manitoba	1
	03/31/1996	LEE	Ontario-South	14
	03/31/1996	LEE	Ontario-North	3
	03/31/1996	LEE	Quebec	7
	03/31/1996	LEE	Manitoba	3
	03/31/1996	GOUNOT	Ontario-South	2
	03/31/1996	GOUNOT	Quebec	1
	04/01/1996	LUCCHESSI	Ontario-South	3
	04/01/1996	LUCCHESSI	Manitoba	1
	04/01/1996	LEE	Ontario-South	8
	04/01/1996	LEE	Ontario-North	-
	04/01/1996	LEE	Quebec	8
	04/01/1996	LEE	Manitoba	9
	04/01/1996	GOUNOT	Ontario-South	3
	04/01/1996	GOUNOT	Ontario-North	1
	04/01/1996	GOUNOT	Quebec	3
	04/01/1996	GOUNOT	Manitoba	7

STAFF table

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	varchar(9)	smallint	char(5)	smallint	dec(7,2)	dec(7,2)
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O'Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	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

STAFFG table (double-byte code pages only)

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
Type:	smallint not null	vargraphic(9)	smallint	graphic(5)	smallint	dec(9,0)	dec(9,0)
Desc:	Employee number	Employee name	Department number	Job type	Years of service	Current salary	Commission
Values:	10	Sanders	20	Mgr	7	18357.50	-
	20	Pernal	20	Sales	8	18171.25	612.45
	30	Marenghi	38	Mgr	5	17506.75	-
	40	O’Brien	38	Sales	6	18006.00	846.55
	50	Hanes	15	Mgr	10	20659.80	-
	60	Quigley	38	Sales	-	16808.30	650.25
	70	Rothman	15	Sales	7	16502.83	1152.00
	80	James	20	Clerk	-	13504.60	128.20

STAFFG table (double-byte code pages only)

Name:	ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
	90	Koonitz	42	Sales	6	18001.75	1386.70
	100	Plotz	42	Mgr	7	18352.80	-
	110	Ngan	15	Clerk	5	12508.20	206.60
	120	Naughton	38	Clerk	-	12954.75	180.00
	130	Yamaguchi	42	Clerk	6	10505.90	75.60
	140	Fraye	51	Mgr	6	21150.00	-
	150	Williams	51	Sales	6	19456.50	637.65
	160	Molinare	10	Mgr	7	22959.20	-
	170	Kermisch	15	Clerk	4	12258.50	110.10
	180	Abrahams	38	Clerk	3	12009.75	236.50
	190	Sneider	20	Clerk	8	14252.75	126.50
	200	Scoutten	42	Clerk	-	11508.60	84.20
	210	Lu	10	Mgr	10	20010.00	-
	220	Smith	51	Sales	7	17654.50	992.80
	230	Lundquist	51	Clerk	3	13369.80	189.65
	240	Daniels	10	Mgr	5	19260.25	-
	250	Wheeler	51	Clerk	6	14460.00	513.30
	260	Jones	10	Mgr	12	21234.00	-
	270	Lea	66	Mgr	9	18555.50	-
	280	Wilson	66	Sales	9	18674.50	811.50
	290	Quill	84	Mgr	10	19818.00	-
	300	Davis	84	Sales	5	15454.50	806.10
	310	Graham	66	Sales	13	21000.00	200.30
	320	Gonzales	66	Sales	4	16858.20	844.00
	330	Burke	66	Clerk	1	10988.00	55.50
	340	Edwards	84	Sales	7	17844.00	1285.00
	350	Gafney	84	Clerk	5	13030.50	188.00

Sample files with BLOB and CLOB data type

This section shows the data found in the EMP_PHOTO files (pictures of employees) and EMP_RESUME files (resumes of employees).

Quintana photo

Figure 14. Dolores M. Quintana

Quintana resume

The following text is found in the db200130.asc and db200130.scr files.

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

Nicholls photo

Figure 15. Heather A. Nicholls

Nicholls resume

The following text is found in the db200140.asc and db200140.scr files.

Resume: Heather A. Nicholls

Personal Information

Address:	844 Don Mills Ave Mellonville, Idaho 83734
Phone:	(208) 555-2310
Birthdate:	January 19, 1946
Sex:	Female
Marital Status:	Single
Height:	5′8″
Weight:	130 lbs.

Department Information

Employee Number:	000140
Dept Number:	C01
Manager:	Sally Kwan
Position:	Analyst
Phone:	(208) 555-1793
Hire Date:	1976-12-15

Education

1972	Computer Engineering, Ph.D. University of Washington
1969	Music and Physics, M.A. Vassar College

Work History

2/83 - present	Architect, OCR Development Designing the architecture of OCR products.

12/76 - 1/83	Text Programmer Optical character recognition (OCR) programming in PL/I.
9/72 - 11/76	Punch Card Quality Analyst Checking punch cards met quality specifications.

Interests

- Model railroading
- Interior decorating
- Embroidery
- Knitting

Adamson photo

Figure 16. Bruce Adamson

Adamson resume

The following text is found in the db200150.asc and db200150.scr files.

Resume: Bruce Adamson

Personal Information

Address:	3600 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-4489
Birthdate:	May 17, 1947
Sex:	Male
Marital Status:	Married
Height:	6′0″
Weight:	175 lbs.

Department Information

Employee Number:	000150
Dept Number:	D11
Manager:	Irving Stern
Position:	Designer
Phone:	(208) 555-4510
Hire Date:	1972-02-12

Education

1971	Environmental Engineering, M.Sc. Johns Hopkins University
1968	American History, B.A. Northwestern University

Work History

8/79 - present	Neural Network Design Developing neural networks for machine intelligence products.
2/72 - 7/79	Robot Vision Development Developing rule-based systems to emulate sight.
9/71 - 1/72	Numerical Integration Specialist Helping bank systems communicate with each other.

Interests

- Racing motorcycles
- Building loudspeakers
- Assembling personal computers
- Sketching

Walker photo

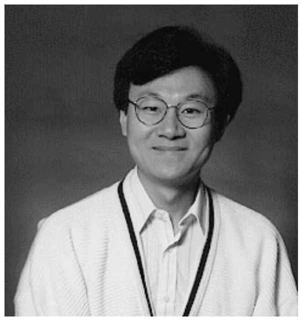


Figure 17. James H. Walker

Walker resume

The following text is found in the db200190.asc and db200190.scr files.

Resume: James H. Walker

Personal Information

Address:	3500 Steeles Ave Mellonville, Idaho 83757
Phone:	(208) 555-7325
Birthdate:	June 25, 1952
Sex:	Male
Marital Status:	Single
Height:	5′11″
Weight:	166 lbs.

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

Walker resume

Appendix G. Reserved schema names and reserved words

There are restrictions on the use of certain names that are required by the database manager. In some cases, names are reserved, and cannot be used by application programs. In other cases, certain names are not recommended for use by application programs, although their use is not prevented by the database manager.

The reserved schema names are:
- SYSCAT
- SYSFUN
- SYSIBM
- SYSSTAT
- SYSPROC

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 user-defined functions, user-defined types, triggers, or aliases can be placed into a schema whose name starts with 'SYS' (SQLSTATE 42939).

The DB2QP schema and the SYSTOOLS schema are set aside for use by DB2 tools. It is recommended that users not explicitly define objects in these schemas, although their use is not prevented by the database manager.

It is also recommended that SESSION not be used as a schema name. Because declared temporary tables must be qualified by SESSION, it is possible to have an application declare a temporary table with a name that is identical to that of a persistent table, complicating the application logic. To avoid this possibility, do not use the schema SESSION except when dealing with declared temporary tables.

There are no specifically reserved words in DB2 Version 8. 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 SQL99 and other DB2 Universal Database products include reserved words that are not enforced by DB2 Universal Database; however, it is recommended that these words not be used as ordinary identifiers, because it reduces portability.

For portability across the DB2 Universal Database products, the following should be considered reserved words:

ADD	DETERMINISTIC	LEAVE	RESTART
AFTER	DISALLOW	LEFT	RESTRICT
ALIAS	DISCONNECT	LIKE	RESULT
ALL	DISTINCT	LINKTYPE	RESULT_SET_LOCATOR
ALLOCATE	DO	LOCAL	RETURN
ALLOW	DOUBLE	LOCALE	RETURNS
ALTER	DROP	LOCATOR	REVOKE
AND	DSNHATTR	LOCATORS	RIGHT
ANY	DSSIZE	LOCK	ROLLBACK
APPLICATION	DYNAMIC	LOCKMAX	ROUTINE

AS	EACH	LOCKSIZE	ROW
ASSOCIATE	EDITPROC	LONG	ROWS
ASUTIME	ELSE	LOOP	RRN
AUDIT	ELSEIF	MAXVALUE	RUN
AUTHORIZATION	ENCODING	MICROSECOND	SAVEPOINT
AUX	END	MICROSECONDS	SCHEMA
AUXILIARY	END-EXEC	MINUTE	SCRATCHPAD
BEFORE	END-EXEC1	MINUTES	SECOND
BEGIN	ERASE	MINVALUE	SECONDS
BETWEEN	ESCAPE	MODE	SECQTY
BINARY	EXCEPT	MODIFIES	SECURITY
BUFFERPOOL	EXCEPTION	MONTH	SELECT
BY	EXCLUDING	MONTHS	SENSITIVE
CACHE	EXECUTE	NEW	SET
CALL	EXISTS	NEW_TABLE	SIGNAL
CALLED	EXIT	NO	SIMPLE
CAPTURE	EXTERNAL	NOCACHE	SOME
CARDINALITY	FENCED	NOCYCLE	SOURCE
CASCADED	FETCH	NODENAME	SPECIFIC
CASE	FIELDPROC	NODENUMBER	SQL
CAST	FILE	NOMAXVALUE	SQLID
CCSID	FINAL	NOMINVALUE	STANDARD
CHAR	FOR	NOORDER	START
CHARACTER	FOREIGN	NOT	STATIC
CHECK	FREE	NULL	STAY
CLOSE	FROM	NULLS	STOGROUP
CLUSTER	FULL	NUMPARTS	STORES
COLLECTION	FUNCTION	OBID	STYLE
COLLID	GENERAL	OF	SUBPAGES
COLUMN	GENERATED	OLD	SUBSTRING
COMMENT	GET	OLD_TABLE	SYNONYM
COMMIT	GLOBAL	ON	SYSFUN
CONCAT	GO	OPEN	SYSIBM
CONDITION	GOTO	OPTIMIZATION	SYSPROC
CONNECT	GRANT	OPTIMIZE	SYSTEM
CONNECTION	GRAPHIC	OPTION	TABLE
CONSTRAINT	GROUP	OR	TABLESPACE
CONTAINS	HANDLER	ORDER	THEN
CONTINUE	HAVING	OUT	TO
COUNT	HOLD	OUTER	TRANSACTION
COUNT_BIG	HOUR	OVERRIDING	TRIGGER
CREATE	HOURS	PACKAGE	TRIM
CROSS	IDENTITY	PARAMETER	TYPE
CURRENT	IF	PART	UNDO
CURRENT_DATE	IMMEDIATE	PARTITION	UNION
CURRENT_LC_CTYPE	IN	PATH	UNIQUE
CURRENT_PATH	INCLUDING	PIECESIZE	UNTIL
CURRENT_SERVER	INCREMENT	PLAN	UPDATE
CURRENT_TIME	INDEX	POSITION	USAGE
CURRENT_TIMESTAMP	INDICATOR	PRECISION	USER
CURRENT_TIMEZONE	INHERIT	PREPARE	USING
CURRENT_USER	INNER	PRIMARY	VALIDPROC
CURSOR	INOUT	PRIQTY	VALUES
CYCLE	INSENSITIVE	PRIVILEGES	VARIABLE
DATA	INSERT	PROCEDURE	VARIANT
DATABASE	INTEGRITY	PROGRAM	VCAT
DAY	INTO	PSID	VIEW
DAYS	IS	QUERYNO	VOLUMES
DB2GENERAL	ISOBID	READ	WHEN
DB2GENRL	ISOLATION	READS	WHERE
DB2SQL	ITERATE	RECOVERY	WHILE
DBINFO	JAR	REFERENCES	WITH
DECLARE	JAVA	REFERENCING	WLM
DEFAULT	JOIN	RELEASE	WRITE
DEFAULTS	KEY	RENAME	YEAR

DEFINITION	LABEL	REPEAT	YEARS
DELETE	LANGUAGE	RESET	
DESCRIPTOR	LC_CTYPE	RESIGNAL	

The following list contains the ISO/ANSI SQL99 reserved words that are not in the previous list:

ABSOLUTE	DESCRIBE	MODULE	SESSION
ACTION	DESTROY	NAMES	SESSION_USER
ADMIN	DESTRUCTOR	NATIONAL	SETS
AGGREGATE	DIAGNOSTICS	NATURAL	SIZE
ARE	DICTIONARY	NCHAR	SMALLINT
ARRAY	DOMAIN	NCLOB	SPACE
ASC	EQUALS	NEXT	SPECIFICTYPE
ASSERTION	EVERY	NONE	SQLEXCEPTION
AT	EXEC	NUMERIC	SQLSTATE
BIT	FALSE	OBJECT	SQLWARNING
BLOB	FIRST	OFF	STATE
BOOLEAN	FLOAT	ONLY	STATEMENT
BOTH	FOUND	OPERATION	STRUCTURE
BREADTH	GROUPING	ORDINALITY	SYSTEM_USER
CASCADE	HOST	OUTPUT	TEMPORARY
CATALOG	IGNORE	PAD	TERMINATE
CLASS	INITIALIZE	PARAMETERS	THAN
CLOB	INITIALLY	PARTIAL	TIME
COLLATE	INPUT	POSTFIX	TIMESTAMP
COLLATION	INT	PREFIX	TIMEZONE_HOUR
COMPLETION	INTEGER	PREORDER	TIMEZONE_MINUTE
CONSTRAINTS	INTERSECT	PRESERVE	TRAILING
CONSTRUCTOR	INTERVAL	PRIOR	TRANSLATION
CORRESPONDING	LARGE	PUBLIC	TREAT
CUBE	LAST	REAL	TRUE
CURRENT_ROLE	LATERAL	RECURSIVE	UNDER
DATE	LEADING	REF	UNKNOWN
DEALLOCATE	LESS	RELATIVE	UNNEST
DEC	LEVEL	ROLE	VALUE
DECIMAL	LIMIT	ROLLUP	VARCHAR
DEFERRABLE	LOCALTIME	SCOPE	VARYING
DEFERRED	LOCALTIMESTAMP	SCROLL	WHENEVER
DEPTH	MAP	SEARCH	WITHOUT
DEREF	MATCH	SECTION	WORK
DESC	MODIFY	SEQUENCE	ZONE

Appendix H. Interaction of triggers and constraints

This appendix describes the interaction of triggers with referential constraints and check constraints that may result from an update operation. Figure 18 and the associated description are representative of the processing that is performed for an SQL statement that updates data in the database.

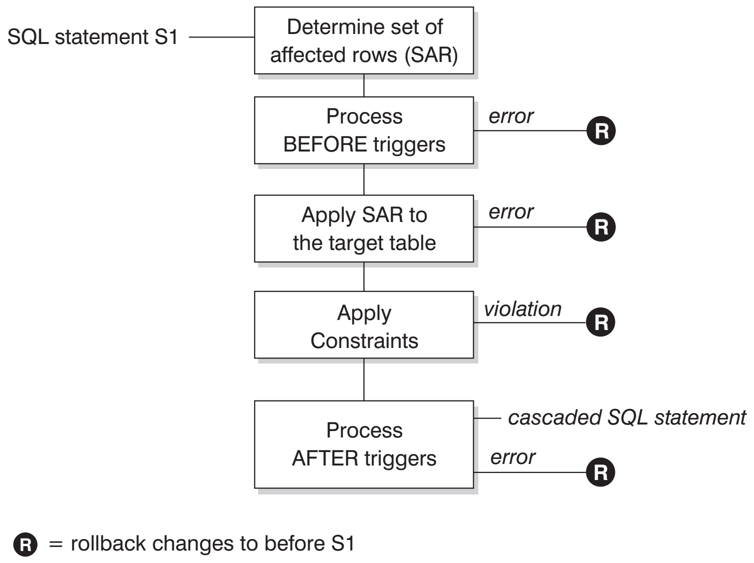


Figure 18. Processing an SQL statement with associated triggers and constraints

Figure 18 shows the general order of processing for an SQL statement that updates a table. It assumes a situation where the table includes before triggers, referential constraints, check constraints and after triggers that cascade. The following is a description of the boxes and other items found in Figure 18.

- SQL statement S_1

  This is the DELETE, INSERT, or UPDATE statement that begins the process. The SQL statement S_1 identifies a table (or an updatable view over some table) referred to as the *target table* throughout this description.

- Determine set of affected rows (*SAR*)

  This step is the starting point for a process that repeats for referential constraint delete rules of CASCADE and SET NULL and for cascaded SQL statements from after triggers.

  The purpose of this step is to determine the *set of affected rows* for the SQL statement. The set of rows included in *SAR* is based on the statement:

  – for DELETE, all rows that satisfy the search condition of the statement (or the current row for a positioned DELETE)
  – for INSERT, the rows identified by the VALUES clause or the fullselect
  – for UPDATE, all rows that satisfy the search condition (or the current row for a positioned update).

  If *SAR* is empty, there will be no BEFORE triggers, changes to apply to the target table, or constraints to process for the SQL statement.

- Process BEFORE triggers

All BEFORE triggers are processed in ascending order of creation. Each BEFORE trigger will process the triggered action once for each row in *SAR*.

An error may occur during the processing of a triggered action in which case all changes made as a result of the original SQL statement S_1 (so far) are rolled back.

If there are no BEFORE triggers or the *SAR* is empty, this step is skipped.

- Apply *SAR* to the target table

  The actual delete, insert, or update is applied using *SAR* to the target table in the database.

  An error may occur when applying *SAR* (such as attempting to insert a row with a duplicate key where a unique index exists) in which case all changes made as a result of the original SQL statement S_1 (so far) are rolled back.
- Apply Constraints

  The constraints associated with the target table are applied if *SAR* is not empty. This includes unique constraints, unique indexes, referential constraints, check constraints and checks related to the WITH CHECK OPTION on views. Referential constraints with delete rules of cascade or set null may cause additional triggers to be activated.

  A violation of any constraint or WITH CHECK OPTION results in an error and all changes made as a result of S_1 (so far) are rolled back.
- Process AFTER triggers

  All AFTER triggers activated by S_1 are processed in ascending order of creation.

  FOR EACH STATEMENT triggers will process the triggered action exactly once, even if *SAR* is empty. FOR EACH ROW triggers will process the triggered action once for each row in *SAR*.

  An error may occur during the processing of a triggered action in which case all changes made as a result of the original S_1 (so far) are rolled back.

  The triggered action of a trigger may include triggered SQL statements that are DELETE, INSERT or UPDATE statements. For the purposes of this description, each such statement is considered a *cascaded SQL statement*.

  A cascaded SQL statement is a DELETE, INSERT, or UPDATE statement that is processed as part of the triggered action of an AFTER trigger. This statement starts a cascaded level of trigger processing. This can be thought of as assigning the triggered SQL statement as a new S_1 and performing all of the steps described here recursively.

  Once all triggered SQL statements from all AFTER triggers activated by each S_1 have been processed to completion, the processing of the original S_1 is completed.
- **R** = roll back changes to before S_1

  Any error (including constraint violations) that occurs during processing results in a roll back of all the changes made directly or indirectly as a result of the original SQL statement S_1. The database is therefore back in the same state as immediately prior to the execution of the original SQL statement S_1

Appendix I. Explain tables

Explain tables

The Explain tables capture access plans when the Explain facility is activated. The Explain tables must be created before Explain can be invoked. You can create them using the documented table definitions, or you can create them by invoking the sample command line processor (CLP) script provided in the EXPLAIN.DDL file located in the misc subdirectory of the sqllib directory. To invoke the script, connect to the database where the Explain tables are required, then issue the command:

```
db2 -tf EXPLAIN.DDL
```

The 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.

Related reference:

- "EXPLAIN_ARGUMENT table" on page 710
- "EXPLAIN_OBJECT table" on page 716
- "EXPLAIN_OPERATOR table" on page 719
- "EXPLAIN_PREDICATE table" on page 721
- "EXPLAIN_STREAM table" on page 725
- "ADVISE_INDEX table" on page 727
- "ADVISE_WORKLOAD table" on page 734
- "EXPLAIN_INSTANCE table" on page 714
- "EXPLAIN_STATEMENT table" on page 723
- "ADVISE_INSTANCE table" on page 730
- "ADVISE_MQT table" on page 731
- "ADVISE_PARTITION table" on page 732
- "ADVISE_TABLE table" on page 733

EXPLAIN_ARGUMENT table

The EXPLAIN_ARGUMENT table represents the unique characteristics for each individual operator, if there are any.

Table 156. 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 157. ARGUMENT_TYPE and ARGUMENT_VALUE column values

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
AGGMODE	COMPLETE PARTIAL INTERMEDIATE FINAL	Partial aggregation indicators.
BITFLTR	INTEGER FALSE	Size of bit filter used by hash join.
BLD_LEVEL	DB2 Build Identifier	Internal identification string for source code version.
BLKLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT SHARE NONE SHARE UPDATE	Block level lock intent.
CSERQY	TRUE FALSE	Remote query is a common subexpression.
CSETEMP	TRUE FALSE	Temporary Table over Common Subexpression Flag.
DIRECT	TRUE	Direct fetch indicator.
DSTSEVER	Server name	Destination (ship from) server.
DUPLWARN	TRUE FALSE	Duplicates Warning flag.

Table 157. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
EARLYOUT	LEFT RIGHT NONE	Early out indicator.
ENVVAR	Each row of this type will contain: • Environment variable name • Environment variable value	Environment variable affecting the optimizer
FETCHMAX	IGNORE INTEGER	Override value for MAXPAGES argument on FETCH operator.
GREEDY	TRUE	Indicates optimizer used greedy algorithm to plan access.
GROUPBYC	TRUE FALSE	Whether Group By columns were provided.
GROUPBYN	Integer	Number of comparison columns.
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.
HASHCODE	24 32	Size (in bits) of hash code used for hash join.
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.
ISCANMAX	IGNORE INTEGER	Override value for MAXPAGES argument on ISCAN operator.
JN_INPUT	INNER OUTER	Indicates if operator is the operator feeding the inner or outer of a join.
LISTENER	TRUE FALSE	Listener Table Queue indicator.
MAXPAGES	ALL NONE INTEGER	Maximum pages expected for Prefetch.
MAXRIDS	NONE INTEGER	Maximum Row Identifiers to be included in each list prefetch request.
NUMROWS	INTEGER	Number of rows expected to be sorted.
ONEFETCH	TRUE FALSE	One Fetch indicator.
OUTERCOL	Each row of this type will contain: • 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.

Table 157. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
OUTERJN	LEFT RIGHT FULL LEFT (ANTI) RIGHT (ANTI)	Outer join indicator.
PARTCOLS	Name of Column	Partitioning columns for operator.
PREFETCH	LIST NONE SEQUENTIAL	Type of Prefetch Eligible.
REOPT	ALWAYS ONCE	The statement is optimized using bind-in values for parameter markers, host variables, and special registers.
RMTQTEXT	Query text	Remote Query Text
RNG_PROD	Function name	Range producing function for extended index access.
ROWLOCK	EXCLUSIVE NONE REUSE SHARE SHORT (INSTANT) SHARE UPDATE	Row Lock Intent.
ROWWIDTH	INTEGER	Width of row to be sorted.
RSUFFIX	Query text	Remote SQL suffix.
SCANDIR	FORWARD REVERSE	Scan Direction.
SCANGRAN	INTEGER	Intra-partition parallelism, granularity of the intra-partition parallel scan, expressed in SCANUNITs.
SCANTYPE	LOCAL PARALLEL	intra-partition parallelism, Index or Table scan.
SCANUNIT	ROW PAGE	Intra-partition parallelism, scan granularity unit.
SHARED	TRUE	Intra-partition parallelism, shared TEMP indicator.
SLOWMAT	TRUE FALSE	Slow Materialization flag.
SNGLPROD	TRUE FALSE	Intra-partition parallelism sort or temp produced by a single agent.
SORTKEY	Each row of this type will contain: • 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	Intra-partition parallelism, sort type.
SRCSEVER	Server name	Source (ship to) server.
SPILLED	INTEGER	Estimated number of pages in SORT spill.
SQLCA	Warning information	Warnings and reason codes issued during Explain operation.

Table 157. ARGUMENT_TYPE and ARGUMENT_VALUE column values (continued)

ARGUMENT_TYPE Value	Possible ARGUMENT_VALUE Values	Description
STMTHEAP	INTEGER	Size of statement heap at start of statement compile.
| STREAM	TRUE FALSE	Remote source is streaming.
TABLOCK	EXCLUSIVE INTENT EXCLUSIVE INTENT NONE INTENT SHARE REUSE SHARE SHARE INTENT EXCLUSIVE SUPER EXCLUSIVE UPDATE	Table Lock Intent.
| TEMPSIZE	INTEGER	Temporary table page size.
TQDEGREE	INTEGER	intra-partition parallelism, number of subagents accessing Table Queue.
TQMERGE	TRUE FALSE	Merging (sorted) Table Queue indicator.
TQREAD	READ AHEAD STEPPING SUBQUERY STEPPING	Table Queue reading property.
TQSEND	BROADCAST DIRECTED SCATTER SUBQUERY DIRECTED	Table Queue send property.
TQTYPE	LOCAL	Intra-partition parallelism, Table Queue.
TRUNCSRT	TRUE	Truncated sort (limits number of rows produced).
UNIQUE	TRUE FALSE	Uniqueness indicator.
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.
VOLATILE	TRUE	Volatile table

EXPLAIN_INSTANCE table

The EXPLAIN_INSTANCE table is the main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. The EXPLAIN_INSTANCE table gives basic information about the source of the SQL statements being explained as well as information about the environment in which the explanation took place.

Table 158. 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
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	Product release number for DB2 Universal Database which processed this explain request. Format is vv.rr.m, where: **vv** Version Number **rr** Release Number **m** Maintenance Release Number
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. For more information, see the BLOCK column in SYSCAT.PACKAGES. Possible values are: **N** No Blocking **U** Block Unambiguous Cursors **B** Block All Cursors

Table 158. EXPLAIN_INSTANCE Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
ISOLATION	CHAR(2)	No	No	Indicates what type of isolation was used when compiling the SQL statements. For more information, see the ISOLATION column in SYSCAT.PACKAGES. Possible values are: **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.
AVG_APPLS	INTEGER	No	No	Contains the value of the AVG_APPLS configuration parameter at the time of the Explain invocation.
SORTHEAP	INTEGER	No	No	Contains the value of the SORTHEAP database configuration setting at the time of the Explain invocation.
LOCKLIST	INTEGER	No	No	Contains the value of the LOCKLIST database configuration setting at the time of the Explain invocation.
MAXLOCKS	SMALLINT	No	No	Contains the value of the MAXLOCKS database configuration setting at the time of the Explain invocation.
LOCKS_AVAIL	INTEGER	No	No	Contains the number of locks assumed to be available by the optimizer for each user. (Derived from LOCKLIST and MAXLOCKS.)
CPU_SPEED	DOUBLE	No	No	Contains the value of the CPUSPEED database manager configuration setting at the time of the Explain invocation.
REMARKS	VARCHAR(254)	Yes	No	User-provided comment.
DBHEAP	INTEGER	No	No	Contains the value of the DBHEAP database configuration setting at the time of Explain invocation.
COMM_SPEED	DOUBLE	No	No	Contains the value of the COMM_BANDWIDTH database configuration setting at the time of Explain invocation.
PARALLELISM	CHAR(2)	No	No	Possible values are: • N = No parallelism • P = Intra-partition parallelism • IP = Inter-partition parallelism • BP = Intra-partition parallelism and inter-partition parallelism
DATAJOINER	CHAR(1)	No	No	Possible values are: • N = Non-federated systems plan • Y = Federated systems plan

EXPLAIN_OBJECT table

The EXPLAIN_OBJECT table identifies those data objects required by the access plan generated to satisfy the SQL statement.

Table 159. 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	INTEGER	No	No	Estimated number of pages that the object occupies in the buffer pool. Set to -1 for a table function.
DISTINCT	CHAR(1)	No	No	Indicates 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.
OVERHEAD	DOUBLE	No	No	Total estimated overhead, in milliseconds, for a single random I/O to the specified table space. Includes controller overhead, disk seek, and latency times. Set to -1 if no table space is involved.
TRANSFER_RATE	DOUBLE	No	No	Estimated time to read a data page, in milliseconds, from the specified table space. Set to -1 if no table space is involved.
PREFETCHSIZE	INTEGER	No	No	Number of data pages to be read when prefetch is performed. Set to -1 for a table function.
EXTENTSIZE	INTEGER	No	No	Size of extent, in data pages. This many pages are written to one container in the table space before switching to the next container. Set to -1 for a table function.

Table 159. 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
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	INTEGER	No	No	Number of leaf pages this index object's values occupy. Set to -1 for a table, table function, or if this statistic is not available.
NLEVELS	INTEGER	No	No	Number of index levels in this index object's tree. Set to -1 for a table, table function, or if this statistic is not available.
FULLKEYCARD	BIGINT	No	No	Number of distinct full key values contained in this index object. Set to -1 for a table, table function, or if this statistic is not available.
OVERFLOW	INTEGER	No	No	Total number of overflow records in the table. Set to -1 for an index, table function, or if this statistic is not available.
FIRSTKEYCARD	BIGINT	No	No	Number of distinct first key values. Set to −1 for a table, table function, or if this statistic is not available.
FIRST2KEYCARD	BIGINT	No	No	Number of distinct first key values using the first {2,3,4} columns of the index. Set to −1 for a table, table function, or if this statistic is not available.
FIRST3KEYCARD	BIGINT	No	No	
FIRST4KEYCARD	BIGINT	No	No	
SEQUENTIAL_PAGES	INTEGER	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. Set to −1 for a table, table function, or if this statistic is not available.
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percentage (integer between 0 and 100). Set to −1 for a table, table function, or if this statistic is not available.
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.

Table 160. Possible OBJECT_TYPE Values

Value	Description
IX	Index

Table 160. Possible OBJECT_TYPE Values (continued)

Value	Description
NK	Nickname
RX	RCT Index
TA	Table
TF	Table Function
+A	Compiler-referenced Alias
+C	Compiler-referenced Constraint
+F	Compiler-referenced Function
+G	Compiler-referenced Trigger
+N	Compiler-referenced Nickname
+T	Compiler-referenced Table
+V	Compiler-referenced View

EXPLAIN_OPERATOR table

The EXPLAIN_OPERATOR table contains all the operators needed to satisfy the SQL statement by the SQL compiler.

Table 161. EXPLAIN_OPERATOR Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	FK	Level of Explain information for which this row is relevant.
STMTNO	INTEGER	No	FK	Statement number within package to which this explain information is related.
SECTNO	INTEGER	No	FK	Section number within package to which this explain information is related.
OPERATOR_ID	INTEGER	No	No	Unique ID for this operator within this query.
OPERATOR_TYPE	CHAR(6)	No	No	Descriptive label for the type of operator.
TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of executing the chosen access plan up to and including this operator.
IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of executing the chosen access plan up to and including this operator.
CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of executing the chosen access plan up to and including this operator.
FIRST_ROW_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the first row for the access plan up to and including this operator. This value includes any initial overhead required.
RE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative cost (in timerons) of fetching the next row for the chosen access plan up to and including this operator.
RE_IO_COST	DOUBLE	No	No	Estimated cumulative I/O cost (in data page I/Os) of fetching the next row for the chosen access plan up to and including this operator.
RE_CPU_COST	DOUBLE	No	No	Estimated cumulative CPU cost (in instructions) of fetching the next row for the chosen access plan up to and including this operator.
COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost (in TCP/IP frames) of executing the chosen access plan up to and including this operator.
FIRST_COMM_COST	DOUBLE	No	No	Estimated cumulative communications cost (in TCP/IP frames) of fetching the first row for the chosen access plan up to and including this operator. This value includes any initial overhead required.
BUFFERS	DOUBLE	No	No	Estimated buffer requirements for this operator and its inputs.
REMOTE_TOTAL_COST	DOUBLE	No	No	Estimated cumulative total cost (in timerons) of performing operation(s) on remote database(s).

Table 161. EXPLAIN_OPERATOR Table (continued). PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
REMOTE_COMM_COST	DOUBLE	No	No	Estimated cumulative communication cost of executing the chosen remote access plan up to and including this operator.

Table 162. 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	Index Scan
MSJOIN	Merge Scan Join
NLJOIN	Nested loop Join
RETURN	Result
RIDSCN	Row Identifier (RID) Scan
SHIP	Ship query to remote system
SORT	Sort
TBSCAN	Table Scan
TEMP	Temporary Table Construction
TQ	Table Queue
UNION	Union
UNIQUE	Duplicate Elimination
UPDATE	Update

EXPLAIN_PREDICATE table

The EXPLAIN_PREDICATE table identifies which predicates are applied by a specific operator.

Table 163. 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.
HOW_APPLIED	CHAR(5)	No	No	How predicate is being used by the specified operator.
WHEN_EVALUATED	CHAR(3)	No	No	Indicates when the subquery used in this predicate is evaluated. Possible values are: **blank** This predicate does not contain a subquery. **EAA** The subquery used in this predicate is evaluated at application (EAA). That is, it is re-evaluated for every row processed by the specified operator, as the predicate is being applied. **EAO** The subquery used in this predicate is evaluated at open (EAO). That is, it is re-evaluated only once for the specified operator, and its results are re-used in the application of the predicate for each row. **MUL** There is more than one type of subquery in this predicate.
RELOP_TYPE	CHAR(2)	No	No	The type of relational operator used in this predicate.
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.

Table 163. 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
PREDICATE_TEXT	CLOB(2M)	Yes	No	The text of the predicate as recreated from the internal representation of the SQL 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 brackets. The value will be stored in the EXPLAIN_PREDICATE table only if the statement is executed by a user who has DBADM authority, or if the DB2 registry variable DB2_VIEW_REOPT_VALUES is set to YES; otherwise, empty brackets will appear at the end of the predicate text. Null if not available.

Table 164. Possible HOW_APPLIED Values

Value	Description
BSARG	Evaluated as a sargable predicate once for every block
JOIN	Used to join tables
RESID	Evaluated as a residual predicate
SARG	Evaluated as a sargable predicate for index or data page
START	Used as a start condition
STOP	Used as a stop condition

Table 165. Possible RELOP_TYPE Values

Value	Description
blanks	Not Applicable
EQ	Equals
GE	Greater Than or Equal
GT	Greater Than
IN	In list
LE	Less Than or Equal
LK	Like
LT	Less Than
NE	Not Equal
NL	Is Null
NN	Is Not Null

EXPLAIN_STATEMENT table

The EXPLAIN_STATEMENT table contains the text of the SQL statement as it exists for the different levels of Explain information. The original SQL statement as entered by the user is stored in this table along with the version used (by the optimizer) to choose an access plan to satisfy the SQL statement. The latter version may bear little resemblance to the original as it may have been rewritten and/or enhanced with additional predicates as determined by the SQL Compiler.

Table 166. EXPLAIN_STATEMENT Table. PK means that the column is part of a primary key; FK means that the column is part of a foreign key.

Column Name	Data Type	Nullable?	Key?	Description
EXPLAIN_REQUESTER	VARCHAR(128)	No	PK, FK	Authorization ID of initiator of this Explain request.
EXPLAIN_TIME	TIMESTAMP	No	PK, FK	Time of initiation for Explain request.
SOURCE_NAME	VARCHAR(128)	No	PK, FK	Name of the package running when the dynamic statement was explained or name of the source file when the static SQL was explained.
SOURCE_SCHEMA	VARCHAR(128)	No	PK, FK	Schema, or qualifier, of source of Explain request.
SOURCE_VERSION	VARCHAR(64)	No	FK	Version of the source of the Explain request.
EXPLAIN_LEVEL	CHAR(1)	No	PK	Level of Explain information for which this row is relevant. Valid values are: **O** Original Text (as entered by user) **P** PLAN SELECTION
STMTNO	INTEGER	No	PK	Statement number within package to which this explain information is related. Set to 1 for dynamic Explain SQL statements. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
SECTNO	INTEGER	No	PK	Section number within package that contains this SQL statement. For dynamic Explain SQL statements, this is the section number used to hold the section for this statement at runtime. For static SQL statements, this value is the same as the value used for the SYSCAT.STATEMENTS catalog view.
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: **S** Select **D** Delete **DC** Delete where current of cursor **I** Insert **U** Update **UC** Update where current of cursor

Table 166. 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
UPDATABLE	CHAR(1)	No	No	Indicates if this statement is considered updatable. This is particularly relevant to SELECT statements which may be determined to be potentially updatable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes
DELETABLE	CHAR(1)	No	No	Indicates if this statement is considered deletable. This is particularly relevant to SELECT statements which may be determined to be potentially deletable. Possible values are: **' '** Not applicable (blank) **N** No **Y** Yes
TOTAL_COST	DOUBLE	No	No	Estimated total cost (in timerons) of executing the chosen access plan for this statement; set to 0 (zero) if EXPLAIN_LEVEL is *O* (original text) since no access plan has been chosen at this time.
STATEMENT_TEXT	CLOB(2M)	No	No	Text or portion of the text of the SQL statement being explained. The text shown for the Plan Selection level of Explain has been reconstructed from the internal representation and is SQL-like in nature; that is, the reconstructed statement is not guaranteed to follow correct SQL syntax.
SNAPSHOT	BLOB(10M)	Yes	No	Snapshot of internal representation for this SQL statement at the Explain_Level shown. This column is intended for use with DB2 Visual Explain. Column is set to null if EXPLAIN_LEVEL is *0* (original statement) since no access plan has been chosen at the time that this specific version of the statement is captured.
QUERY_DEGREE	INTEGER	No	No	Indicates the degree of intra-partition parallelism at the time of Explain invocation. For the original statement, this contains the directed degree of intra-partition parallelism. For the PLAN SELECTION, this contains the degree of intra-partition parallelism generated for the plan to use.

EXPLAIN_STREAM table

The EXPLAIN_STREAM table represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are to be found in the EXPLAIN_OPERATOR table.

Table 167. 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.

Table 167. 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	Partitioning map ID.
SINGLE_NODE	CHAR(5)	Yes	No	Indicates if this data stream is on a single or multiple partitions: **MULT** On multiple 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 **Numberic** Directed to predetermined single node
| PARTITION_COLUMNS	CLOB(2M)	Yes	No	List of columns this data stream is partitioned on.

ADVISE_INDEX table

The ADVISE_INDEX table represents the recommended indexes.

Table 168. 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	INTEGER	No	No	Number of leaf pages; –1 if statistics are not gathered.
NLEVELS	SMALLINT	No	No	Number of index levels; –1 if statistics are not gathered.
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.

Table 168. 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
CLUSTERRATIO	SMALLINT	No	No	Degree of data clustering with the index; -1 if statistics are not gathered or if detailed index statistics are gathered (in which case, CLUSTERFACTOR will be used instead).
CLUSTERFACTOR	DOUBLE	No	No	Finer measurement of degree of clustering, or −1 if detailed index statistics have not been gathered or if the index is defined on a nickname.
USERDEFINED	SMALLINT	No	No	Defined by the user.
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(254)	No	No	A list of pairs of integers, represented in character form. Each pair represents the number of pages in a hypothetical buffer, and the number of page fetches required to scan the table with this index using that hypothetical buffer. (Zero-length string if no data available.)
REMARKS	VARCHAR(254)	Yes	No	User-supplied comment, or null.
DEFINER	VARCHAR(128)	No	No	User who created the index.
CONVERTED	CHAR(1)	No	No	Reserved for future use.
SEQUENTIAL_PAGES	INTEGER	No	No	Number of leaf pages located on disk in index key order with few or no large gaps between them. (−1 if no statistics are available.)
DENSITY	INTEGER	No	No	Ratio of SEQUENTIAL_PAGES to number of pages in the range of pages occupied by the index, expressed as a percent (integer between 0 and 100, −1 if no statistics are available.)
FIRST2KEYCARD	BIGINT	No	No	Number of distinct keys using the first two columns of the index (−1 if no statistics or inapplicable)
FIRST3KEYCARD	BIGINT	No	No	Number of distinct keys using the first three columns of the index (−1 if no statistics or inapplicable)
FIRST4KEYCARD	BIGINT	No	No	Number of distinct keys using the first four columns of the index (−1 if no statistics or inapplicable)
PCTFREE	SMALLINT	No	No	Percentage of each index leaf page to be reserved during initial building of the index. This space is available for future inserts after the index is built.

Table 168. 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
| 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.
RIDTOBLOCK	CHAR(1)	No	No	Set to 'Y' if the RID index was used to make a block index in the Design Advisor.

ADVISE_INSTANCE table

The ADVISE_INSTANCE table contains information about **db2advis** execution, including start time. Contains one row for each execution of **db2advis**. Other ADVISE tables have a foreign key (RUN_ID) that links to the START_TIME column of the ADVISE_INSTANCE table for rows created during the same Design Advisor run.

Table 169. 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 170. 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.
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 171. 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 partition map ID.
TBSPACE	VARCHAR(128)	Yes	No	Specifies the table space in which the table resides.
COLNAMES	CLOB(2M)	Yes	No	Specifies partition column names, separated by commas.
COLCOUNT	SMALLINT	Yes	No	Specifies the number of partitioning columns.
REPLICATE	CHAR(1)	Yes	No	Specifies whether or not the partition is replicated.
COST	DOUBLE	Yes	No	Specifies the cost of using the partition.
USEIT	CHAR(1)	Yes	No	Specifies whether or not the partition is used in EVALUATE PARTITION mode. A 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 partitioning.

Table 172. 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 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 173. 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) that this statments belongs to.
STATEMENT_NO	INTEGER	No	No	Statement number within the workload to which this explain information is related.
STATEMENT_TEXT	CLOB(1M)	No	No	Content of the SQL statement.
STATEMENT_TAG	VARCHAR(256)	No	No	Identifier tag for each explained SQL statement.
FREQUENCY	INTEGER	No	No	The number of times this statement appears within the workload.
IMPORTANCE	DOUBLE	No	No	Importance of the statement.
WEIGHT	DOUBLE	No	No	Priority of the statement.
COST_BEFORE	DOUBLE	Yes	No	The cost (in timerons) of the query if the recommended indexes are not created.
COST_AFTER	DOUBLE	Yes	No	The cost (in timerons) of the query if the recommended indexes are created.
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-character SQLCA.sqlcaid field, followed by a colon (:) and an 8-character SQLCA.sqlstate field, which is the return code for the SQL statement.

Appendix J. Explain register values

Following is a description of the interaction of the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values, both with each other and with the PREP and BIND commands.

With dynamic SQL, the CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact as follows.

Table 174. 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.

Table 174. 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 175. 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).

Table 175. 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 176. 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).

Table 176. 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 K. Exception tables

Exception tables are user-created tables that mimic the definition of the tables that are specified to be checked using SET INTEGRITY with the IMMEDIATE CHECKED option. They are used to store copies of the rows that violate constraints in the tables being checked.

The exception tables used with LOAD are identical to the ones used here. They can therefore be reused during checking with the SET INTEGRITY statement.

Rules for creating an exception table

The rules for creating an exception table are as follows:

- The first "n" columns of the exception table are the same as the columns of the table being checked. All column attributes including name, type and length should be identical.
- All the columns of the exception table must be free of any constraints and triggers. Constraints include referential integrity, check constraints as well as unique index constraints that could cause errors on insert.
- The "(n+1)" column of the exception table is an optional TIMESTAMP column. This serves to identify successive invocations of checking by the SET INTEGRITY statement on the same table, if the rows within the exception table have not been deleted before issuing the SET INTEGRITY statement to check the data.
- The "(n+2)" column should be of type CLOB(32K) or larger. This column is optional but recommended, and will be used to give the names of the constraints that the data within the row violates. If this column is not provided (as could be warranted if, for example, the original table had the maximum number of columns allowed), then only the row where the constraint violation was detected is copied.
- The exception table should be created with both "(n+1)" and "(n+2)" columns.
- There is no enforcement of any particular name for the above additional columns. However, the type specification must be exactly followed.
- No additional columns are allowed.
- If the original table has DATALINK columns, the corresponding columns in the exception table should specify NO LINK CONTROL. This ensures that a file is not linked when a row (with DATALINK column) is inserted and an access token is not generated when rows are selected from the exception table.
- If the original table has generated columns (including the IDENTITY property), the corresponding columns in the exception table should not specify the generated property.
- It should also be noted that users invoking SET INTEGRITY to check the data must have INSERT privilege on the exception tables.

The information in the "message" column will be according to the following structure:

Table 177. Exception Table Message Column Structure

Field number	Contents	Size	Comments
1	Number of constraint violations	5 characters	Right justified padded with '0'
2	Type of first constraint violation	1 character	'K' - Check Constraint violation 'F' - Foreign Key violation 'G' - Generated Column violation 'I' - Unique Index violation[a] 'L' - DATALINK load violation 'D' - Delete Cascade violation
3	Length of constraint/column[b] /index ID[c]/DLVDESC[d]	5 characters	Right justified padded with '0'
4	Constraint name/Column name[b]/index ID[c]/DLVDESC[d]	length from the previous field	
5	Separator	3 characters	<space><colon><space>
6	Type of next constraint violation	1 character	'K' - Check Constraint violation 'F' - Foreign Key violation 'G' - Generated Column violation 'I' - Unique Index violation 'L' - DATALINK load violation 'D' - Delete Cascade violation
7	Length of constraint/column/index ID/ DLVDESC	5 characters	Right justified padded with '0'
8	Constraint name/Column name/Index ID/ DLVDESC	length from the previous field	
.....	.....	.....	Repeat Field 5 through 8 for each violation

- [a] Unique index violations will not occur with checking using SET INTEGRITY. This will be reported, however, when running LOAD if the FOR EXCEPTION option is chosen. LOAD, on the other hand, will not report check constraint, generated column, and foreign key 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 characters 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]DLVDESC is a DATALINK Load Violation DESCriptor described below.

Table 178. DATALINK Load Violation DESCriptor (DLVDESC)

Field number	Contents	Size	Comments
1	Number of violating DATALINK columns	4 characters	Right justified padded with '0'
2	DATALINK column number of the first violating column	4 characters	Right justified padded with '0'
2	DATALINK column number of the second violating column	4 characters	Right justified padded with '0'
.....	.....	.....	Repeat for each violating column number

Note:
- DATALINK column number is COLNO in SYSCAT.COLUMNS for the appropriate table.

Handling rows in an exception table

The information in the exception tables can be processed in any manner desired. The rows could be used to correct the data and re-insert the rows into the original tables.

If there are no INSERT triggers on the original table, transfer the corrected rows by issuing an INSERT statement with a subquery on the exception table.

If there are INSERT triggers and you wish to complete the load with the corrected rows from exception tables without firing the triggers, the following ways are suggested:

- Design the INSERT triggers to be fired depending on the value in a column defined explicitly for the purpose.
- Unload the data from the exception tables and append them using LOAD. In this case if we re-check the data, it should be noted that in DB2 Version 8 the constraint violation checking is not confined to the appended rows only.
- Save the trigger text from the relevant catalog table. Then drop the INSERT trigger and use INSERT to transfer the corrected rows from the exception tables. Finally recreate the trigger using the saved information.

In DB2 Version 8, no explicit provision is made to prevent the firing of triggers when inserting rows from the exception tables.

Only one violation per row will be reported for unique index violations.

If values with long string or LOB data types are in the table, the values will not be inserted into the exception table in case of unique index violation.

Querying exception tables

The message column structure in an exception table is a concatenated list of constraint names, lengths and delimiters as described earlier. You may wish to write a query on this information.

For example, let's write a query to obtain a list of all the violations, repeating each row with only the constraint name along with it. Let us assume that our original table T1 had two columns C1 and C2. Assume also, that the corresponding exception table E1 has columns C1, C2 pertaining to those in T1 and MSGCOL as the message column. The following query (using recursion) will list one constraint name per row (repeating the row for more than one violation):

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
     CHAR(SUBSTR(MSGCOL, 12,
                 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
     1,
     15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
    FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, J+6,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
      I+1,
      J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
    FROM IV
    WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV;
```

If we want all the rows that violated a particular constraint, we could extend this query as follows:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, I, J) AS
 (SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, 12,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
      1,
      15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
    FROM E1
  UNION ALL
  SELECT C1, C2, MSGCOL,
      CHAR(SUBSTR(MSGCOL, J+6,
                 INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
      I+1,
      J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
    FROM IV
    WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTNAME = 'constraintname';
```

To obtain all the Check Constraint violations, one could execute the following:

```
WITH IV  (C1, C2, MSGCOL, CONSTNAME, CONSTTYPE, I, J) AS
  (SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, 12,
                   INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, 6, 1)),
       1,
       15+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,7,5)),5,0))
     FROM E1
   UNION ALL
   SELECT C1, C2, MSGCOL,
       CHAR(SUBSTR(MSGCOL, J+6,
                  INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0)))),
       CHAR(SUBSTR(MSGCOL, J, 1)),
       I+1,
       J+9+INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,J+1,5)),5,0))
     FROM IV
     WHERE I < INTEGER(DECIMAL(VARCHAR(SUBSTR(MSGCOL,1,5)),5,0))
 ) SELECT C1, C2, CONSTNAME FROM IV WHERE CONSTTYPE = 'K';
```

Appendix L. SQL statements allowed in routines

The following table indicates whether or not an SQL statement (specified in the first column) is allowed to execute in a routine with the specified SQL data access indication. If an executable SQL statement is encountered in a routine defined with NO SQL, SQLSTATE 38001 is returned. For other execution contexts, SQL statements that are not supported in any context return SQLSTATE 38003. For other SQL statements not allowed in a CONTAINS SQL context, SQLSTATE 38004 is returned. In a READS SQL DATA context, SQLSTATE 38002 is returned. During creation of an SQL routine, a statement that does not match the SQL data access indication will cause SQLSTATE 42985 to be returned.

If a statement invokes a routine, the effective SQL data access indication for the statement will be the greater of:

- The SQL data access indication of the statement from the following table.
- The SQL data access indication of the routine specified when the routine was created.

For example, the CALL statement has an SQL data access indication of CONTAINS SQL. However, if a stored procedure defined as READS SQL DATA is called, the effective SQL data access indication for the CALL statement is READS SQL DATA.

When a routine invokes an SQL statement, the effective SQL data access indication for the statement must not exceed the SQL data access indication declared for the routine. For example, a function defined as READS SQL DATA could not call a stored procedure defined as MODIFIES SQL DATA.

Table 179. SQL Statement and SQL Data Access Indication

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
ALTER...	N	N	N	Y
BEGIN DECLARE SECTION	Y(1)	Y	Y	Y
CALL	N	Y	Y	Y
CLOSE	N	N	Y	Y
COMMENT ON	N	N	N	Y
COMMIT	N	N(4)	N(4)	N(4)
COMPOUND SQL	N	Y	Y	Y
CONNECT(2)	N	N	N	N
CREATE	N	N	N	Y
DECLARE CURSOR	Y(1)	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE	N	N	N	Y
DELETE	N	N	N	Y
DESCRIBE	N	Y	Y	Y
DISCONNECT(2)	N	N	N	N
DROP ...	N	N	N	Y
END DECLARE SECTION	Y(1)	Y	Y	Y

Table 179. SQL Statement and SQL Data Access Indication (continued)

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
EXECUTE	N	Y(3)	Y(3)	Y
EXECUTE IMMEDIATE	N	Y(3)	Y(3)	Y
EXPLAIN	N	N	N	Y
FETCH	N	N	Y	Y
FREE LOCATOR	N	Y	Y	Y
FLUSH EVENT MONITOR	N	N	N	Y
GRANT ...	N	N	N	Y
INCLUDE	Y(1)	Y	Y	Y
INSERT	N	N	N	Y
LOCK TABLE	N	Y	Y	Y
OPEN	N	N	Y(5)	Y
PREPARE	N	Y	Y	Y
REFRESH TABLE	N	N	N	Y
RELEASE CONNECTION(2)	N	N	N	N
RELEASE SAVEPOINT	N	N	N	Y
RENAME TABLE	N	N	N	Y
REVOKE ...	N	N	N	Y
ROLLBACK	N	N(4)	N(4)	N(4)
ROLLBACK TO SAVEPOINT	N	N	N	Y
SAVEPOINT	N	N	N	Y
SELECT INTO	N	N	Y(5)	Y
SET CONNECTION(2)	N	N	N	N
SET INTEGRITY	N	N	N	Y
SET special register	N	Y	Y	Y
UPDATE	N	N	N	Y
VALUES INTO	N	N	Y	Y
WHENEVER	Y(1)	Y	Y	Y

Notes:

1. Although the NO SQL option implies that no SQL statements can be specified, non-executable statements are not restricted.
2. Connection management statements are not allowed in any routine execution context.
3. It depends on the statement being executed. The statement specified for the EXECUTE statement must be a statement that is allowed in the context of the particular SQL access level in effect. For example, if the SQL access level READS SQL DATA is in effect, the statement cannot be INSERT, UPDATE, or DELETE.
4. The COMMIT statement and the ROLLBACK statement without the TO SAVEPOINT clause can be used in a stored procedure, but only if the stored

procedure is called directly from an application, or indirectly through nested stored procedure calls from an application. (If any trigger, function, method, or atomic compound statement is in the call chain to the stored procedure, COMMIT or ROLLBACK of a unit of work is not allowed.)

5. If the SQL access level READS SQL DATA is in effect, no SQL data change statement can be embedded in the SELECT INTO statement or in the cursor referenced by the OPEN statement.

Appendix M. CALL invoked from a compiled statement

Invokes a procedure stored at the location of a database. A stored procedure, for example, executes at the location of the database, and returns data to the client application.

Programs using the SQL CALL statement are designed to run in two parts, one on the client and the other on the server. The server procedure at the database runs within the same transaction as the client application. If the client application and stored procedure are on the same partition, the stored procedure is executed locally.

Note: This form of the CALL statement is deprecated, and is only being provided for compatibility with previous versions of DB2.

Invocation:

This form of the CALL statement can only be embedded in an application program precompiled with the CALL_RESOLUTION DEFERRED option. It cannot be used in triggers, SQL procedures, or any other non-application contexts. It is an executable statement that cannot be dynamically prepared. However, the procedure name can be specified through a host variable and this, coupled with the use of the USING DESCRIPTOR clause, allows both the procedure name and the parameter list to be provided at run time, which achieves an effect similar to that of a dynamically prepared statement.

Authorization:

The privileges held by the authorization ID of the CALL statement *at run time* must include at least one of the following:

- EXECUTE privilege for the package associated with the stored procedure (EXECUTE privilege on the stored procedure is *not* checked.)
- CONTROL privilege for the package associated with the stored procedure
- SYSADM or DBADM authority

Syntax:

```
►►──CALL──┬─procedure-name─┬──┬──────────────────────────────────────────┬──────────►◄
          └─host-variable──┘  ├─(──┬───────────────────────┬──)──────────┤
                              │    │  ┌─,────────────────┐ │              │
                              │    └──▼─host-variable───┴─┘              │
                              └─USING DESCRIPTOR──descriptor-name────────┘
```

Description:

procedure-name **or** *host-variable*
: Identifies the procedure to call. The procedure name may be specified either directly or within a host variable. The procedure identified must exist at the current server (SQLSTATE 42724).

  If *procedure-name* is specified, it must be an ordinary identifier that is not greater than 254 bytes. Because this can only be an ordinary identifier, it cannot contain blanks or special characters. The value is converted to uppercase. If it is necessary to use lowercase names, blanks, or special characters, the name must be specified via a *host-variable*.

If *host-variable* is specified, it must be a character-string variable with a length attribute that is not greater than 254 bytes, and it must not include an indicator variable. The value is *not* converted to uppercase. The character string must be left-justified.

The procedure name can take one of several forms:

procedure-name
: The name (with no extension) of the procedure to execute. The procedure that is invoked is determined as follows.

1. The *procedure-name* is used to search the defined procedures (in SYSCAT.ROUTINES) for a matching procedure. A matching procedure is determined using the steps that follow.
   a. Find the procedures (ROUTINETYPE is 'P') from the catalog (SYSCAT.ROUTINES), where the ROUTINENAME matches the specified *procedure-name*, and the ROUTINESCHEMA is a schema name in the SQL path (CURRENT PATH special register). If the schema name is explicitly specified, the SQL path is ignored, and only procedures with the specified schema name are considered.
   b. Next, eliminate any of these procedures that do not have the same number of parameters as the number of arguments specified in the CALL statement.
   c. Chose the remaining procedure that is earliest in the SQL path.

   If a procedure is selected, DB2 will invoke the procedure defined by the external name.
2. If no matching procedure was found, *procedure-name* is used both as the name of the stored procedure library, and the function name within that library. For example, if *procedure-name* is `proclib`, the DB2 server will load the stored procedure library named `proclib` and execute the function routine `proclib()` within that library.

   In UNIX-based systems, the default directory for stored procedure libraries is `sqllib/function`. The default directory for unfenced stored procedures is `sqllib/function/unfenced`.

   In Windows-based systems, the default directory for stored procedure libraries is `sqllib\function`. The default directory for unfenced stored procedures is `sqllib\function\unfenced`.

If the library or function could not be found, an error is returned (SQLSTATE 42884).

procedure-library!function-name
: The exclamation character (!) acts as a delimiter between the library name and the function name of the stored procedure. For example, if `proclib!func` is specified, `proclib` is loaded into memory, and the function `func` from that library is executed. This allows multiple functions to be placed in the same stored procedure library.

The stored procedure library is located in the directories or specified in the LIBPATH variable, as described in *procedure-name*.

absolute-path!function-name
: The *absolute-path* specifies the complete path to the stored procedure library.

In a UNIX-based system, for example, if `/u/terry/proclib!func` is specified, the stored procedure library `proclib` is obtained from the directory `/u/terry`, and the function `func` from that library is executed.

In all of these cases, the total length of the procedure name, including its implicit or explicit full path, must not be longer than 254 bytes.

(host-variable,...)
Each specification of *host-variable* is a parameter of the CALL statement. The *n*th parameter of the CALL corresponds to the *n*th parameter of the server's stored procedure.

Each *host-variable* is assumed to be used for exchanging data in both directions between client and server. To avoid sending unnecessary data between client and server, the client application should provide an indicator variable with each parameter, and set the indicator to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the indicator variable to -128 for any parameter that is not used to return data to the client application.

If the server is DB2 Universal Database, the parameters must have matching data types in both the client and server program.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of host variables. The *n*th SQLVAR element corresponds to the *n*th parameter of the server's stored procedure.

Before the CALL statement is processed, the application must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement
- SQLVAR occurrences to indicate the attributes of the variables. The following fields of each Base SQLVAR element passed must be initialized:
  - SQLTYPE
  - SQLLEN
  - SQLDATA
  - SQLIND

  The following fields of each Secondary SQLVAR element passed must be initialized:
  - LEN.SQLLONGLEN
  - SQLDATALEN
  - SQLDATATYPE_NAME

The SQLDA is assumed to be used for exchanging data in both directions between client and server. To avoid sending unnecessary data between client and server, the client application should set the SQLIND field to -1 if the parameter is not used to transmit data to the stored procedure. The stored procedure should set the SQLIND field -128 for any parameter that is not used to return data to the client application.

Notes:

- ***Use of Large Object (LOB) data types***:

  If the client and server application needs to specify LOB data from an SQLDA, allocate double the number of SQLVAR entries.

  LOB data types have been supported by stored procedures since DB2 Version 2. The LOB data types are not supported by all down level clients or servers.
- ***Retrieving the RETURN_STATUS from an SQL procedure***:

  If an SQL procedure successfully issues a RETURN statement with a status value, this value is returned in the first SQLERRD field of the SQLCA. If the CALL statement is issued in an SQL procedure, use the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS value. The value is −1 if the SQLSTATE indicates an error.
- ***Returning result sets from stored procedures***:

  If the client application program is written using CLI, result sets can be returned directly to the client application. The stored procedure indicates that a result set is to be returned by declaring a cursor on that result set, opening a cursor on the result set, and leaving the cursor open when exiting the procedure.

  At the end of a procedure:
  - For every cursor that has been left open, a result set is returned to the application.
  - If more than one cursor is left open, the result sets are returned in the order in which their cursors were opened.
  - Only unread rows are passed back. For example, if the result set of a cursor has 500 rows, and 150 of those rows have been read by the stored procedure at the time the stored procedure is terminated, rows 151 through 500 will be returned to the stored procedure.
- ***Handling of special registers***:

  The settings of special registers for the caller are inherited by the stored procedure on invocation, and restored upon return to the caller. Special registers may be changed within a stored procedure, but these changes do not affect the caller. This is not true for legacy stored procedures (those defined with parameter style DB2DARI, or found in the default library), where the changes made to special registers in a procedure become the settings for the caller.
- ***Compatibilities***:

  There is a newer, preferred, form of the CALL statement that can be embedded in an application (by precompiling the application with the CALL_RESOLUTION IMMEDIATE option), or that can be dynamically prepared.

Examples:

Example 1:

In C, invoke a procedure called TEAMWINS in the ACHIEVE library, passing it a parameter stored in the host variable HV_ARGUMENT.

```
strcpy(HV_PROCNAME, "ACHIEVE!TEAMWINS");
CALL :HV_PROCNAME (:HV_ARGUMENT);
```

Example 2:

In C, invoke a procedure called :SALARY_PROC, using the SQLDA named INOUT_SQLDA.

```
struct sqlda *INOUT_SQLDA;
/* Setup code for SQLDA variables goes here */
CALL :SALARY_PROC
USING DESCRIPTOR :*INOUT_SQLDA;
```

Example 3:

A Java stored procedure is defined in the database, using the following statement:

```
CREATE PROCEDURE PARTS_ON_HAND (IN PARTNUM INTEGER,
                                OUT COST DECIMAL(7,2),
                                OUT QUANTITY INTEGER)
   EXTERNAL NAME 'parts!onhand'
   LANGUAGE JAVA
   PARAMETER STYLE DB2GENERAL;
```

A Java application calls this stored procedure using the following code fragment:

```
...
CallableStatement stpCall;

String sql = "CALL PARTS_ON_HAND (?,?,?)";

stpCall = con.prepareCall( sql ) ; /* con is the connection */

stpCall.setInt( 1, variable1 ) ;
stpCall.setBigDecimal( 2, variable2 ) ;
stpCall.setInt( 3, variable3 ) ;

stpCall.registerOutParameter( 2, Types.DECIMAL, 2 ) ;
stpCall.registerOutParameter( 3, Types.INTEGER ) ;

stpCall.execute() ;

variable2 = stpCall.getBigDecimal(2) ;
variable3 = stpCall.getInt(3) ;
...
```

This application code fragment will invoke the Java method *onhand* in class *parts*, because the procedure name specified on the CALL statement is found in the database and has the external name 'parts!onhand'.

Related reference:

- "CALL statement" in the *SQL Reference, Volume 2*

Appendix N. Japanese and traditional-Chinese extended UNIX code (EUC) considerations

Extended UNIX Code (EUC) for Japanese and Traditional-Chinese defines a set of encoding rules that can support from 1 to 4 character sets. In some cases, such as Japanese EUC (eucJP) and Traditional-Chinese EUC (eucTW), a character may be encoded using more than two bytes. Use of such an encoding scheme has implications when used as the code page of the database server or the database client. The key considerations involve the following:

- Expansion or contraction of strings when converting between EUC code pages and double-byte code pages
- Use of Universal Character Set-2 (UCS-2) as the code page for graphic data stored in a database server defined with the eucJP (Japanese) or eucTW (Traditional-Chinese) code pages.

With the exception of these considerations, the use of EUC is consistent with the double-byte character set (DBCS) support. Throughout this book (and others), references to *double-byte* have been changed to *multi-byte* to reflect support for encoding rules that allow for character representations that require more than 2 bytes. Detailed considerations for support of Japanese and Traditional-Chinese EUC are included here. This information should be considered by anyone using SQL with an EUC database server or an EUC database client, and used in conjunction with application development information.

Language elements

Characters

Each multi-byte character is considered a *letter* with the exception of the double-byte blank character which is considered a *special character*.

Tokens

Multi-byte lowercase alphabetic letters are not folded to uppercase. This differs from the single byte lowercase alphabetic letters in tokens which are generally folded to uppercase.

Identifiers

SQL identifiers

Conversion between a double-byte code page and an EUC code page may result in the conversion of double-byte characters to multi-byte characters encoded with more than 2 bytes. As a result, an identifier that fits the length maximum in the double-byte code page may exceed the length in the EUC code page. Selecting identifiers for this type of environment must be done carefully to avoid expansion beyond the maximum identifier length.

Data types

Character strings

In an MBCS database, character strings may contain a mixture of characters from a single-byte character set (SBCS) and from multi-byte character sets (MBCS). When using such strings, operations may provide different results if they are character based (treat the data as characters) or byte based (treat the data as bytes). Check the function or operation description to determine how mixed strings are processed.

Graphic strings

A graphic string is defined as a sequence of double-byte character data. In order to allow Japanese or Traditional-Chinese EUC data to be stored in graphic columns, EUC characters are encoded in UCS-2. Characters that are not double-byte characters under all supported encoding schemes (for example, PC or EBCDIC DBCS) should not be used with graphic columns. The results of using other than double-byte characters may result in replacement by substitution characters during conversion. Retrieval of such data will not return the same value as was entered.

Assignments and comparisons

String assignments: Conversion of a string is performed prior to the assignment. In cases involving an eucJP/eucTW code page and a DBCS code page, a character string may become longer (DBCS to eucJP/eucTW) or shorter (eucJP/eucTW to DBCS). This may result in errors on storage assignment and truncation on retrieval assignment. When the error on storage assignment is due to expansion during conversion, SQLSTATE 22524 is returned instead of SQLSTATE 22001.

Similarly, assignments involving graphic strings may result in the conversion of a UCS-2 encoded double-byte character to a substitution character in a PC or EBCDIC DBCS code page for characters that do not have a corresponding double-byte character. Assignments that replace characters with substitution characters will indicate this by setting the SQLWARN10 field of the SQLCA to 'W'.

In cases of truncation during retrieval assignment involving multi-byte character strings, the point of truncation may be part of a multi-byte character. In this case, each byte of the character fragment is replaced with a single-byte blank. This means that more than one single-byte blank may appear at the end of a truncated character string.

String comparisons: String comparisons are performed on a byte basis. Character strings also use the collating sequence defined for the database. Graphic strings do not use the collating sequence and, in an eucJP or eucTW database, are encoded using UCS-2. Thus, the comparison of two mixed character strings may have a different result from the comparison of two graphic strings even though they contain the same characters. Similarly, the resulting sort order of a mixed character column and a graphic column may be different.

Rules for result data types

The resulting data type for character strings is not affected by the possible expansion of the string. For example, a union of two CHAR operands will still be a CHAR. However, if one of the character string operands will be converted such that the maximum expansion makes the length attribute the largest of the two operands, then the resulting character string length attribute is affected. For example, consider the result expressions of a CASE expression that have data types of VARCHAR(100) and VARCHAR(120). Assume the VARCHAR(100) expression is a mixed string host variable (that may require conversion) and the VARCHAR(120) expression is a column in the eucJP database. The resulting data type is VARCHAR(200) since the VARCHAR(100) is doubled to allow for possible conversion. The same scenario without the involvement of an eucJP or eucTW database would have a result type of VARCHAR(120).

Notice that the doubling of the host variable length is based on the fact that the database server is Japanese EUC or Traditional-Chinese EUC. Even if the client is also eucJP or eucTW, the doubling is still applied. This allows the same application package to be used by double-byte or multi-byte clients.

Rules for string conversions

The types of operations listed in the corresponding section of the SQL Reference may convert operands to either the application or the database code page.

If such operations are done in a mixed code page environment that includes Japanese or Traditional-Chinese EUC, expansion or contraction of mixed character string operands can occur. Therefore, the resulting data type has a length attribute that accommodates the maximum expansion, if possible. In cases where there are restrictions on the length attribute of the data type, the maximum allowed length for the data type is used. For example in an environment where maximum growth is double, a VARCHAR(200) host variable is treated as if it is a VARCHAR(400), but CHAR(200) host variable is treated as if it is a CHAR(254). A run-time error may occur when conversion is performed if the converted string would exceed the maximum length for the data type. For example, the union of CHAR(200) and CHAR(10) would have a result type of CHAR(254). When the value from the left side of the UNION is converted, if more than 254 characters are required, an error occurs.

In some cases, allowing for the maximum growth for conversion will cause the length attribute to exceed a limit. For example, UNION only allows columns up to 254 bytes. Thus, a query with a union that included a host variable in the column list (call it :hv1) that was a DBCS mixed character string defined as a varying length character string 128 bytes long, would set the data type to VARCHAR(256) resulting in an error preparing the query, even though the query in the application does not appear to have any columns greater than 254. In a situation where the actual string is not likely to cause expansion beyond 254 bytes, the following can be used to prepare the statement.

```
SELECT CAST(:hv1 CONCAT ' AS VARCHAR(254)), C2 FROM T1
UNION
SELECT C1, C2 FROM T2
```

The concatenation of the null string with the host variable will force the conversion to occur before the cast is done. This query can be prepared in the DBCS to eucJP/eucTW environment although a truncation error may occur at run time.

This technique (null string concat with cast) can be used to handle the similar 254-byte limit for SELECT DISTINCT or use of the column in ORDER BY or GROUP BY clauses.

Constants

Graphic string constants

Japanese or Traditional-Chinese EUC client, may contain single or multi-byte characters (like a mixed character string). The string should not contain more than 2 000 characters. It is recommended that only characters that convert to double-byte characters in all related PC and EBCDIC double-byte code pages be used in graphic constants. A graphic string constant in an SQL statement is converted from the client code page to the double-byte encoding at the database server. For a Japanese or Traditional-Chinese EUC server, the constant is converted to UCS-2, the double-byte encoding used for graphic strings. For a double-byte server, the constant is converted from the client code page to the DBCS code page of the server.

Functions

The design of user-defined functions should consider the impact of supporting Japanese or Tradition-Chinese EUC on the parameter data types. One part of function resolution considers the data types of the arguments to a function call. Mixed character string arguments involving a Japanese or Traditional-Chinese EUC client may require additional bytes to specify the argument. This may require that the data type change to allow the increased length. For example, it may take 4001 bytes to represent a character string in the application (a LONG VARCHAR) that fits into a VARCHAR(4000) string at the server. If a function signature is not included that allows the argument to be a LONG VARCHAR, function resolution will fail to find a function.

Some functions exist that do not allow long strings for various reasons. Use of LONG VARCHAR or CLOB arguments with such functions will not succeed. For example, LONG VARCHAR as the second argument of the built-in POSSTR function, will fail function resolution (SQLSTATE 42884).

Expressions

With the concatenation operator

The potential expansion of one of the operands of concatenation may cause the data type and length of concatenated operands to change when in an environment that includes a Japanese or Traditional-Chinese EUC database server. For example, with an EUC server where the value from a host variable may double in length, consider the following example.

```
CHAR200 CONCAT :char50
```

The column *CHAR200* is of type CHAR(200). The host variable *char50* is defined as CHAR(50). The result type for this concatenation operation would normally be CHAR(250). However, given an eucJP or eucTW database server, the assumption is that the string may expand to double the length. Hence *char50* is treated as a CHAR(100) and the resulting data type is VARCHAR(300). Note that even though the result is a VARCHAR, it will always have 300 bytes of data including trailing blanks. If the extra trailing blanks are not desired, define the host variable as VARCHAR(50) instead of CHAR(50).

Predicates

LIKE predicate

For a LIKE predicate involving mixed character strings in an EUC database:

- An SBCS halfwidth underscore character refers to one SBCS character.
- A non-SBCS fullwidth underscore character refers to one non-SBCS character.
- An SBCS halfwidth or non-SBCS fullwidth percent sign character refers to zero or more SBCS or non-SBCS characters.

The escape character must be one SBCS or non-SBCS character. In a character column, the escape character can also be a binary string containing exactly one byte.

Note that use of the underscore character may produce different results, depending on the code page of the LIKE operation. For example, Katakana characters in

Japanese EUC are multi-byte characters (CS2) but in the Japanese DBCS code page they are single-byte characters. A query with the single-byte underscore in the *pattern-expression* would return occurrences of Katakana character in the position of the underscore from a Japanese DBCS server. However, the same rows from the equivalent table in a Japanese EUC server would not be returned, since the Katakana characters will only match with a double-byte underscore.

For a LIKE predicate involving graphic strings in an EUC database:
- A fullwidth underscore character (U+FF3F) refers to one Unicode character.
- A fullwidth percent sign character (U+FF05) refers to zero or more Unicode characters.

Functions

LENGTH

The processing of this function is no different for mixed character strings in an EUC environment. The value returned is the length of the string in the code page of the argument. As of Version 8, if the argument is a host variable, the value returned is the length of the string in the database code page. When using this function to determine the length of a value, careful consideration should be given to how the length is used. This is especially true for mixed string constants since the length is given in bytes, not characters. For example, the length of a mixed string column in a DBCS database returned by the LENGTH function may be less than the length of the retrieved value of that column on an eucJP or eucTW client due to the conversion of some DBCS characters to multi-byte eucJP or eucTW characters.

SUBSTR

The SUBSTR function operates on mixed character strings on a byte basis. The resulting string may therefore include fragments of multi-byte characters at the beginning or end of the resulting string. No processing is provided to detect or process fragments of characters.

TRANSLATE

The TRANSLATE function supports mixed character strings including multi-byte characters. The corresponding characters of the *to-string-exp* and the *from-string-exp* must have the same number of bytes and cannot end with part of a multi-byte character.

The *pad-char-exp* must result in a single-byte character when the *char-string-exp* is a character string. Since TRANSLATE is performed in the code page of the *char-string-exp*, the *pad-char-exp* may be converted from a multi-byte character to a single-byte character.

A *char-string-exp* that ends with part of a multi-byte character will not have those bytes translated.

VARGRAPHIC

The VARGRAPHIC function on a character string operand in a Japanese or Traditional-Chinese EUC code page returns a graphic string in the UCS-2 code page.
- Single-byte characters are converted first to their corresponding double-byte character in the code set to which they belong (eucJP or eucTW). Then they are

converted to the corresponding UCS-2 representation. If there is no double-byte representation, the character is converted to the double-byte substitution character defined for that code set before being converted to UCS-2 representation.

- Characters from eucJP that are Katakana (eucJP CS2) are actually single byte characters in some encoding schemes. They are thus converted to corresponding double-byte characters in eucJP or to the double-byte substitution character before converting to UCS-2.
- Multi-byte characters are converted to their UCS-2 representations.

Statements

CONNECT

The processing of a successful CONNECT statement returns information in the SQLCA that is important when the possibility exists for applications to process data in an environment that includes a Japanese or Traditional-Chinese EUC code page at the client or server. The *SQLERRD(1)* field gives the maximum expansion of a mixed character string when converted from the application code page to the database code page. The *SQLERRD(2)* field gives the maximum expansion of a mixed character string when converted from the database code page to the application code page. The value is positive if expansion could occur and negative if contraction could occur. If the value is negative, the value is always -1 since the worst case is that no contraction occurs and the full length of the string is required after conversion. Positive values may be as large as 2, meaning that in the worst case, double the string length may be required for the character string after conversion.

The code page of the application server and the application client are also available in the SQLERRMC field of the SQLCA.

PREPARE

The data types determined for untyped parameter markers are not changed in an environment that includes Japanese or Traditional-Chinese EUC. As a result, it may be necessary in some cases to use typed parameter markers to provide sufficient length for mixed character strings in eucJP or eucTW. For example, consider an insert to a CHAR(10) column. Preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (?)
```

would result in a data type of CHAR(10) for the parameter marker. If the client was eucJP or eucTW, more than 10 bytes may be required to represent the string to be inserted but the same string in the DBCS code page of the database is not more than 10 bytes. In this case, the statement to prepare should include a typed parameter marker with a length greater than 10. Thus, preparing the statement:

```
INSERT INTO T1 (CH10) VALUES (CAST(? AS VARCHAR(20))
```

would result in a data type of VARCHAR(20) for the parameter marker.

Related reference:

- “PREPARE statement” in the *SQL Reference, Volume 2*

Appendix O. Backus-Naur form (BNF) specifications for DATALINKs

A DATALINK value is an encapsulated value that contains a logical reference from the database to a file stored outside the database.

The data location attribute of this encapsulated value is a logical reference to a file in the form of a uniform resource locator (URL). The value of this attribute conforms to the syntax for URLs as specified by the following BNF, based on RFC 1738 : Uniform Resource Locators (URL), T. Berners-Lee, L. Masinter, M. McCahill, December 1994. (BNF is an acronym for ″Backus-Naur Form″, a formal notation to describe the syntax of a given language.)

The following conventions are used in the BNF specification:
- "|" is used to designate alternatives
- brackets [ ] are used around optional or repeated elements
- literals are quoted with ""
- elements may be preceded with [*n*]* to designate *n* or more repetitions of the following element; if *n* is not specified, the default is 0

The BNF specification for DATALINKs:

URL

```
url          =    httpurl | fileurl | uncurl | emptyurl
```

HTTP

```
httpurl      =    "http://" hostport [ "/" hpath ]
hpath        =    hsegment *[ "/" hsegment ]
hsegment     =    *[ uchar | ";" | ":" | "@" | "&" | "=" ]
```

Note that the search element from the original BNF in RFC1738 has been removed, because it is not an essential part of the file reference and does not make sense in DATALINKs context.

FILE

```
fileurl      =    "file://" host "/" fpath
fpath        =    fsegment *[ "/" fsegment ]
fsegment     =    *[ uchar | "?" | ":" | "@" | "&" | "=" ]
```

Note that `host` is not optional and the ″`localhost`″ string does not have any special meaning, in contrast with RFC1738. This avoids confusing interpretations of ″`localhost`″ in client/server and partitioned database configurations.

UNC

```
uncurl       =    "unc:\\" hostname "\" sharename "\" uncpath
sharename    =    *uchar
uncpath      =    fsegment *[ "\" fsegment ]
```

Supports the commonly used UNC naming convention on Windows. This is not a standard scheme in RFC1738.

EMPTYURL

```
emptyurl     =    ""
hostport     =    host [ ":" port ]
host         =    hostname | hostnumber
```

```
hostname     =    *[ domainlabel "." ] toplabel
domainlabel  =    alphadigit | alphadigit *[ alphadigit | "-" ] alphadigit
toplabel     =    alpha | alpha *[ alphadigit | "-" ] alphadigit
alphadigit   =    alpha | digit
hostnumber   =    digits "." digits "." digits "." digits
port         =    digits
```

Empty (zero-length) URLs are also supported for DATALINK values. These are useful to update DATALINK columns when reconcile exceptions are reported and non-nullable DATALINK columns are involved. A zero-length URL is used to update the column and cause a file to be unlinked.

Miscellaneous Definitions

```
lowalpha      =   "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
                  "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
                  "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
                  "y" | "z"
hialpha       =   "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
                  "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                  "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
                  "Y" | "Z"
alpha         =   lowalpha | hialpha
digit         =   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                  "8" | "9"
safe          =   "$" | "-" | "_" | "." | "+"
extra         =   "!" | "*" | "'" | "(" | ")" | ","
hex           =   digit | "A" | "B" | "C" | "D" | "E" | "F" |
                  "a" | "b" | "c" | "d" | "e" | "f"
escape        =   "%" hex hex
unreserved    =   alpha | digit | safe | extra
uchar         =   unreserved | escape
digits        =   1*digit
```

Leading and trailing blank characters are trimmed by DB2 while parsing. Also, the scheme names ('HTTP', 'FILE', 'UNC') and `host` are case-insensitive, and are always stored in the database in uppercase.

Appendix P. DB2 Universal Database technical information

DB2 documentation and help

DB2® technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- Downloadable PDF files, PDF files on CD, and printed books
  - Guides
  - Reference manuals
- Command line help
  - Command help
  - Message help
  - SQL state help
- Installed source code
  - Sample programs

You can access additional DB2 Universal Database™ technical information such as technotes, white papers, and Redbooks™ online at ibm.com®. Access the DB2 Information Management software library site at www.ibm.com/software/data/pubs/.

DB2 documentation updates

IBM® may periodically make documentation FixPaks and other documentation updates to the DB2 Information Center available. If you access the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/, you will always be viewing the most up-to-date information. If you have installed the DB2 Information Center locally, then you need to install any updates manually before you can view them. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* when new information becomes available.

The Information Center is updated more frequently than either the PDF or the hardcopy books. To get the most current DB2 technical information, install the documentation updates as they become available or go to the DB2 Information Center at the www.ibm.com site.

Related concepts:

- "CLI sample programs" in the *CLI Guide and Reference, Volume 1*
- "Java sample programs" in the *Application Development Guide: Building and Running Applications*
- "DB2 Information Center" on page 764

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 781

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Invoking message help from the command line processor" on page 782
- "Invoking command help from the command line processor" on page 783
- "Invoking SQL state help from the command line processor" on page 783

Related reference:

- "DB2 PDF and printed documentation" on page 775

DB2 Information Center

The DB2® Information Center gives you access to all of the information you need to take full advantage of DB2 family products, including DB2 Universal Database™, DB2 Connect™, DB2 Information Integrator and DB2 Query Patroller™. The DB2 Information Center also contains information for major DB2 features and components including replication, data warehousing, and the DB2 extenders.

The DB2 Information Center has the following features if you view it in Mozilla 1.0 or later or Microsoft® Internet Explorer 5.5 or later. Some features require you to enable support for JavaScript™:

Flexible installation options

You can choose to view the DB2 documentation using the option that best meets your needs:

- To effortlessly ensure that your documentation is always up to date, you can access all of your documentation directly from the DB2 Information Center hosted on the IBM® Web site at http://publib.boulder.ibm.com/infocenter/db2help/
- To minimize your update efforts and keep your network traffic within your intranet, you can install the DB2 documentation on a single server on your intranet
- To maximize your flexibility and reduce your dependence on network connections, you can install the DB2 documentation on your own computer

Search

You can search all of the topics in the DB2 Information Center by entering a search term in the **Search** text field. You can retrieve exact matches by enclosing terms in quotation marks, and you can refine your search with wildcard operators (*, ?) and Boolean operators (AND, NOT, OR).

Task-oriented table of contents

You can locate topics in the DB2 documentation from a single table of contents. The table of contents is organized primarily by the kind of tasks you may want to perform, but also includes entries for product overviews, goals, reference information, an index, and a glossary.

- Product overviews describe the relationship between the available products in the DB2 family, the features offered by each of those products, and up to date release information for each of these products.
- Goal categories such as installing, administering, and developing include topics that enable you to quickly complete tasks and develop a deeper understanding of the background information for completing those tasks.

- Reference topics provide detailed information about a subject, including statement and command syntax, message help, and configuration parameters.

Show current topic in table of contents
You can show where the current topic fits into the table of contents by clicking the **Refresh / Show Current Topic** button in the table of contents frame or by clicking the **Show in Table of Contents** button in the content frame. This feature is helpful if you have followed several links to related topics in several files or arrived at a topic from search results.

Index You can access all of the documentation from the index. The index is organized in alphabetical order by index term.

Glossary
You can use the glossary to look up definitions of terms used in the DB2 documentation. The glossary is organized in alphabetical order by glossary term.

Integrated localized information
The DB2 Information Center displays information in the preferred language set in your browser preferences. If a topic is not available in your preferred language, the DB2 Information Center displays the English version of that topic.

For iSeries™ technical information, refer to the IBM eServer™ iSeries information center at www.ibm.com/eserver/iseries/infocenter/.

Related concepts:
- "DB2 Information Center installation scenarios" on page 765

Related tasks:
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Displaying topics in your preferred language in the DB2 Information Center" on page 774
- "Invoking the DB2 Information Center" on page 772
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 768
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 770

DB2 Information Center installation scenarios

Different working environments can pose different requirements for how to access DB2® information. The DB2 Information Center can be accessed on the IBM® Web site, on a server on your organization's network, or on a version installed on your computer. In all three cases, the documentation is contained in the DB2 Information Center, which is an architected web of topic-based information that you view with a browser. By default, DB2 products access the DB2 Information Center on the IBM Web site. However, if you want to access the DB2 Information Center on an intranet server or on your own computer, you must install the DB2 Information Center using the DB2 Information Center CD found in your product Media Pack. Refer to the summary of options for accessing DB2 documentation which follows, along with the three installation scenarios, to help determine which method of accessing the DB2 Information Center works best for you and your work environment, and what installation issues you might need to consider.

Summary of options for accessing DB2 documentation:

The following table provides recommendations on which options are possible in your work environment for accessing the DB2 product documentation in the DB2 Information Center.

Internet access	**Intranet access**	**Recommendation**
Yes	Yes	Access the DB2 Information Center on the IBM Web site, or access the DB2 Information Center installed on an intranet server.
Yes	No	Access the DB2 Information Center on the IBM Web site.
No	Yes	Access the DB2 Information Center installed on an intranet server.
No	No	Access the DB2 Information Center on a local computer.

Scenario: Accessing the DB2 Information Center on your computer:

Tsu-Chen owns a factory in a small town that does not have a local ISP to provide him with Internet access. He purchased DB2 Universal Database™ to manage his inventory, his product orders, his banking account information, and his business expenses. Never having used a DB2 product before, Tsu-Chen needs to learn how to do so from the DB2 product documentation.

After installing DB2 Universal Database on his computer using the typical installation option, Tsu-Chen tries to access the DB2 documentation. However, his browser gives him an error message that the page he tried to open cannot be found. Tsu-Chen checks the installation manual for his DB2 product and discovers that he has to install the DB2 Information Center if he wants to access DB2 documentation on his computer. He finds the *DB2 Information Center CD* in the media pack and installs it.

From the application launcher for his operating system, Tsu-Chen now has access to the DB2 Information Center and can learn how to use his DB2 product to increase the success of his business.

Scenario: Accessing the DB2 Information Center on the IBM Web site:

Colin is an information technology consultant with a training firm. He specializes in database technology and SQL and gives seminars on these subjects to businesses all over North America using DB2 Universal Database. Part of Colin's seminars includes using DB2 documentation as a teaching tool. For example, while teaching courses on SQL, Colin uses the DB2 documentation on SQL as a way to teach basic and advanced syntax for database queries.

Most of the businesses at which Colin teaches have Internet access. This situation influenced Colin's decision to configure his mobile computer to access the DB2 Information Center on the IBM Web site when he installed the latest version of DB2 Universal Database. This configuration allows Colin to have online access to the latest DB2 documentation during his seminars.

However, sometimes while travelling Colin does not have Internet access. This posed a problem for him, especially when he needed to access to DB2 documentation to prepare for seminars. To avoid situations like this, Colin installed a copy of the DB2 Information Center on his mobile computer.

Colin enjoys the flexibility of always having a copy of DB2 documentation at his disposal. Using the **db2set** command, he can easily configure the registry variables on his mobile computer to access the DB2 Information Center on either the IBM Web site, or his mobile computer, depending on his situation.

Scenario: Accessing the DB2 Information Center on an intranet server:

Eva works as a senior database administrator for a life insurance company. Her administration responsibilities include installing and configuring the latest version of DB2 Universal Database on the company's UNIX® database servers. Her company recently informed its employees that, for security reasons, it would not provide them with Internet access at work. Because her company has a networked environment, Eva decides to install a copy of the DB2 Information Center on an intranet server so that all employees in the company who use the company's data warehouse on a regular basis (sales representatives, sales managers, and business analysts) have access to DB2 documentation.

Eva instructs her database team to install the latest version of DB2 Universal Database on all of the employee's computers using a response file, to ensure that each computer is configured to access the DB2 Information Center using the host name and the port number of the intranet server.

However, through a misunderstanding Migual, a junior database administrator on Eva's team, installs a copy of the DB2 Information Center on several of the employee computers, rather than configuring DB2 Universal Database to access the DB2 Information Center on the intranet server. To correct this situation Eva tells Migual to use the **db2set** command to change the DB2 Information Center registry variables (DB2_DOCHOST for the host name, and DB2_DOCPORT for the port number) on each of these computers. Now all of the appropriate computers on the network have access to the DB2 Information Center, and employees can find answers to their DB2 questions in the DB2 documentation.

Related concepts:

- "DB2 Information Center" on page 764

Related tasks:

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 768
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 770
- "Setting the location for accessing the DB2 Information Center: Common GUI help"

Related reference:

- "db2set - DB2 Profile Registry Command" in the *Command Reference*

Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a UNIX operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on UNIX computers.

- **Hardware requirements**

  You require one of the following processors:
  - PowerPC (AIX)
  - HP 9000 (HP-UX)
  - Intel 32–bit (Linux)
  - Solaris UltraSPARC computers (Solaris Operating Environment)
- **Operating system requirements**

  You require one of the following operating systems:
  - IBM AIX 5.1 (on PowerPC)
  - HP-UX 11i (on HP 9000)
  - Red Hat Linux 8.0 (on Intel 32–bit)
  - SuSE Linux 8.1 (on Intel 32–bit)
  - Sun Solaris Version 8 (on Solaris Operating Environment UltraSPARC computers)

  Note: The DB2 Information Center runs on a subset of the UNIX operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center from the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**
  - The following browser is supported:
    - Mozilla Version 1.0 or greater
- The DB2 Setup wizard is a graphical installer. You must have an implementation of the X Window System software capable of rendering a graphical user interface for the DB2 Setup wizard to run on your computer. Before you can run the DB2 Setup wizard you must ensure that you have properly exported your display. For example, enter the following command at the command prompt:

  ```
  export DISPLAY=9.26.163.144:0.
  ```
- **Communication requirements**
  - TCP/IP

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system.
2. Insert and mount the DB2 Information Center product CD on your system.
3. Change to the directory where the CD is mounted by entering the following command:
   ```
   cd /cd
   ```
   where */cd* represents the mount point of the CD.
4. Enter the **./db2setup** command to start the DB2 Setup wizard.
5. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
6. On the **Select the product you would like to install** page, click **Next**.
7. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
8. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
9. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
10. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
11. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
12. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can also install the DB2 Information Center using a response file.

The installation logs db2setup.his, db2setup.log, and db2setup.err are located, by default, in the /tmp directory.

The db2setup.log file captures all DB2 product installation information, including errors. The db2setup.his file records all DB2 product installations on your computer. DB2 appends the db2setup.log file to the db2setup.his file. The db2setup.err file captures any error output that is returned by Java, for example, exceptions and trap information.

When the installation is complete, the DB2 Information Center will be installed in one of the following directories, depending upon your UNIX operating system:
- AIX: /usr/opt/db2_08_01
- HP-UX: /opt/IBM/db2/V8.1
- Linux: /opt/IBM/db2/V8.1
- Solaris Operating Environment: /opt/IBM/db2/V8.1

Related concepts:
- “DB2 Information Center” on page 764
- “DB2 Information Center installation scenarios” on page 765

Related tasks:

- "Installing DB2 using a response file (UNIX)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Displaying topics in your preferred language in the DB2 Information Center" on page 774
- "Invoking the DB2 Information Center" on page 772
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 770

Installing the DB2 Information Center using the DB2 Setup wizard (Windows)

DB2 product documentation can be accessed in three ways: on the IBM Web site, on an intranet server, or on a version installed on your computer. By default, DB2 products access DB2 documentation on the IBM Web site. If you want to access the DB2 documentation on an intranet server or on your own computer, you must install the DB2 documentation from the *DB2 Information Center CD*. Using the DB2 Setup wizard, you can define your installation preferences and install the DB2 Information Center on a computer that uses a Windows operating system.

Prerequisites:

This section lists the hardware, operating system, software, and communication requirements for installing the DB2 Information Center on Windows.

- **Hardware requirements**

  You require one of the following processors:
  - 32-bit computers: a Pentium or Pentium compatible CPU
- **Operating system requirements**

  You require one of the following operating systems:
  - Windows 2000
  - Windows XP

  Note: The DB2 Information Center runs on a subset of the Windows operating systems on which DB2 clients are supported. It is therefore recommended that you either access the DB2 Information Center on the IBM Web site, or that you install and access the DB2 Information Center on an intranet server.
- **Software requirements**
  - The following browsers are supported:
    - Mozilla 1.0 or greater
    - Internet Explorer Version 5.5 or 6.0 (Version 6.0 for Windows XP)
- **Communication requirements**
  - TCP/IP

Restrictions:

- You require an account with administrative privileges to install the DB2 Information Center.

Procedure:

To install the DB2 Information Center using the DB2 Setup wizard:

1. Log on to the system with the account that you have defined for the DB2 Information Center installation.
2. Insert the CD into the drive. If enabled, the auto-run feature starts the IBM DB2 Setup Launchpad.
3. The DB2 Setup wizard determines the system language and launches the setup program for that language. If you want to run the setup program in a language other than English, or the setup program fails to auto-start, you can start the DB2 Setup wizard manually.

   To start the DB2 Setup wizard manually:

   a. Click **Start** and select **Run**.
   b. In the **Open** field, type the following command:

      ```
      x:\setup.exe /i 2-letter language identifier
      ```

      where *x:* represents your CD drive, and *2-letter language identifier* represents the language in which the setup program will be run.
   c. Click **OK**.
4. The IBM DB2 Setup Launchpad opens. To proceed directly to the installation of the DB2 Information Center, click **Install Product**. Online help is available to guide you through the remaining steps. To invoke the online help, click **Help**. You can click **Cancel** at any time to end the installation.
5. On the **Select the product you would like to install** page, click **Next**.
6. Click **Next** on the **Welcome to the DB2 Setup wizard** page. The DB2 Setup wizard will guide you through the program setup process.
7. To proceed with the installation, you must accept the license agreement. On the **License Agreement** page, select **I accept the terms in the license agreement** and click **Next**.
8. Select **Install DB2 Information Center on this computer** on the **Select the installation action** page. If you want to use a response file to install the DB2 Information Center on this or other computers at a later time, select **Save your settings in a response file**. Click **Next**.
9. Select the languages in which the DB2 Information Center will be installed on **Select the languages to install** page. Click **Next**.
10. Configure the DB2 Information Center for incoming communication on the **Specify the DB2 Information Center port** page. Click **Next** to continue the installation.
11. Review the installation choices you have made in the **Start copying files** page. To change any settings, click **Back**. Click **Install** to copy the DB2 Information Center files onto your computer.

You can install the DB2 Information Center using a response file. You can also use the **db2rspgn** command to generate a response file based on an existing installation.

For information on errors encountered during installation, see the db2.log and db2wi.log files located in the 'My Documents'\DB2LOG\ directory. The location of the 'My Documents' directory will depend on the settings on your computer.

The db2wi.log file captures the most recent DB2 installation information. The db2.log captures the history of DB2 product installations.

Related concepts:

- "DB2 Information Center" on page 764
- "DB2 Information Center installation scenarios" on page 765

Related tasks:

- "Installing a DB2 product using a response file (Windows)" in the *Installation and Configuration Supplement*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Displaying topics in your preferred language in the DB2 Information Center" on page 774
- "Invoking the DB2 Information Center" on page 772
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 768

Related reference:

- "db2rspgn - Response File Generator Command (Windows)" in the *Command Reference*

Invoking the DB2 Information Center

The DB2 Information Center gives you access to all of the information that you need to use DB2 products for Linux, UNIX, and Windows operating systems such as DB2 Universal Database, DB2 Connect, DB2 Information Integrator, and DB2 Query Patroller.

You can invoke the DB2 Information Center from one of the following places:

- Computers on which a DB2 UDB client or server is installed
- An intranet server or local computer on which the DB2 Information Center installed
- The IBM Web site

Prerequisites:

Before you invoke the DB2 Information Center:

- *Optional*: Configure your browser to display topics in your preferred language
- *Optional*: Configure your DB2 client to use the DB2 Information Center installed on your computer or intranet server

Procedure:

To invoke the DB2 Information Center on a computer on which a DB2 UDB client or server is installed:

- From the Start Menu (Windows operating system): Click **Start** ⟶ **Programs** ⟶ **IBM DB2** ⟶ **Information** ⟶ **Information Center**.
- From the command line prompt:
  - For Linux and UNIX operating systems, issue the **db2icdocs** command.
  - For the Windows operating system, issue the **db2icdocs.exe** command.

To open the DB2 Information Center installed on an intranet server or local computer in a Web browser:

- Open the Web page at http://<host-name>:<port-number>/, where <host-name> represents the host name and <port-number> represents the port number on which the DB2 Information Center is available.

To open the DB2 Information Center on the IBM Web site in a Web browser:
- Open the Web page at publib.boulder.ibm.com/infocenter/db2help/.

Related concepts:
- "DB2 Information Center" on page 764
- "DB2 Information Center installation scenarios" on page 765

Related tasks:
- "Displaying topics in your preferred language in the DB2 Information Center" on page 774
- "Invoking contextual help from a DB2 tool" on page 781
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 773
- "Invoking command help from the command line processor" on page 783
- "Setting the location for accessing the DB2 Information Center: Common GUI help"

Related reference:
- "HELP Command" in the *Command Reference*

Updating the DB2 Information Center installed on your computer or intranet server

The DB2 Information Center available from http://publib.boulder.ibm.com/infocenter/db2help/ will be periodically updated with new or changed documentation. IBM may also make DB2 Information Center updates available to download and install on your computer or intranet server. Updating the DB2 Information Center does not update DB2 client or server products.

Prerequisites:

You must have access to a computer that is connected to the Internet.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:
1. Open the DB2 Information Center hosted on the IBM Web site at: http://publib.boulder.ibm.com/infocenter/db2help/
2. In the Downloads section of the welcome page under the Service and Support heading, click the **DB2 Universal Database documentation** link.
3. Determine if the version of your DB2 Information Center is out of date by comparing the latest refreshed documentation image level to the documentation level you have installed. The documentation level you have installed is listed on the DB2 Information Center welcome page.

4. If a more recent version of the DB2 Information Center is available, download the latest refreshed *DB2 Information Center* image applicable to your operating system.
5. To install the refreshed *DB2 Information Center* image, follow the instructions provided on the Web page.

Related concepts:

- "DB2 Information Center installation scenarios" on page 765

Related tasks:

- "Invoking the DB2 Information Center" on page 772
- "Installing the DB2 Information Center using the DB2 Setup wizard (UNIX)" on page 768
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" on page 770

Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

Procedure:

To display topics in your preferred language in the Internet Explorer browser:

1. In Internet Explorer, click the **Tools** —> **Internet Options** —> **Languages...** button. The Language Preferences window opens.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     Note: Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

To display topics in your preferred language in the Mozilla browser:

1. In Mozilla, select the **Edit** —> **Preferences** —> **Languages** button. The Languages panel is displayed in the Preferences window.
2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

Related concepts:
- "DB2 Information Center" on page 764

DB2 PDF and printed documentation

The following tables provide official book names, form numbers, and PDF file names. To order hardcopy books, you must know the official book name. To print a PDF file, you must know the PDF file name.

The DB2 documentation is categorized by the following headings:
- Core DB2 information
- Administration information
- Application development information
- Business intelligence information
- DB2 Connect information
- Getting started information
- Tutorial information
- Optional component information
- Release notes

The following tables describe, for each book in the DB2 library, the information needed to order the hard copy, or to print or view the PDF for that book. A full description of each of the books in the DB2 library is available from the IBM Publications Center at www.ibm.com/shop/publications/order

Core DB2 information

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect, DB2 Warehouse Manager, or other DB2 products.

Table 180. Core DB2 information

Name	Form Number	PDF File Name
IBM DB2 Universal Database Command Reference	SC09-4828	db2n0x81
IBM DB2 Universal Database Glossary	No form number	db2t0x81
IBM DB2 Universal Database Message Reference, Volume 1	GC09-4840, not available in hardcopy	db2m1x81
IBM DB2 Universal Database Message Reference, Volume 2	GC09-4841, not available in hardcopy	db2m2x81
IBM DB2 Universal Database What's New	SC09-4848	db2q0x81

Administration information

The information in these books covers those topics required to effectively design, implement, and maintain DB2 databases, data warehouses, and federated systems.

Table 181. Administration information

Name	Form number	PDF file name
IBM DB2 Universal Database Administration Guide: Planning	SC09-4822	db2d1x81
IBM DB2 Universal Database Administration Guide: Implementation	SC09-4820	db2d2x81
IBM DB2 Universal Database Administration Guide: Performance	SC09-4821	db2d3x81
IBM DB2 Universal Database Administrative API Reference	SC09-4824	db2b0x81
IBM DB2 Universal Database Data Movement Utilities Guide and Reference	SC09-4830	db2dmx81
IBM DB2 Universal Database Data Recovery and High Availability Guide and Reference	SC09-4831	db2hax81
IBM DB2 Universal Database Data Warehouse Center Administration Guide	SC27-1123	db2ddx81
IBM DB2 Universal Database SQL Reference, Volume 1	SC09-4844	db2s1x81
IBM DB2 Universal Database SQL Reference, Volume 2	SC09-4845	db2s2x81
IBM DB2 Universal Database System Monitor Guide and Reference	SC09-4847	db2f0x81

Application development information

The information in these books is of special interest to application developers or programmers working with DB2 Universal Database (DB2 UDB). You will find information about supported languages and compilers, as well as the documentation required to access DB2 UDB using the various supported programming interfaces, such as embedded SQL, ODBC, JDBC, SQLJ, and CLI. If you are using the DB2 Information Center, you can also access HTML versions of the source code for the sample programs.

Table 182. Application development information

Name	Form number	PDF file name
IBM DB2 Universal Database Application Development Guide: Building and Running Applications	SC09-4825	db2axx81
IBM DB2 Universal Database Application Development Guide: Programming Client Applications	SC09-4826	db2a1x81
IBM DB2 Universal Database Application Development Guide: Programming Server Applications	SC09-4827	db2a2x81

Table 182. Application development information (continued)

Name	Form number	PDF file name
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 1	SC09-4849	db2l1x81
IBM DB2 Universal Database Call Level Interface Guide and Reference, Volume 2	SC09-4850	db2l2x81
IBM DB2 Universal Database Data Warehouse Center Application Integration Guide	SC27-1124	db2adx81
IBM DB2 XML Extender Administration and Programming	SC27-1234	db2sxx81

Business intelligence information

The information in these books describes how to use components that enhance the data warehousing and analytical capabilities of DB2 Universal Database.

Table 183. Business intelligence information

Name	Form number	PDF file name
IBM DB2 Warehouse Manager Standard Edition Information Catalog Center Administration Guide	SC27-1125	db2dix81
IBM DB2 Warehouse Manager Standard Edition Installation Guide	GC27-1122	db2idx81
IBM DB2 Warehouse Manager Standard Edition Managing ETI Solution Conversion Programs with DB2 Warehouse Manager	SC18-7727	iwhe1mstx80

DB2 Connect information

The information in this category describes how to access data on mainframe and midrange servers using DB2 Connect Enterprise Edition or DB2 Connect Personal Edition.

Table 184. DB2 Connect information

Name	Form number	PDF file name
IBM Connectivity Supplement	No form number	db2h1x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition	GC09-4833	db2c6x81
IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition	GC09-4834	db2c1x81
IBM DB2 Connect User's Guide	SC09-4835	db2c0x81

Getting started information

The information in this category is useful when you are installing and configuring servers, clients, and other DB2 products.

Table 185. Getting started information

Name	Form number	PDF file name
IBM DB2 Universal Database Quick Beginnings for DB2 Clients	GC09-4832, not available in hardcopy	db2itx81
IBM DB2 Universal Database Quick Beginnings for DB2 Servers	GC09-4836	db2isx81
IBM DB2 Universal Database Quick Beginnings for DB2 Personal Edition	GC09-4838	db2i1x81
IBM DB2 Universal Database Installation and Configuration Supplement	GC09-4837, not available in hardcopy	db2iyx81
IBM DB2 Universal Database Quick Beginnings for DB2 Data Links Manager	GC09-4829	db2z6x81

Tutorial information

Tutorial information introduces DB2 features and teaches how to perform various tasks.

Table 186. Tutorial information

Name	Form number	PDF file name
Business Intelligence Tutorial: Introduction to the Data Warehouse	No form number	db2tux81
Business Intelligence Tutorial: Extended Lessons in Data Warehousing	No form number	db2tax81
Information Catalog Center Tutorial	No form number	db2aix81
Video Central for e-business Tutorial	No form number	db2twx81
Visual Explain Tutorial	No form number	db2tvx81

Optional component information

The information in this category describes how to work with optional DB2 components.

Table 187. Optional component information

Name	Form number	PDF file name
IBM DB2 Cube Views Guide and Reference	SC18–7298	db2aax81

Table 187. Optional component information (continued)

Name	Form number	PDF file name
IBM DB2 Query Patroller Guide: Installation, Administration and Usage Guide	GC09–7658	db2dwx81
IBM DB2 Spatial Extender and Geodetic Extender User's Guide and Reference	SC27-1226	db2sbx81
IBM DB2 Universal Database Data Links Manager Administration Guide and Reference	SC27-1221	db2z0x82
DB2 Net Search Extender Administration and User's Guide **Note:** HTML for this document is *not* installed from the HTML documentation CD.	SH12-6740	N/A

Release notes

The release notes provide additional information specific to your product's release and FixPak level. The release notes also provide summaries of the documentation updates incorporated in each release, update, and FixPak.

Table 188. Release notes

Name	Form number	PDF file name
DB2 Release Notes	See note.	See note.
DB2 Installation Notes	Available on product CD-ROM only.	Not available.

Note: The Release Notes are available in:

- XHTML and Text format, on the product CDs
- PDF format, on the PDF Documentation CD

In addition the portions of the Release Notes that discuss *Known Problems and Workarounds* and *Incompatibilities Between Releases* also appear in the DB2 Information Center.

To view the Release Notes in text format on UNIX-based platforms, see the Release.Notes file. This file is located in the DB2DIR/Readme/*%L* directory, where `%L` represents the locale name and `DB2DIR` represents:

- For AIX operating systems: /usr/opt/db2_08_01
- For all other UNIX-based operating systems: /opt/IBM/db2/V8.1

Related concepts:

- "DB2 documentation and help" on page 763

Related tasks:

- "Printing DB2 books from PDF files" on page 780
- "Ordering printed DB2 books" on page 780
- "Invoking contextual help from a DB2 tool" on page 781

Printing DB2 books from PDF files

You can print DB2 books from the PDF files on the *DB2 PDF Documentation* CD. Using Adobe Acrobat Reader, you can print either the entire book or a specific range of pages.

Prerequisites:

Ensure that you have Adobe Acrobat Reader installed. If you need to install Adobe Acrobat Reader, it is available from the Adobe Web site at www.adobe.com

Procedure:

To print a DB2 book from a PDF file:
1. Insert the *DB2 PDF Documentation* CD. On UNIX operating systems, mount the DB2 PDF Documentation CD. Refer to your *Quick Beginnings* book for details on how to mount a CD on UNIX operating systems.
2. Open index.htm. The file opens in a browser window.
3. Click on the title of the PDF you want to see. The PDF will open in Acrobat Reader.
4. Select **File** → **Print** to print any portions of the book that you want.

Related concepts:
- "DB2 Information Center" on page 764

Related tasks:
- "Mounting the CD-ROM (AIX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (HP-UX)" in the *Quick Beginnings for DB2 Servers*
- "Mounting the CD-ROM (Linux)" in the *Quick Beginnings for DB2 Servers*
- "Ordering printed DB2 books" on page 780
- "Mounting the CD-ROM (Solaris Operating Environment)" in the *Quick Beginnings for DB2 Servers*

Related reference:
- "DB2 PDF and printed documentation" on page 775

Ordering printed DB2 books

If you prefer to use hardcopy books, you can order them in one of three ways.

Procedure:

Printed books can be ordered in some countries or regions. Check the IBM Publications website for your country or region to see if this service is available in your country or region. When the publications are available for ordering, you can:
- Contact your IBM authorized dealer or marketing representative. To find a local IBM representative, check the IBM Worldwide Directory of Contacts at www.ibm.com/planetwide
- Phone `1-800-879-2755` in the United States or `1-800-IBM-4YOU` in Canada.

- Visit the IBM Publications Center at http://www.ibm.com/shop/publications/order. The ability to order books from the IBM Publications Center may not be available in all countries.

At the time the DB2 product becomes available, the printed books are the same as those that are available in PDF format on the *DB2 PDF Documentation CD*. Content in the printed books that appears in the *DB2 Information Center CD* is also the same. However, there is some additional content available in DB2 Information Center CD that does not appear anywhere in the PDF books (for example, SQL Administration routines and HTML samples). Not all books available on the DB2 PDF Documentation CD are available for ordering in hardcopy.

Note: The DB2 Information Center is updated more frequently than either the PDF or the hardcopy books; install documentation updates as they become available or refer to the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2help/ to get the most current information.

Related tasks:

- "Printing DB2 books from PDF files" on page 780

Related reference:

- "DB2 PDF and printed documentation" on page 775

Invoking contextual help from a DB2 tool

Contextual help provides information about the tasks or controls that are associated with a particular window, notebook, wizard, or advisor. Contextual help is available from DB2 administration and development tools that have graphical user interfaces. There are two types of contextual help:

- Help accessed through the **Help** button that is located on each window or notebook
- Infopops, which are pop-up information windows displayed when the mouse cursor is placed over a field or control, or when a field or control is selected in a window, notebook, wizard, or advisor and F1 is pressed.

The **Help** button gives you access to overview, prerequisite, and task information. The infopops describe the individual fields and controls.

Procedure:

To invoke contextual help:

- For window and notebook help, start one of the DB2 tools, then open any window or notebook. Click the **Help** button at the bottom right corner of the window or notebook to invoke the contextual help.

  You can also access the contextual help from the **Help** menu item at the top of each of the DB2 tools centers.

  Within wizards and advisors, click on the Task Overview link on the first page to view contextual help.
- For infopop help about individual controls on a window or notebook, click the control, then click **F1**. Pop-up information containing details about the control is displayed in a yellow window.

Note: To display infopops simply by holding the mouse cursor over a field or control, select the **Automatically display infopops** check box on the **Documentation** page of the Tool Settings notebook.

Similar to infopops, diagnosis pop-up information is another form of context-sensitive help; they contain data entry rules. Diagnosis pop-up information is displayed in a purple window that appears when data that is not valid or that is insufficient is entered. Diagnosis pop-up information can appear for:

– Compulsory fields.

– Fields whose data follows a precise format, such as a date field.

Related tasks:

- "Invoking the DB2 Information Center" on page 772
- "Invoking message help from the command line processor" on page 782
- "Invoking command help from the command line processor" on page 783
- "Invoking SQL state help from the command line processor" on page 783
- "Access to the DB2 Information Center: Concepts help"
- "How to use the DB2 UDB help: Common GUI help"
- "Setting the location for accessing the DB2 Information Center: Common GUI help"
- "Setting up access to DB2 contextual help and documentation: Common GUI help"

Invoking message help from the command line processor

Message help describes the cause of a message and describes any action you should take in response to the error.

Procedure:

To invoke message help, open the command line processor and enter:

```
? XXXnnnnn
```

where *XXXnnnnn* represents a valid message identifier.

For example, `? SQL30081` displays help about the SQL30081 message.

Related concepts:

- "Introduction to messages" in the *Message Reference Volume 1*

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking command help from the command line processor

Command help explains the syntax of commands in the command line processor.

Procedure:

To invoke command help, open the command line processor and enter:

```
? command
```

where *command* represents a keyword or the entire command.

For example, `? catalog` displays help for all of the CATALOG commands, while `? catalog database` displays help only for the CATALOG DATABASE command.

Related tasks:

- "Invoking contextual help from a DB2 tool" on page 781
- "Invoking the DB2 Information Center" on page 772
- "Invoking message help from the command line processor" on page 782
- "Invoking SQL state help from the command line processor" on page 783

Related reference:

- "db2 - Command Line Processor Invocation Command" in the *Command Reference*

Invoking SQL state help from the command line processor

DB2 Universal Database returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure:

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

Related tasks:

- "Invoking the DB2 Information Center" on page 772
- "Invoking message help from the command line processor" on page 782
- "Invoking command help from the command line processor" on page 783

DB2 tutorials

The DB2® tutorials help you learn about various aspects of DB2 Universal Database. The tutorials provide lessons with step-by-step instructions in the areas of developing applications, tuning SQL query performance, working with data warehouses, managing metadata, and developing Web services using DB2.

Before you begin:

You can view the XHTML versions of the tutorials from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some tutorial lessons use sample data or code. See each tutorial for a description of any prerequisites for its specific tasks.

DB2 Universal Database tutorials:

Click on a tutorial title in the following list to view that tutorial.

Business Intelligence Tutorial: Introduction to the Data Warehouse Center
: Perform introductory data warehousing tasks using the Data Warehouse Center.

Business Intelligence Tutorial: Extended Lessons in Data Warehousing
: Perform advanced data warehousing tasks using the Data Warehouse Center.

Information Catalog Center Tutorial
: Create and manage an information catalog to locate and use metadata using the Information Catalog Center.

Visual Explain Tutorial
: Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2® products.

DB2 documentation
: Troubleshooting information can be found throughout the DB2 Information Center, as well as throughout the PDF books that make up the DB2 library. You can refer to the ″Support and troubleshooting″ branch of the DB2 Information Center navigation tree (in the left pane of your browser window) to see a complete listing of the DB2 troubleshooting documentation.

DB2 Technical Support Web site
: Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs), FixPaks and the latest listing of internal DB2 error codes, and other resources. You can search through this knowledge base to find possible solutions to your problems.

  Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/winos2unix/support

DB2 Problem Determination Tutorial Series
: Refer to the DB2 Problem Determination Tutorial Series Web site to find information on how to quickly identify and resolve problems you might encounter while working with DB2 products. One tutorial introduces you to the DB2 problem determination facilities and tools available, and helps you decide when to use them. Other tutorials deal with related topics, such

as ″Database Engine Problem Determination″, ″Performance Problem Determination″, and ″Application Problem Determination″.

See the full set of DB2 problem determination tutorials on the DB2 Technical Support site at http://www.ibm.com/software/data/support/pdm/db2tutorials.html

Related concepts:

- "DB2 Information Center" on page 764
- "Introduction to problem determination - DB2 Technical Support tutorial" in the *Troubleshooting Guide*

Accessibility

Accessibility features help users with physical disabilities, such as restricted mobility or limited vision, to use software products successfully. The following list specifies the major accessibility features in DB2® Version 8 products:

- All DB2 functionality is available using the keyboard for navigation instead of the mouse. For more information, see "Keyboard input and navigation."
- You can customize the size and color of the fonts on DB2 user interfaces. For more information, see "Accessible display."
- DB2 products support accessibility applications that use the Java™ Accessibility API. For more information, see "Compatibility with assistive technologies" on page 786.
- DB2 documentation is provided in an accessible format. For more information, see "Accessible documentation" on page 786.

Keyboard input and navigation

Keyboard input

You can operate the DB2 tools using only the keyboard. You can use keys or key combinations to perform operations that can also be done using a mouse. Standard operating system keystrokes are used for standard operating system operations.

For more information about using keys or key combinations to perform operations, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard navigation

You can navigate the DB2 tools user interface using keys or key combinations.

For more information about using keys or key combinations to navigate the DB2 Tools, see Keyboard shortcuts and accelerators: Common GUI help.

Keyboard focus

In UNIX® operating systems, the area of the active window where your keystrokes will have an effect is highlighted.

Accessible display

The DB2 tools have features that improve accessibility for users with low vision or other visual impairments. These accessibility enhancements include support for customizable font properties.

Font settings

You can select the color, size, and font for the text in menus and dialog windows, using the Tools Settings notebook.

For more information about specifying font settings, see Changing the fonts for menus and text: Common GUI help.

Non-dependence on color

You do not need to distinguish between colors in order to use any of the functions in this product.

Compatibility with assistive technologies

The DB2 tools interfaces support the Java Accessibility API, which enables you to use screen readers and other assistive technologies with DB2 products.

Accessible documentation

Documentation for DB2 is provided in XHTML 1.0 format, which is viewable in most Web browsers. XHTML allows you to view documentation according to the display preferences set in your browser. It also allows you to use screen readers and other assistive technologies.

Syntax diagrams are provided in dotted decimal format. This format is available only if you are accessing the online documentation using a screen-reader.

Related concepts:

- "Dotted decimal syntax diagrams" on page 786

Related tasks:

- "Keyboard shortcuts and accelerators: Common GUI help"
- "Changing the fonts for menus and text: Common GUI help"

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.
- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one

data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Related concepts:

- "Accessibility" on page 785

Related tasks:

- "Keyboard shortcuts and accelerators: Common GUI help"

Related reference:

- "How to read the syntax diagrams" on page x

Common Criteria certification of DB2 Universal Database products

DB2 Universal Database is being evaluated for certification under the Common Criteria at evaluation assurance level 4 (EAL4). For more information about Common Criteria, see the Common Criteria web site at: http://niap.nist.gov/cc-scheme/.

Appendix Q. Notices

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

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

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

© (*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

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

ACF/VTAM
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
BookManager
C Set++
C/370
CICS
Database 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Information Integrator
DB2 Query Patroller
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eServer
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
iSeries
LAN Distance
MVS
MVS/ESA
MVS/XA
Net.Data
NetView
OS/390
OS/400
PowerPC
pSeries
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/400
SQL/DS
System/370
System/390
SystemView
Tivoli
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WebSphere
WIN-OS/2
z/OS
zSeries

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 UDB documentation library:

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

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Index

Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-888-426-4343 to learn about available service options
- 1-800-IBM-4YOU (426-4968) for DB2 marketing and sales

In Canada, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-800-465-9600 to learn about available service options
- 1-800-IBM-4YOU (1-800-426-4968) for DB2 marketing and sales

To locate an IBM office in your country or region, check IBM's Directory of Worldwide Contacts on the web at http://www.ibm.com/planetwide

Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at http://www.ibm.com/software/data/db2/udb

This site contains the latest information on the technical library, ordering books, product downloads, newsgroups, FixPaks, news, and links to web resources.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-IBM-CALL (1-800-426-2255) to order products or to obtain general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, go to the IBM Worldwide page at www.ibm.com/planetwide

Printed in USA

SC09-4844-01

Spine information:

IBM® DB2 Universal Database™ SQL Reference, Volume 1 Version 8.2