DB2 Server for VSE & VM

SQL Reference

Version 7 Release 5

SC09-2989-02

DB2 Server for VSE & VM

SQL Reference

Version 7 Release 5

SC09-2989-02

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 467.

Contents

Summary of Changes

This is a summary of the technical changes to the DB2 Server for VSE & VM database management system for this edition of the book. Several manuals are affected by some or all of the changes discussed here. For your convenience, the changes made in this edition are identified in the text by a vertical bar (|) in the left margin. This edition may also include minor corrections and editorial changes that are not identified.

This summary does not list incompatibilities between releases of the DB2 Server for VSE & VM product; see either the *DB2 Server for VSE & VM SQL Reference*, *DB2 Server for VM System Administration*, or the *DB2 Server for VSE System Administration* manuals for a discussion of incompatibilities.

Summary of Changes for DB2 Version 7 Release 5

Version 7 Release 5 of the DB2 Server for VSE & VM database management system is intended to run on the Z/VM Version 5 Release 2 or later environment and on the Z/VSE(®) Version 3 Release 1 or later environment.

Enhancements, New Functions, and New Capabilities

The following have been added to DB2 Version 7 Release 5:

Explain Option on DBSU REBIND PACKAGE Command

This new functionality allows the EXPLAIN(YES/NO) option on REBIND PACKAGE command. If EXPLAIN(YES) is issued, then all four update tables (structure, plan, cost, reference) will be updated. If EXPLAIN(NO) is issued, then none of the four update tables will be updated.

For more information, see the following DB2 Server for VSE & VM documentation:

- *DB2 Server for VSE & VM Database Services Utility*
- *DB2 Server for VSE & VM Performance Tuning Handbook*
- *DB2 Server for VSE & VM Quick Reference*
- *DB2 Server for VSE & VM SQL Reference*

For Fetch only

This new functionality accepts the ″FOR FETCH ONLY″ clause after a cursor select statement. It causes a cursor to become read-only (no UPDATEs or DELETEs are permitted using this cursor). If a read-only cursor is referenced in an UPDATE or DELETE statement, SQLCODE -510 will be issued and the statement is not processed. In addition, under the SBLOCK preprocessor option, ″FOR FETCH ONLY″ forces blocking to be used on the read-only cursor regardless of whether there is a COMMIT. If there is no ″FOR FETCH ONLY″ clause, under SBLOCK, blocking would only be done if a COMMIT was absent.

For more information, see the following DB2 Server for VSE & VM documentation:

- *DB2 Server for VM Messages and Codes*
- *DB2 Server for VSE & VM Application Programming*
- *DB2 Server for VSE & VM Performance Tuning Handbook*
- *DB2 Server for VSE & VM Quick Reference*

- *DB2 Server for VSE & VM SQL Reference*

Application Message Formatter

This functionality provides an Application Programming Interface (API) that retrieves the descriptive text for an SQLCODE, given an SQLCA input parameter. The API will be available for Assembly, COBOL, C, PL/I and FORTRAN.

In DB2 for VM and DB2 for VSE Online, the user may specify the language of the returned text. The languages supported by DB2 for VSE/VM are American English (AMENG), uppercase English (UCENG), German (GER), French (FRANC) and Japanese (KANJI). VSE Batch does not support switching to another language. Therefore the default will be used regardless of the user's specification. The values of SQLCODE, SQLSTATE, SQLERRD1 and SQLERRD2 will be automatically appended to the returned text. The user may also specify to have the entire SQLCA included. If the SQLCODE could not be found in the repository, the entire SQLCA will be returned in the buffer.

If the SQLCA was set by another product (such as DB2 UBD), the descriptive text is retrieved if the SQLCODE exists in the DB2 for VM/VSE repositories. However, the token substitutions may not be correct.

For more information, see *DB2 Server for VSE & VM Application Programming*.

Convert buffer read/write to compiler macro

The DRDA code has over 100 small modules. Each call to an external module has a certain amount of overhead associated with it. Certain modules are called very frequently and this can add up to a significant amount of time. This functionality improves the performance by converting few modules to macros or internal procedures, to reduce this overhead.

Modify Build Tree Creation

This functionality modifies Build Tree creation used by DRDA parsing and generation. It is built in such a way that every code point that is used to search through the tree must be converted to a different format before the search can be done. If modified build tree was created with the converted point, then the code point would not have to be converted every time the tree must be searched. This improves the performance of the DRDA code path length with the minimal search.

Split code point search routines

When parsing a data stream within each parser action routine, a binary search is done to find the specific code point. Some action specific routines are quite large, so the binary search can be long. Splitting and spreading the code point evenly among other modules would reduce the overheads and improves the performance of the DRDA code path length.

DRDA Multi-Row Insert

Multi Row insert is a means of caching homogenous insert statements and sending them as a block to the server for processing. This reduces the overhead of sending a large number of singular inserts and receiving as many responses.

Buffering of homogenous inserts eliminates the need to send an SQL statement to the DB2 server every time an insert is made, thereby improving performance over DRDA.

For more information, see the following DB2 Server for VSE & VM documentation:

- *DB2 Server for VSE & VM Application Programming*

- *DB2 Server for VSE & VM Database Administration*
- *DB2 Server for VM System Administration*
- *DB2 Server for VSE & VM Performance Tuning Handbook*
- *DB2 Server for VSE & VM Quick Reference*
- *DB2 Server for VSE & VM SQL Reference*

Connection Pooling for DRDA TCP/IP in Online Resource Adapter

Connection pooling is a technique that allows multiple users to share a cached set of pre-established connections that provide access to a database. Establishing a connection between a user and a server takes a sizeable time. Users who have validated their entry to a database once need not establish a connection every time a request is submitted. Instead, they can use a pre-established connection from a pool of such connections and get their results much faster.

From the user's point of view, there is a considerable improvement in response time after this line item is implemented.

For more information, see the following documentation on DB2 Server for VSE & VM:
- *DB2 Server for VSE System Administration*
- *DB2 Server for VSE & VM Application Programming*
- *DB2 Server for VSE & VM Operation*
- *DB2 Server for VSE & VM Performance Tuning Handbook*

IBM DB2 Server for VSE, Client Edition

This feature allows the customer the flexibility to install and use only the client (run-time support) component of DB2 Server for VSE without the requirement to buy and install the server component during the installation process of DB2 server for VSE product. The client-only installation enables customers to reduce the total cost of ownership when they have their databases residing on a non-local platform (like VM, z/OS, LUW) and have a large number of their DB2 applications on VSE (like ISQL on CICS, DBSU on VSE, other online/batch applications on VSE).

For more information, see the following DB2 Server for VSE & VM documentation:
- *DB2 Server for VSE System Administration*
- *DB2 Server for VSE Program Directory*

IBM DB2 Server for VM, Client Edition

This feature allows the customer the flexibility to install and use only the client (run-time support) component of DB2 Server for VM without the requirement to buy and install the server component during the installation process of DB2 server for VM product. The client-only installation enables our customers to reduce the total cost of ownership when they have their databases residing on a non-local platform (like VM, z/OS, LUW) and have a large number of their DB2 applications on VM (like ISQL, DBSU, other user applications on VM).

For more information, see the following DB2 Server for VSE & VM documentation:
- *DB2 Server for VM System Administration*
- *DB2 Server for VM Program Directory*

Handling Commit Responses from DB2 UDB Stored Procedures

This feature will allow DB2 Resource Manager on VSE/VM to accept and process results of a stored procedure running in a UDB server with a COMMIT statement in the stored procedure.

Currently, DB2 for VM/VSE client does not handle responses from ’COMMIT’ statements coded in DB2 UDB stored procedures. Implementation of this feature will enable handling responses of COMMIT statements in DB2 UDB stored procedures and thus allow users to have COMMIT statements in their stored procedures, while using DB2 for VM/VSE client.

COMMIT statements, however, are not allowed in stored procedures on the DB2 Server for VM/VSE.

For more information, see *DB2 Server for VSE & VM Application Programming*.

Make on-line programs AMODE 31 RMODE ANY

This feature converts DB2 server for VSE online program which presently operate under 24 bit addressing mode from AMODE 24, to AMODE 31 RMODE ANY. Presently, all the online programs are loaded below 16M line. Implementation of this line item ensures that all the online program will be loaded above the 16M line, which results in more virtual storage below the line, which can be utilized by other applications.

For more information, see the following DB2 Server for VSE & VM documentation:

- *DB2 Server for VSE System Administration*
- *DB2 Server for VSE Program Directory*

Provide BIND File Support in VM and in VSE Batch Environments

This feature provides the facility of binding packages across servers. The process of binding is achieved by dividing the program preparation method into two steps. The first step does the precompilation of the embedded SQL programs with the prep parameter ’BIND’. Invocation of VSE/VM preprocessor creates a ’bindfile’. The bindfile can be bound against any DB2 server using VSE/VM binder. During this process, the access path is generated, SQL statements are verified, authorization checks are performed, and package on the target server is created. This line item eliminates the need of re-prepping the source code or porting of packages across DB2 servers.

For more information, see the following DB2 Server for VSE & VM documentation:

- *DB2 REXX SQL for VM/ESA Installation and Reference*
- *DB2 Server for VM Messages and Codes*
- *DB2 Server for VSE & VM Application Programming*
- *DB2 Server for VSE & VM Database Administration*
- *DB2 Server for VM Program Directory*
- *DB2 Server for VSE Program Directory*

Convert TCP/IP LE/C interface to EZASMI API

The feature of converting TCP/IP LE/C interface to EZASMI API intends to replace the current LE/C interface and implement the EZA Assembler Interface (EZASMI)to enhance performance in DB2 Client/Server for VSE over DRDA. Currently, either LE/C interface or CSI Assembler Interface is used for TCP/IP functions. The EZASMI interface makes the code all Assembler.

For more information, see *DB2 Server for VSE Program Directory*

Chapter 1. Introduction

This introductory chapter:
- Identifies the book's purpose and audience
- Explains how the book is organized
- Explains how to use the book
- Explains how to read the syntax diagrams.

Note: For ease of reading, this book follows the following convention:
- the term **DB2 Server for VSE & VM** is used where the discussion refers to both operating system environments (VM and VSE)
- the terms **DB2 Server for VM** and **DB2 Server for VSE** are used where the discussion must refer explicitly to either the VM or VSE operating system environments.

Who This Book Is For

This book is for programmers, system administrators, and database administrators who want to use SQL to access a DB2 Server for VSE & VM database. This book is a reference rather than a tutorial or guide. It assumes you are already familiar with SQL. This book also assumes that you will be writing applications for the VM or VSE environment and therefore presents the full functions of the DB2 Server for VSE & VM program.

Prerequisite Knowledge

It is assumed that you possess an understanding of system administration, database administration, or application programming in the DB2 Server for VM or DB2 Server for VSE environment, as provided by the appropriate guide, and you have some knowledge of the following:
- VM (CMS, CP) or VSE (CICS or batch, as applicable)
- A programming language
- Structured Query Language (SQL).

It also assumes that you are familiar with the information found in the *DB2 Server for VSE & VM Overivew* manual.

How This Book Is Organized

This book has the following sections:
- Chapter 1, "Introduction," on page 1 identifies the purpose, the audience, and the use of the book.
- Chapter 2, "Concepts," on page 9 discusses the basic concepts of relational databases and SQL.
- Chapter 3, "Language Elements," on page 35 describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- Chapter 4, "Functions," on page 91 contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.
- Chapter 5, "Queries," on page 121 describes the various forms of a query, which is a component of various SQL statements.

- Chapter 6, “Statements,” on page 137 contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- The appendixes contain information about SQL limits, SQLCA, SQLDA, system catalog tables, SQL reserved words, supplied sample tables, and terminology differences.

Syntax Notation Conventions

Throughout this manual, syntax is described using the structure defined below.

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

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

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

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

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

  Diagrams of syntactical units that are not complete statements start with the ►── symbol and end with the ──► symbol.
- Some SQL statements, Interactive SQL (ISQL) commands, or database services utility (DBS Utility) commands can stand alone. For example:

```
►►──SAVE──────────────────────────────────────────────────────────────►◄
```

  Others must be followed by one or more keywords or variables. For example:

```
►►──SET AUTOCOMMIT OFF────────────────────────────────────────────────►◄
```

- Keywords may have parameters associated with them which represent user-supplied names or values. These names or values can be specified as either constants or as user-defined variables called *host_variables* (*host_variables* can only be used in programs).

```
►►──DROP SYNONYM──synonym─────────────────────────────────────────────►◄
```

- Keywords appear in either uppercase (for example, SAVE) or mixed case (for example, CHARacter). All uppercase characters in keywords must be present; you can omit those in lowercase.
- Parameters appear in lowercase and in italics (for example, *synonym*).
- If such symbols as punctuation marks, parentheses, or arithmetic operators are shown, you must use them as indicated by the syntax diagram.
- All items (parameters and keywords) must be separated by one or more blanks.
- Required items appear on the same horizontal line (the main path). For example, the parameter *integer* is a required item in the following command:

```
►►──SHOW DBSPACE──integer──────────────────────────────────────────────►◄
```

This command might appear as:

```
SHOW DBSPACE 1
```

- Optional items appear below the main path. For example:

```
►►──CREATE──┬────────┬──INDEX──────────────────────────────────────────►◄
            └─UNIQUE─┘
```

This statement could appear as either:

```
CREATE INDEX
```

or

```
CREATE UNIQUE INDEX
```

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

  If you must choose one of the items, one item appears on the main path. For example:

```
►►──SHOW LOCK DBSPACE──┬─ALL─────┬─────────────────────────────────────►◄
                       └─integer─┘
```

Here, the command could be either:

```
SHOW LOCK DBSPACE ALL
```

or

```
SHOW LOCK DBSPACE 1
```

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

```
►►──BACKWARD──┬─────────┬──────────────────────────────────────────────►◄
              ├─integer─┤
              └─MAX─────┘
```

Here, the command could be:

```
BACKWARD
```

or

```
BACKWARD 2
```

or

```
BACKWARD MAX
```

- The repeat symbol indicates that an item can be repeated. For example:

```
           ┌──────┐
           ▼      │
►►──ERASE────name─┴──────────────────────────────────────────────────►◄
```

  This statement could appear as:

  ```
  ERASE NAME1
  ```

  or

  ```
  ERASE NAME1 NAME2
  ```

  A repeat symbol above a stack indicates that you can make more than one choice from the stacked items, or repeat a choice. For example:

```
                  ┌─,──────────────────┐
                  ▼                    │
►►──VALUES──(──────┬─constant──────────┬┴──)──────────────────────────►◄
                   ├─host_variable_list─┤
                   ├─NULL───────────────┤
                   └─special_register───┘
```

- If an item is above the main line, it represents a default, which means that it will be used if no other item is specified. In the following example, the ASC keyword appears above the line in a stack with DESC. If neither of these values is specified, the command would be processed with option ASC.

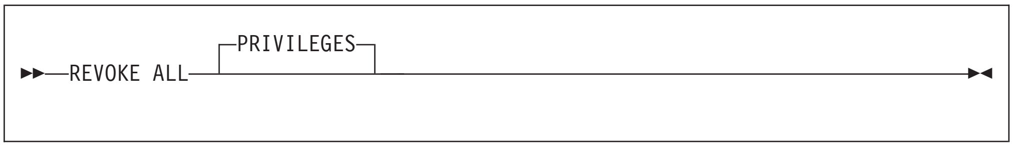
```
     ┌─ASC──┐
►►───┤      ├──────────────────────────────────────────────────────────►◄
     └─DESC─┘
```

- When an optional keyword is followed on the same path by an optional default parameter, the default parameter is assumed if the keyword is not entered. However, if this keyword is entered, one of its associated optional parameters must also be specified.

  In the following example, if you enter the optional keyword PCTFREE =, you also have to specify one of its associated optional parameters. If you do not enter PCTFREE =, the database manager will set it to the default value of 10.

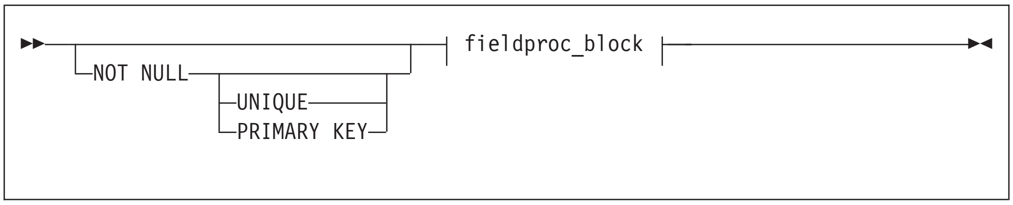
```
     ┌─PCTFREE = 10──────┐
►►───┤                   ├─────────────────────────────────────────────►◄
     └─PCTFREE = integer─┘
```

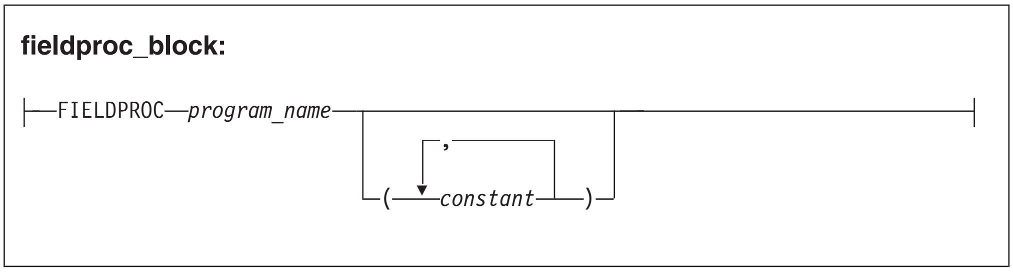
- Words that are only used for readability and have no effect on the execution of the statement are shown as a single uppercase default. For example:



Here, specifying either REVOKE ALL or REVOKE ALL PRIVILEGES means the same thing.

- Sometimes a single parameter represents a fragment of syntax that is expanded below. In the following example, **fieldproc_block** is such a fragment and it is expanded following the syntax diagram containing it.





SQL Reserved Words

The following words are reserved in the SQL language. They cannot be used in SQL statements except for their defined meaning in the SQL syntax or as host variables, preceded by a colon.

In particular, they cannot be used as names for tables, indexes, columns, views, or dbspaces unless they are enclosed in double quotation marks (").

ACQUIRE
ADD
ALL
ALTER
AND
ANY
AS
ASC
AVG

BETWEEN
BY

CALL
CHAR
CHARACTER
COLUMN
COMMENT
COMMIT
CONCAT
CONNECT
COUNT
CREATE
CURRENT

DBA
DBSPACE
DELETE
DESC
DISTINCT
DOUBLE
DROP

EXCLUSIVE
EXECUTE
EXISTS
EXPLAIN

FIELDPROC
FOR
FROM

GRANT
GRAPHIC
GROUP

HAVING

IDENTIFIED
IN
INDEX
INSERT
INTO
IS

LIKE
LOCK
LONG

MAX
MIN
MODE

NAMED
NHEADER
NOT
NULL

OF
ON
OPTION
OR
ORDER

PACKAGE
PAGE
PAGES
PCTFREE
PCTINDEX
PRIVATE
PRIVILEGES
PROGRAM
PUBLIC

RESOURCE
REVOKE
ROLLBACK
ROW
RUN

SCHEDULE
SELECT
SET
SHARE
SOME
STATISTICS
STORPOOL
SUM
SYNONYM

TABLE
TO

UNION
UNIQUE
UPDATE
USER

VALUES
VIEW

WHERE
WITH
WORK

Conventions for Representing Mixed Data Values

When mixed data values are shown in examples the following conventions apply:

Convention	**Meaning**
<	Represents the mixed *shift-out* character (X'0E').
>	Represents the mixed *shift-in* character (X'0F').
x	Represents an SBCS character (where x can be any lowercase character).
XX	Represents a DBCS character (where XX can be any double uppercase character).

Short Forms Used in Syntax Diagrams

Some words have been shortened in some of the syntax diagrams in this book. The words are:

Full Word	Short Form
character	char
expressions	exp
string	str

Chapter 2. Concepts

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

SQL statements are processed 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. This transformation occurs when the SQL statement is *prepared*. Statement preparation is also known as *binding*.

All executable SQL statements must be prepared before they can be processed. 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.

Static SQL

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

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

The preparation of an SQL application program includes parsing and validation, the binding of its SQL statements, and the compilation of the modified source program.

Dynamic SQL

A *dynamic* SQL statement is prepared during the execution of an SQL application and the operational form of the statement does not persist beyond the unit of work. The source form of the statement is a character string that is passed to the database manager by the program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.

SQL statements embedded in a REXX application are dynamic SQL statements. [1] SQL statements submitted to an interactive SQL facility are also dynamic SQL statements.

1. The DB2 REXX SQL feature must be installed.

Interactive SQL

An interactive SQL facility is associated with the database manager. Essentially, every interactive SQL facility is an SQL application program that reads statements from a terminal, prepares and processes them dynamically, and displays the results to the user. Such SQL statements are said to be issued *interactively*. The *DB2 Server for VSE & VM Interactive SQL Guide and Reference* manual discusses interactive SQL in greater detail. An associated product, Query Management Facility (QMF), also uses DB2 Server for VSE & VM interactively.

Extended Dynamic SQL

Extended dynamic statements support direct creation and maintenance of packages for DB2 Server for VSE & VM data and provide a function similar to that provided by the DB2 Server for VSE & VM preprocessors. These functions are particularly useful where:

- The current preprocessors do not support the language of the application or support program that is needed.
- SQL statements are conceived and built dynamically, but are processed repetitively (in a different logical unit of work). In this case it is a performance benefit to avoid having to repeat the preprocessing of statements each time they are processed, as would be required for normal dynamic statements.
- It is desirable to build and maintain an application package of SQL statements to be shared by a group of users.

Individual SQL statements can be added or deleted without affecting or repeating the preprocessing of other SQL statements in the group (a group can be stored in one package).

By using the extended dynamic statements, development programmers can write their own preprocessors or database interface routines that support preplanned access to the database manager. Preplanned access means that access paths to data are optimized once when the statement is prepared. They need not be prepared again for each execution.

Extended dynamic SQL statements may be used only in Assembler and REXX programs.

Relational Database

A *relational database* is a database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

Tables

Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. A *column* is the vertical component of a table. It has a name and a defined data type (for example, character, decimal, or integer). A *row* is the horizontal component of a table. At the intersection of every column and row is a specific data item called a *value*. A row contains a sequence of values such that the *n*th value is a value of the *n*th column of the table. There is no inherent order of the rows within a table but there is a defined order of columns for a table.

A *base table* is created with the CREATE TABLE statement and holds persistent user data. A *result table* or an *active set* is a set of rows that the database manager selects or generates from one or more base tables.

Keys

A *key* is one or more columns identified as such in the description of a table, an index, or a referential constraint. The same column can be part of more than one key. A key composed of more than one column is called a composite key.

A *composite key* is an ordered set of columns of the same table. The ordering of the columns is not constrained by their ordering within the table. The term *value* when used with respect to a composite key denotes a composite value. Thus, a rule such as "the value of the foreign key must be equal to the value of the primary key" means that each component of the value of the foreign key must be equal to the corresponding component of the value of the primary key.

A *unique key* is a key that is constrained so that no two of its values are equal. The constraint is enforced by the database manager during the execution of INSERT and UPDATE statements. The mechanism used to enforce the constraint is called a unique index. Thus, every unique key is a key of a unique index.

Primary Keys

A *primary key* is a unique key that is part of the definition of a table. A table can have at most one primary key, and the columns of a primary key cannot contain null values. Primary keys are optional and can be defined in CREATE TABLE statements or ALTER TABLE statements.

The unique index on a primary key is called the *primary index*. When a primary key is defined in a CREATE TABLE statement, the primary index is automatically created by the database manager.

When a primary key is defined in an ALTER TABLE statement, a primary index is automatically created by the database manager even if a unique index exists on the same columns of that primary key.

Integrity

Integrity refers to the accuracy of data in the database. Integrity is maintained in the following ways:

1. An entire group of related changes is either made to the database, or the entire operation is canceled. For example, when money is transferred from one bank account to another, the database manager ensures that both the deduction from the one account and the deposit to the other account complete successfully or none of the changes are made. This is called *atomic integrity*; it protects other users and programs from using inconsistent data.
2. Duplicate rows of information for the same entity can be avoided. For example, an EMPLOYEE table consisting of employee number, employee name, and department number can be defined so that values are unique for an employee number and name, eliminating duplicate rows for any employee. This is called *entity integrity*.
3. Integrity of data in related tables can be ensured. For example, a DEPARTMENT table may contain department numbers and other information related to the EMPLOYEE table. A relation can be defined so that only valid

and existing department numbers as specified in the DEPARTMENT table can appear in the EMPLOYEE table. Changes to the DEPARTMENT table can be automatically reflected in the EMPLOYEE table. This is called *referential integrity*. Rules or referential constraints can be defined by users to ensure referential integrity between tables.

Data Integrity

The database manager protects other users and programs from using inconsistent or wrong data by preventing more than one user or application from simultaneously updating data; by allowing the user to rollback uncommitted changes; and by using entity integrity and referential integrity.

Entity Integrity

Entity integrity may be maintained in two ways: by defining a primary key on a table or placing a unique constraint on a column.

You define a primary key on a table to ensure that duplicate rows do not occur. The database manager enforces the uniqueness of the primary key by automatically creating a unique index on its columns. A primary key is a part of the table definition and is defined when the table is created or altered. Primary keys are also used in defining referential integrity.

Columns that are not used to define a primary key can be defined to have unique values. Define a *unique constraint* on a column when you wish the database manager not to accept a row of data if a unique column already contains the same value in another row.

Referential Integrity

Referential integrity allows the definition of relationships between tables such that the existence of values in one table depends on the existence of the same values in another table. The database manager supports referential integrity by providing for the definition of primary keys, foreign keys, and through a set of rules defining the relationships among the tables. Together, these are known as *referential constraints*.

Relationships Between Tables

The relationship defined by a referential constraint is a set of connections between the rows of two or more tables. The tables are related through matching column values in the tables. A table is considered a parent table if its primary key is referenced in a referential constraint, or a dependent table if it has a foreign key and is related to a parent table through a referential constraint. A table can be both a parent and a dependent table, depending on its relationship to other tables.

The relationship is defined using the CREATE TABLE statement for new tables and the ALTER TABLE statement for existing tables. When you use these statements, you specify the rules that must be followed in both parent and dependent tables when rows are deleted.

The following relationships may occur:

- A *parent table* has a primary key and is a parent in at least one relationship.
- A *dependent table* has at least one foreign key (defined below) and is a dependent in at least one relationship. A table can be a dependent in any number of relationships.
- A table can be both a dependent table and a parent table, but not of itself.

- A table is a *descendent* of table T if it is a dependent of T or it is a dependent of a descendent of T.
- An *independent table* is neither a parent nor a dependent.

Foreign Keys

A *foreign key* consists of one or more columns in the dependent table that together must either take on a value that exists in the primary key of the parent table, or be a null foreign key. When a row is updated or inserted into a dependent table, each non-null foreign key insert or update value must match a value of the corresponding primary key in the parent table. There can be multiple foreign keys defined on a dependent table referencing the same or different parent tables. The columns in the key may be nullable. If any of the columns contain a null value, the foreign key value is considered null. If the foreign key value is not null, then it must match an existing primary key value in the referencing parent table.

Referential Constraints

A referential constraint consists of a foreign key, the identification of a table containing a primary key, a constraint name and rules that govern changes. A referential constraint requires that a value can exist in one table (the dependent table) only if it also exists in another table (the parent table). After referential constraints have been defined, the enforcement of the referential constraint is immediate and the insert, update and delete rules are enforced when the INSERT (or PUT), UPDATE, and DELETE statements are issued. See "ALTER TABLE" on page 157 and "CREATE TABLE" on page 219 for information on how to declare referential constraints.

Referential constraints can be specified when tables are defined, or they can be added later. If they are added later, the database manager checks the references in the existing data. You can either drop or deactivate referential constraints to load large volumes of data, for example. After you load your data, you must recreate or reactivate your referential constraints. See "Activating and Deactivating Keys" on page 17 for more information.

Delete Rules

In order to maintain referential integrity, delete rules are imposed on all relationships. Every relationship includes a delete rule that was implicitly or explicitly specified when the referential constraint was declared by creating a foreign key. The options are:

- RESTRICT

  The deletion of a parent row is restricted. No deletions are allowed on a parent row until it has no dependent rows. This is the default action if no option is specified when the foreign key is created.
- CASCADE

  The deletion of a row in the parent table will cause the deletion of any dependent rows in a dependent table. When a dependent row is deleted, if the dependent table is also a parent table, then the delete rule of the referential constraint applies in turn. Each referential constraint in which a table is a parent has its own delete rule, and all applicable delete rules determine the result of the delete operation. Consequently a row in the parent table cannot be deleted if the deletion cascades to any of its descendents that has a dependent row in a referential constraint with a delete rule of RESTRICT.
- SET NULL

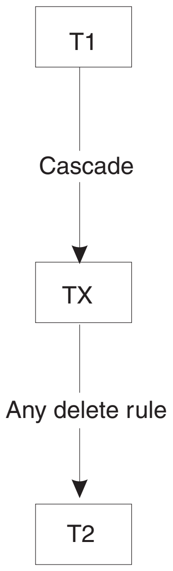
The deletion of a row in a parent table causes the corresponding values of the foreign key in any dependent rows to be set to null. Only nullable columns of the foreign key are set to null.

You may delete rows from a dependent table at any time, without taking any action on the parent table.

The following terminology applies to the delete rules.

- A table T2 is *delete-connected* to another table T1 if a delete of rows in table T1 can involve table T2. The following conditions determine whether tables are considered to be delete-connected:
  1. Dependent tables are always delete-connected to their parents irrespective of the delete rule.
  2. A table T2 is delete-connected to another table T1 if a delete of rows in table T1 can cause a delete of rows in T2's parent table(s). IBM-SQL[2] allows the concept of a self-referencing table, one that is delete-connected to itself. Note that DB2 Server for VSE & VM does not directly support a self-referencing table.

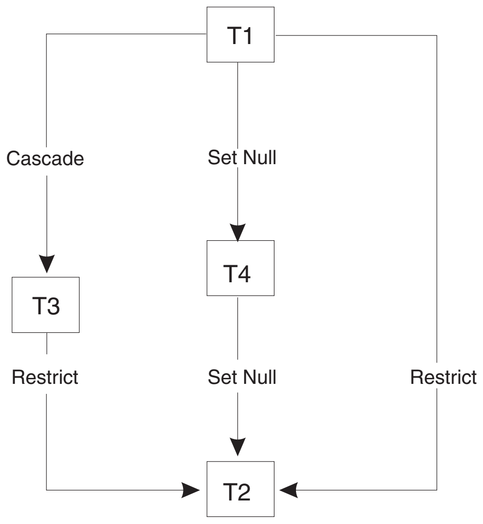
     In the relationship below, T2 and TX are both delete-connected to T1.



- A table T2 is *delete-connected through multiple paths* to table T1 if there is more than one relationship by which T2 is delete-connected to T1.

  In the diagram below, T2 is delete-connected to T1 with a delete rule of RESTRICT through two paths, one through T3 and a second by direct path. Note that T2 is not delete-connected to T1 along the path through T4 because a delete of rows in T1 will not cause a delete of rows in T4, which is T2's parent table along this path.

2. This is the term used to refer to IBM's SQL as published in the IBM SQL Reference, Volume 1 (SC26-8416)



- A *referential cycle* is a set of referential constraints for two or more tables such that each table in the set is a descendent of itself.
- A self-referencing table is a table that is a parent and dependent in the same referential constraint. The constraint is called a self-referencing constraint.

These relationships are depicted in the following example.

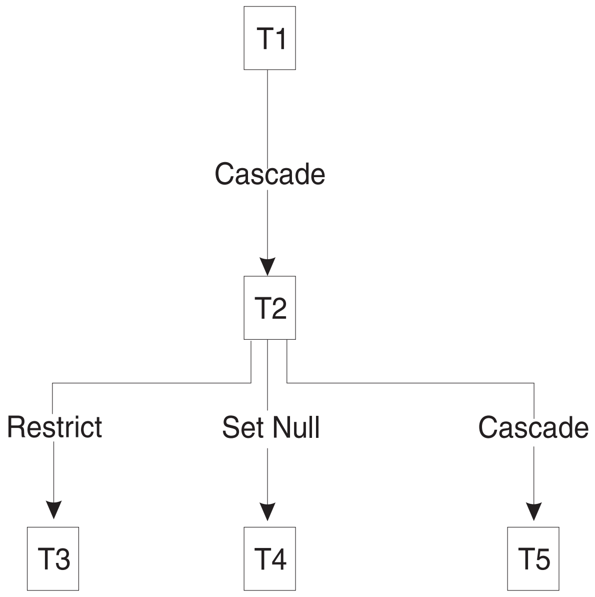


Table T3, T4 and T5 are dependents of table T2 which is a dependent of table T1. Since T1 is a parent table, the delete rule of referential constraint applies when a row of T1 is deleted. Specifically, deletion of a row in table T1 will cause all dependent rows in table T2 to be deleted. Since T2 is also a parent table, the delete rule of referential constraint also applies when a row of T2 is deleted. Specifically, the DELETE RESTRICT rule in table T3, the DELETE SET NULL rule in table T4 and the DELETE CASCADE rule in table T5 will apply. Note that if a row in T2 is to be deleted because its parent row in T1 is to be deleted, and this row has a dependent row in T3, then the entire delete operation will fail and will be rolled back.

DELETE Rule Restrictions

It is necessary to impose restrictions on referential constraint relationships to ensure that operations on delete-connected tables return consistent results with no dependence on a defined order of operations.

Definition Restrictions: The following restrictions are checked whenever a referential constraint is defined when a table is created or altered.

- If a table has more than one referential constraint referencing the same parent, all the delete rules on those constraints must be the same and must not be SET NULL.
- If a table is delete-connected to the same parent through multiple paths, all of the delete rules on a path, except for the last one, must be CASCADE. The last delete rule on all paths must be the same, and must not be SET NULL.
- A referential cycle involving two or more tables must not cause a table to be delete-connected to itself. In general, for a referential cycle involving n tables, where n >= 2, there can be at most n − 2 delete rules that are CASCADE. For example, in a referential cycle involving two tables, neither delete rule can be CASCADE. In a referential cycle involving four tables, at most two delete rules may be CASCADE.

Delete with Subquery Restrictions: The following restriction is enforced when a DELETE statement is prepared or preprocessed with a WHERE clause containing a subquery. If T2 is the object table of a DELETE statement, and T1 is referenced in a subquery of the WHERE clause, T1 must not be a table that can be affected by the DELETE on T2. The following example demonstrates the principle.

```
DELETE FROM T2 WHERE FIELD2 IN (SELECT FIELD1 FROM T1);
```

The following rules are enforced on tables T1 and T2.

1. T1 and T2 must not be the same table.
2. T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL.
3. T1 must not be a dependent of another table T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

Insert Rules

The database manager checks the implicit insert rules when a row is inserted into either a parent table or a dependent table in a referential structure.

When a row is inserted into a parent table, the database manager ensures that:

- The primary key is unique and does not contain a null value.

When a row is inserted into a dependent table, the database manager ensures that either:

- The foreign key has a matching primary key in the parent table, or
- The foreign key contains a null value in one or more of its columns.

Update Rules

When a key value is updated, the database manager checks the implicit update rules.

When a primary key is updated, the primary key must be unique and not null, and all dependent rows must be deleted or updated before the parent row can be updated.

When a foreign key is updated, it must have a matching primary key in the parent table or be a null key. A foreign key is considered null when any of its column values becomes null.

Activating and Deactivating Keys

After a referential constraint has been defined, referential integrity is immediately enforced and the primary and foreign keys are active. The database manager ensures that data integrity is maintained.

You may want to deactivate referential constraints, for example, to improve performance when loading large volumes of data. You can deactivate a table's primary key, any of its foreign keys, or a dependent foreign key. When any of these keys are deactivated, both the parent table and dependent table become unavailable to all users except the owner or someone possessing DBA authority.

After loading the data, referential constraints must be activated again. Activating them causes the database manager to validate the references in the data.

When the keys are reactivated, the referential constraints are automatically enforced once again. See "ALTER TABLE" on page 157 for more information on activating and deactivating keys.

Indexes

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

Indexes are used by the database manager to:

- Improve performance. In most cases, access to data is faster than without an index.
- Ensure uniqueness. A table with a unique index cannot have rows with identical keys.

Views

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

A view is a named specification of a result table. The specification is a SELECT statement that is effectively processed whenever the view is referenced in an SQL statement. Thus, a view can be thought of as having columns and rows just like a base table. For retrieval, all views can be used just like base tables. Whether a view can be used in an insert, update, or delete operation depends on its definition as explained in the description of CREATE VIEW. (See "CREATE VIEW" on page 231 for more information.)

An index cannot be created for a view. However, an index created for a table on which a view is based may improve the performance of operations on the view.

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

Packages

A *package* is an object that contains control structures (called sections) used to process SQL statements. Packages are produced during program preparation. The control structures can be thought of as the bound or operational form of SQL statements. All control structures in a package are derived from the SQL statements embedded in a single source program.

Catalog

The database manager maintains a set of tables containing information about the data it controls. These tables are collectively known as the *catalog*. The *catalog tables* contain information about objects such as tables, views, and indexes.

Tables in the catalog are like any other database tables. If you have authorization, you can use SQL statements to look at data in the catalog tables in the same way that you retrieve data from any other table. The database manager ensures that the catalog contains accurate descriptions of the relational database at all times.

Application Processes, Concurrency, and Recovery

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

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

The database manager acquires locks in order to prevent uncommitted changes made by one application process from being perceived by any other. The database manager will release all locks it has acquired on behalf of an application process when that process ends, but an application process itself can also explicitly request that locks be released sooner. This operation is called *commit*.

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

A *logical unit of work* (LUW), also known as a *unit of work*, is a recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but during the life of an application process there may be many recovery operations performed as a result of the commit or rollback operations.

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

A lock acquired by the database manager on behalf of an application process is held until its associated recovery operation has passed.

The initiation and termination of a unit of work define *points of consistency* within an application process. For example, a banking transaction might involve the transfer of funds from one account to another. Such a transaction would require that these funds be subtracted from the first account, and added to the second. 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 terminate the unit of work, thereby making the changes available to other application processes.

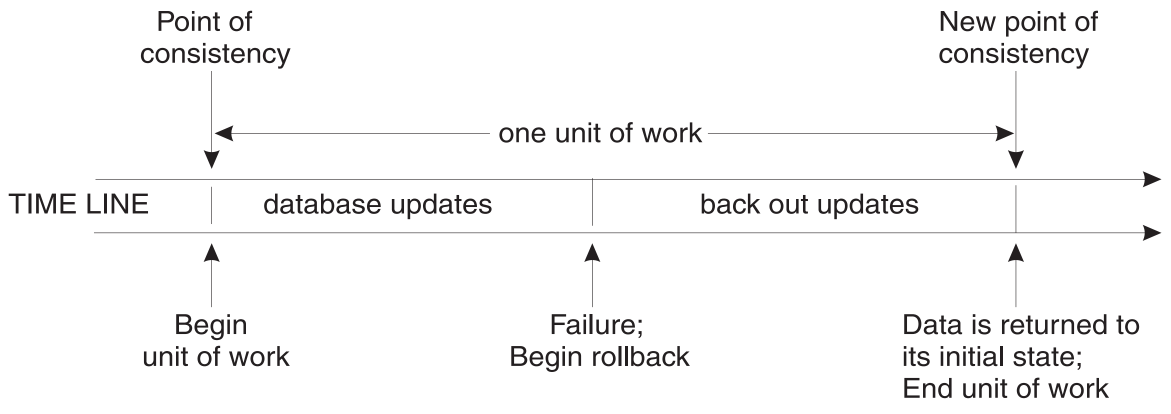


Figure 1. Unit of Work with a Commit Statement

If a problem occurs before the unit of work terminates, the database manager will back out uncommitted changes in order to restore the consistency of the data that it assumes existed when the unit of work was initiated.

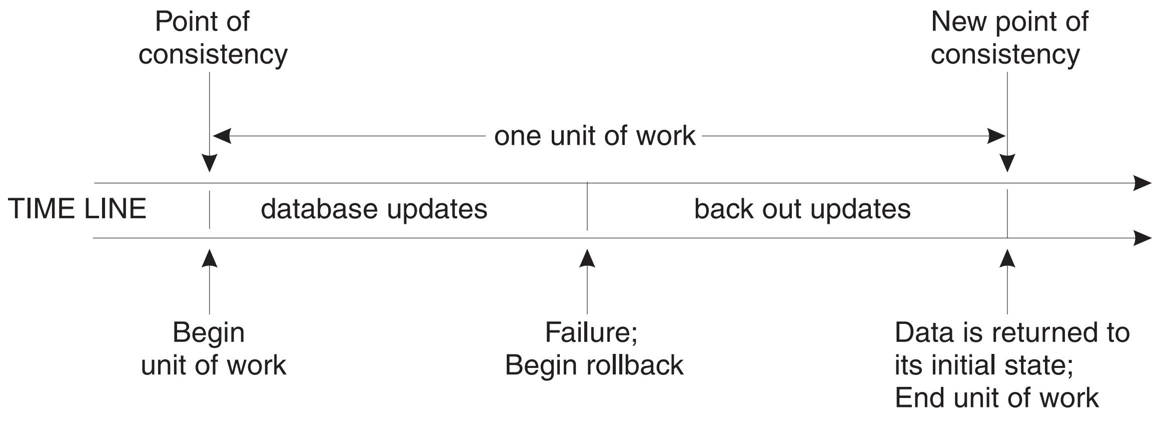


Figure 2. Unit of Work with a Rollback Statement

Cursor operations within a single unit of work are not protected from the result of other operations within the same unit of work. One example is a DELETE statement that deletes a row selected by a previous OPEN statement. Another example is two concurrently open cursors (at least one of which is updateable) operating on some of the same data.

Isolation Level

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

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

Isolation level is specified as an attribute of a package and applies to the application processes that use the package. The database manager provides a means of specifying an isolation level of a package through the program preparation process. The isolation levels are supported by automatically locking the appropriate data. Depending on the type of lock, this limits or prevents access to the data by concurrent application processes. The DB2 Server for VSE & VM database manager supports three types 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. This lock expresses an intent to possibly update the data. If the data is updated, the database manager upgrades the lock to an exclusive lock.

Exclusive
Prevents concurrent application processes from accessing the data in any way.

The following descriptions of isolation levels refer to locking data in row units. Data can be locked in larger physical units than base table rows. However, logically, locking occurs at least at the base table row level. Similarly, the database manager can escalate a lock to a higher level. An application process is guaranteed at least the minimum requested lock level.

The DB2 Server for VSE & VM database manager supports three isolation levels. Other database managers support additional levels (see the *IBM SQL Reference* for details of these additional levels). Regardless of the isolation level, it places exclusive locks on every row that is inserted, updated, or deleted. Thus, all isolation levels ensure that any row that is changed during a unit of work is not accessed by any other application (except for those using an isolation level of UR) until the unit of work is complete. The isolation levels are:

Repeatable Read (RR)

Level RR ensures that:

- Any row that is read during a unit of work is not changed by other application processes until the unit of work is complete.
- Any row that was changed by another application process cannot be read until it is committed by that application process.

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

Cursor Stability (CS)

Like level RR, level CS ensures that:

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

Unlike RR:

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

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

  1. Application process P1 reads the set of rows *n* that satisfy some search condition.
  2. Application process P2 then INSERTs one or more rows that satisfy the search condition and COMMITs those INSERTs.
  3. P1 reads the set of rows again with the same search condition and obtains both the original rows and the rows inserted by P2.
- CS only ensures that the current row of every cursor is not changed by other application processes. Thus, the rows that were read during a unit of work can be changed by other application processes.

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

Uncommitted Read (UR)

Unlike CS or RR, level UR allows:

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

Level UR allows an application to access most uncommitted changes of other applications. However, tables, views and indexes that are being created or dropped by other applications are not available while the application is processing. Any other changes by other applications can be read before they are committed or rolled back.

Non-read-only statements under level UR will behave as if the isolation level were cursor stability.

Like CS, UR does not completely isolate the application process from the effects of concurrent application processes. At level UR, application processes that run the same query more than once might see phantom rows, or may experience *nonrepeatable reads*.

For example, a nonrepeatable read can occur in the following situation:

1. Application process P1 reads the row from the database, then goes on to process other SQL requests.
2. Application process P2 either modifies or deletes the row and COMMITs the change.
3. P1 attempts to read the original row again, and either receives the modified row, or discovers that the original row has been deleted.

An application process running at level UR does not require any share locks.

Isolation Level Restrictions

Isolation levels, Cursor Stability and Uncommitted Read, only apply to Public dbspaces with ROW or PAGE level locking. Private dbspaces and Public dbspaces with DBSPACE level locking always use Repeatable Read isolation level. Data definition statements, such as CREATE, ACQUIRE or GRANT, and any statements that access the System Catalogs are always executed with Repeatable Read, regardless of the isolation level specified.

Isolation Level Escalation

Another relational database manager may request the DB2 Server for VSE & VM database manager to perform an operation on a DB2 Server for VSE & VM database (see “Application Requesters and Application Servers”). If the request specifies an isolation level other than one supported by the DB2 Server for VSE & VM database manager, the level is changed accordingly:
- Read Stability (RS) is changed to level RR

For more information on these isolation levels, see the *IBM SQL Reference.* Lock level escalation is discussed in the chapter on preprocessing and running programs in the *DB2 Server for VSE & VM Application Programming* manual.

Program Control of Isolation

The DB2 Server for VSE & VM database manager supports a facility that allows programs to dynamically modify the isolation level. The fact that a program will use this facility is indicated by the specification of the value USER instead of a specific isolation level when the program is prepared. Refer to the *DB2 Server for VSE & VM Application Programming* manual for details on how to do this.

Application Requesters and Application Servers

Application requesters and application servers work together to provide data to an application, regardless of where that data is located. The *application requester* accepts a database request from an application and passes it to an application server. In a distributed relational database, it transforms a database request from the application into communication protocols suitable for use in a distributed database network. The *application server* receives and processes the requests sent by the application requester.

Note: An application requester is sometimes called a user machine in VM and a user partition in VSE. An application server is sometimes called a database machine in VM and a database partition in VSE.

In this example, an application requester in Rochester is requesting data from an application server in Toronto.

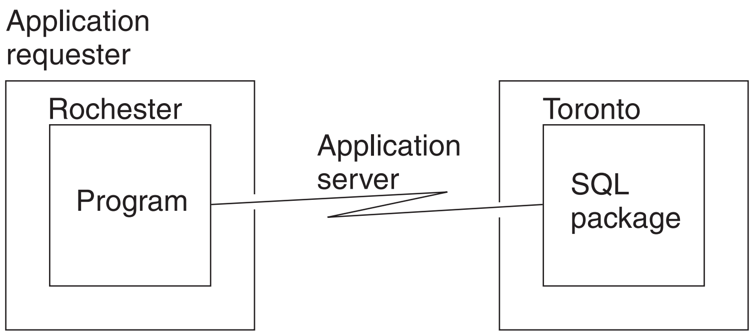


Figure 3. Requesting and Receiving Data Between an Application Requester and Application Server

An application process must be connected to the application server facility of a database manager before SQL statements that reference tables or views can be processed. A CONNECT statement establishes a connection between an application process and its server. VM and CICS/VSE applications may also use an implicit connection, in which case an explicit CONNECT statement is not necessary. An application process has only one server at any time, but the server can change when a CONNECT statement is processed.

Distributed Relational Database

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

The following diagram shows how data is requested and transmitted between two relational database systems participating in a complete Distributed Relational Database Architecture (DRDA) relationship. Each of the two systems may request data from the other.

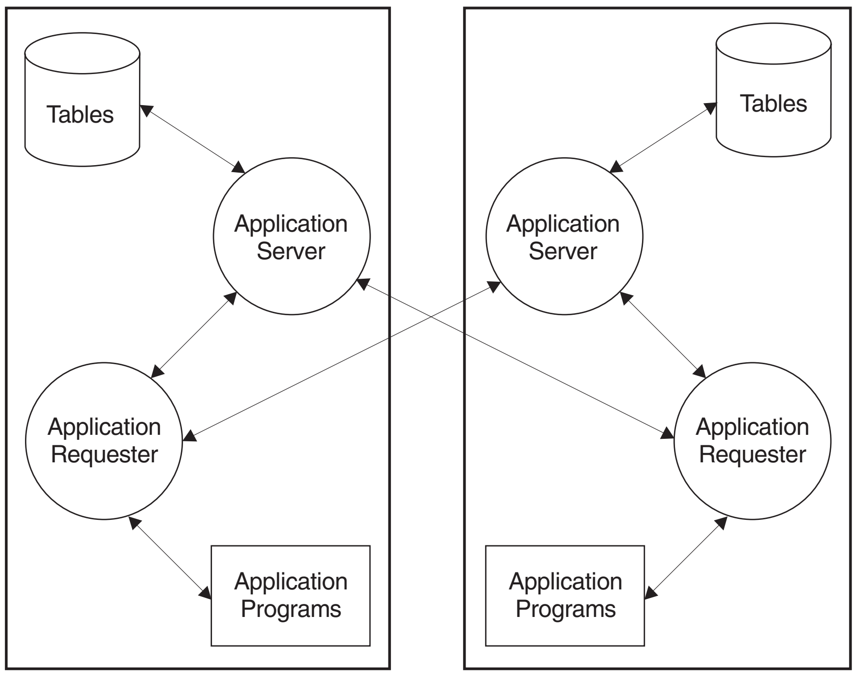


Figure 4. Requester/Server Data Flow

The following diagram shows supported IBM relational database DRDA connections; AIX connections are the same as those for OS/2. Incoming arrows indicate application server support; outgoing arrows indicate application requester support. Note that the relationships displayed are for unlike IBM systems only. Some of the systems shown provide other protocols for like-system connections.

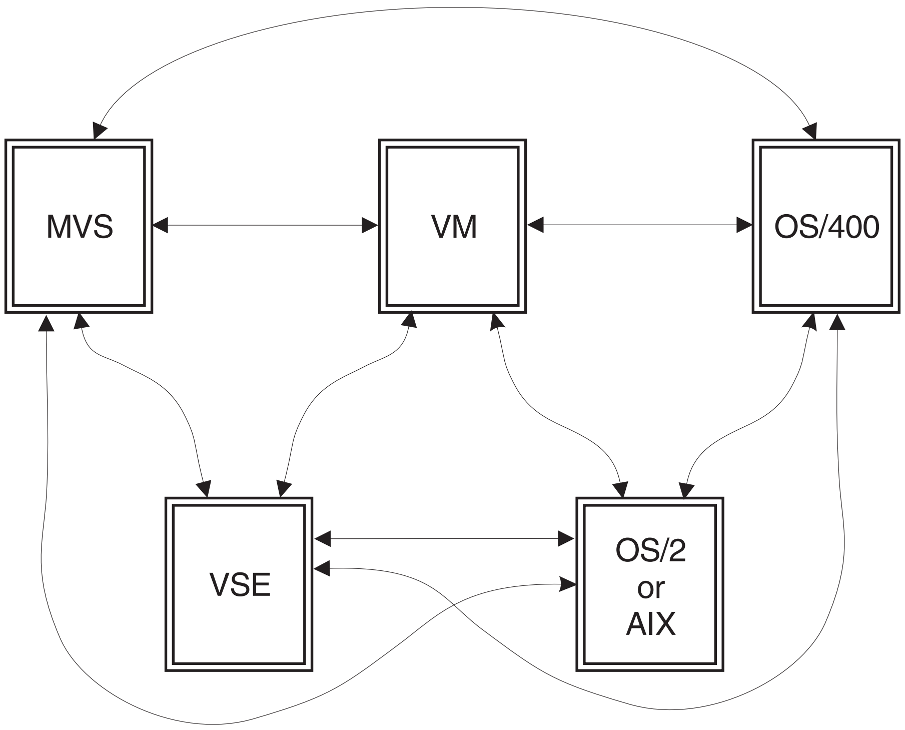


Figure 5. IBM Relational Database Connections

Distributed relational databases are built on formal requester-server protocols and functions. Working together, the application requester and application server handle the communication and location considerations so that the application is isolated from these considerations and can operate as if it were accessing a local database. DB2 Server for VM supports application servers and application requesters for DRDA communication protocols. DB2 Server for VSE supports application servers and application requesters for online CICS/VSE application programs for DRDA. DB2 Server for VSE also provides Remote Unit of Work (RUOW) application requester support for batch applications. For more information on DRDA communication protocols, see the *Distributed Relational Database Architecture Reference*.

Two communication protocols can be used by the DB2 Server for VSE & VM database manager. These protocols allow the data to be used within distributed relational databases or as a non-distributed relational database. The two protocols are:

SQLDS	A protocol for a DB2 Server for VSE & VM database manager to communicate with other *like* database managers.
DRDA	A protocol for communicating with both *like* and *unlike* database managers.

The following table shows the protocols that are used between DB2 Server for VSE & VM application requesters and application servers.

Application Requester	**Communication Protocol**	**Application Server**
DB2 for VSE	SQLDS	DB2 Server for VM Used for guest sharing.
DB2 for VSE	SQLDS	DB2 Server for VSE
DB2 for VSE	DRDA	DB2 Server for VSE
DB2 for VSE	DRDA	DB2 Server for VM
DB2 for VSE	DRDA	DB2 for MVS
DB2 for VSE	DRDA	DB2 for OS/400
DB2 for VSE	DRDA	DB2 for OS/2
DB2 for VSE	DRDA	DB2 for AIX
DB2 for VM	SQLDS	DB2 Server for VM
DB2 for VM	DRDA	DB2 Server for VM
DB2 for VM	DRDA	DB2 Server for VSE
DB2 for VM	DRDA	DB2 for MVS
DB2 for VM	DRDA	DB2 for OS/400
DB2 for VM	DRDA	DB2 for OS/2
DB2 for VM	DRDA	DB2 for AIX

For more information on the communication protocols, see the *DB2 Server for VSE & VM Performance Tuning Handbook*.

Application Servers in DRDA

The application server can be local to or remote from the environment where the process is initiated. This environment includes a local directory that describes the application servers that can be identified in a CONNECT statement. The format and maintenance of this directory are described in the "Network Information" sections for each SQL product in the *Distributed Relational Database Connectivity Guide* manual.

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

Data managed by any remote application server that implements the DRDA architecture can be accessed and manipulated by VSE Batch application programs that have the ability to execute SQL statements.

Remote Unit of Work

A remote unit of work (RUOW) is a logical unit of work that allows 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 logical units of work, process any number of static or dynamic SQL statements that reference objects at B. After terminating a unit of work at B, the application process can connect to an application server at computer system C, and so on.

The DB2 Server for VM requester can remotely prepare and run most SQL statements given the following conditions:

- All objects referenced in a single SQL statement are managed by the same application server.
- All of the SQL statements in a unit of work are processed by the same application server.

The DB2 Server for VSE requester can remotely prepare and run most SQL statements given the following conditions:

- The Online Resource Adapter must be enabled.
- The remote server must be known to the Online Resource Adapter.

DB2 Server for VSE also provides DRDA support that consists of remote unit of work (RUOW) Application Requester (AR) support for Batch applications.

Distributed Unit of Work

A distributed unit of work (DUOW) is a logical unit of work that allows a user or application program to read or update data at multiple locations. An application process at computer system A can connect to an application server at computer system B, process static or dynamic SQL statements that reference objects at B, then connect to an application server at computer system C, and process SQL statements that reference objects at C, and so on, before terminating the unit of work. Each SQL statement can access one application server. Commits and rollbacks are coordinated at all locations so that if a failure occurs anywhere in the system, data integrity is preserved.

The Use of DB2 Family SQL on Various Application Servers

This section will mainly be of interest to people who are writing applications that are:

- DB2 Server for VSE CICS applications, or are to be run on DB2 Server for VM application requesters **and**

- accessing data that is controlled by the application servers of one or more *unlike* relational database managers.

The section will also be of interest to people with the opposite requirement (for instance a DB2 for OS/2 application requester connected to a DB2 Server for VM or DB2 Server for VSE application server).

The DB2 family's support of SQL is a superset of SQL92 Entry Level (SQL92E). [3] Not all DB2 family members support all elements of SQL. For a complete discussion of the individual family members' support of SQL, please see the *IBM SQL Reference, Version 2, Volume 1.*

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

There are some restrictions that apply. Due to the different availability dates of the IBM relational database products, it is not possible to provide a complete list of these. The rest of this section will, therefore, outline some general guidelines that govern the inter-operability of statements and provide examples of statements that can and cannot be used among products.

- All Data Definition and Authorization statements that are supported by an application server can be issued from any application requester.

  Example: A CICS application running as a DB2 Server for VSE application requester connected to a DB2 for MVS application server may use a CREATE TABLESPACE statement. Similarly, an application running on a DB2 for MVS application requester connected to a DB2 Server for VM or DB2 Server for VSE application server may use an ACQUIRE DBSPACE statement.
- Most other statements that do not contain any host variables can be issued from any application requester.

  Example: An application running on a DB2 Server for VM application requester connected to a DB2 for MVS application server may issue the following statement even though the DB2 Server for VM database manager does not support the WITH HOLD clause.

  ```
  EXEC SQL  DECLARE PRIMARY_CURSOR CURSOR WITH HOLD
              FOR SELECT_COURSES;
  ```
- Some statements without host variables are not sent to the application server; rather, they are processed completely by the application requester. Such statements may only be used on application requesters of products that support statements.

  Example 1: An application running on a DB2 Server for VM application requester cannot issue the DECLARE STATEMENT statement against a DB2 for MVS application server.

3. SQL92E is the term used to refer to the combination of the following standards:
    ISO (International Standards Organization) 9075-1992(E)
    ANSI (American National Standard for Information Systems) X3.135-1992
    FIPS (Federal Information Processing Standards) publication 127-2.

The above documents list more than one level of conformance. The levels are Entry, Transitional (FIPS only), Intermediate, and Full SQL. We are concerned with Entry SQL and we designate that with the abbreviation SQL92E.

Example 2: An application running on a DB2 Server for VM application requester cannot issue the DECLARE VARIABLE statement against a DB2 for OS/400 application server.

- Some statements without host variables are the joint responsibility of the application requester and application server. Such statements must be fully understood by the application requester.

  Example: An application running on a DB2 Server for VM application requester connected to a DB2 for OS/400 application server cannot issue the following statement, because the PRIOR clause is not supported by the DB2 Server for VM database manager.

  ```
  EXEC SQL FETCH PRIOR FROM PAGE_CURSOR;
  ```
- If a statement or clause contains host variables and an application requester does not understand that statement or clause, those host variables are assumed to be input host variables. If this is not a valid assumption, the application server will reject the statement.

  Example 1: An application running on a DB2 Server for VM application requester could issue the following statement to a DB2 for MVS application server:

  ```
    EXEC SQL  SET CURRENT SQLID = :CUR_USER;
  ```

  because :CUR_USER references an input host variable.

  Example 2: However, an application running on a DB2 Server for VM application requester could not issue the following statement to a DB2 for MVS application server:

  ```
    EXEC SQL  SET :TIME_UPDATED = CURRENT TIME;
  ```

  because :TIME_UPDATED references an output host variable.
- Only DB2 Server for VSE & VM supports Extended Dynamic SQL. However, an application on a DB2 Server for VM application requester or a DB2 Server for VSE CICS application requester can issue most extended dynamic statements for non-modifiable packages against unlike application servers. A list of restrictions can be found in Appendix G, "DRDA Considerations," on page 425.
- Only DB2 Server for VSE & VM supports Insert Cursors. However, an application on a DB2 Server for VM application requester or a DB2 Server for VSE CICS application requester can declare Insert Cursors and issue PUT statements against unlike application servers. Note that there is no blocking of input data because the application requester turns PUT statements into INSERT statements. The purpose of this support is to allow an application to run without having to make this change in the source program.
- In IBM-SQL, all objects (that is, tables, views, indexes, and packages) have a two-part name. DB2 for MVS application servers also support three-part names for tables, views, and *aliases* (a non-IBM-SQL object). The high order part of the name identifies an application server (also called a *location* in DB2 for MVS). In addition to the heterogeneous remote unit of work facility using DRDA protocols among unlike products, DB2 for MVS supports a homogeneous *distributed unit of work* facility using private protocols. This facility makes use of three-part names in order to allow statements within the same unit of work to be issued against different application servers as long as data is only modified at one of those application servers.

  For example, an application which is run on a DB2 Server for VM application requester can issue the following statements in order to read data controlled by DB2 for MVS application servers at Halifax, Montreal and Toronto and use the information obtained there to update a DB2 for MVS table at Halifax.

```
EXEC SQL  CONNECT TO HALIFAX;

EXEC SQL  SELECT SUM(WEEKLY_NET)       -- processed by DB2 at Halifax
            INTO :TOT3
            FROM FORCASTING.SALES;

EXEC SQL  SELECT SUM(WEEKLY_NET)       -- routed by DB2 at Halifax to be
                                       -- processed by DB2 at Montreal
            INTO :TOT1
            FROM MONTREAL.FORCASTING.SALES;

EXEC SQL  SELECT SUM(WEEKLY_NET)       -- routed by DB2 at Halifax to be
                                       -- processed by DB2 at Toronto
            INTO :TOT2
            FROM TORONTO.FORCASTING.SALES;

TOT = TOT1 + TOT2 + TOT3;

EXEC SQL  UPDATE FORCASTING.TOTALS     -- processed by DB2 at Halifax
            SET WEEKLY_NET = :TOT;
```

DRDA protocols are used in the communications between the application requester and the application server at Halifax. Private DB2 for MVS protocols are used in the communications between Halifax and Toronto as well as the communications between Halifax and Montreal. Three-part names allow statements within the same unit of work to be issued against different application servers.

For more information on distributed unit of work, refer to the DB2 for MVS library.

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 will automatically perform any necessary conversions at the receiving system.

With numeric data, the information needed to perform the conversion is the data type of the data and how that data type is represented by the sending system. For example, when a floating point variable from an OS/400 application requester is assigned to a column of a table at a VM application server, the DB2 Server for VM database manager, knowing the data type and the sending system, converts the number from IEEE format to S/390 format.

With character data, additional information is needed to convert character strings. String conversion depends on both the coded character set of the data and the operation that is to be performed with that data. Character conversions are performed in accordance with the IBM Character Data Representation Architecture (CDRA). For more information on character conversion, refer to *Character Data Representation Architecture Reference and Registry.*

Character Conversion

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

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

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

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

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

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

code page
: A set of assignments of characters to code points. In EBCDIC, for example, "A" is assigned code point X'C1' and "B" is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

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

coded character set
: A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

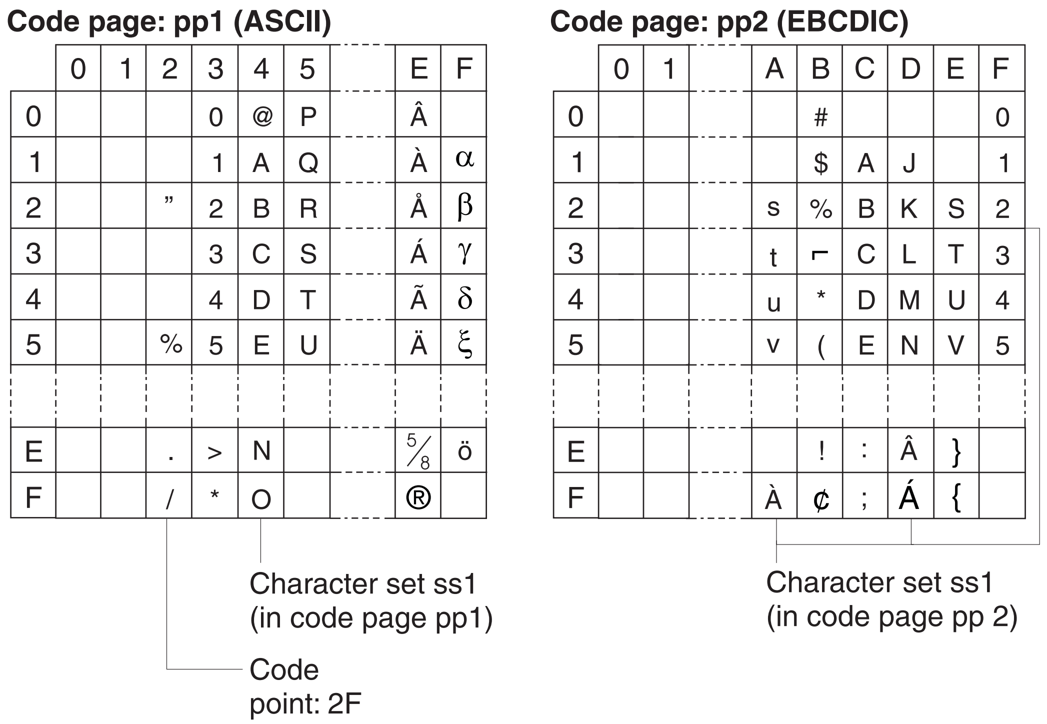
encoding scheme
: A set of rules used to represent character data. For example:
  - Single-Byte EBCDIC
  - Single-Byte ASCII (see Note)
  - Double-Byte EBCDIC
  - Mixed Single-, Double- and Multi-Byte ASCII.

  Note: Single-Byte ASCII is an encoding scheme used to represent strings in many environments, including OS/2. In the OS/2 environment, ASCII refers to the PC Data encoding scheme.

substitution character
: A unique character that is substituted during character conversion for any characters in the source coding representation that do not have a match in the target coding representation.

Character Sets and Code Pages

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


Code page: pp1 (ASCII)
Code page: pp2 (EBCDIC)
Character set ss1 (in code page pp1)
Code point: 2F
Character set ss1 (in code page pp 2)

Even with the same encoding scheme, there are many different coded character sets, and the same code point can represent a different character in different coded character sets. 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 characters, double-byte characters and possibly multi-byte characters. On some platforms, mixed data may be comprised of 2, 3, 4 or more bytes. *Bit* data is not associated with any character set. Note that this is not the case with graphic strings; the database manager assumes that every pair of bytes in every graphic string represents a character from a double-byte character set (DBCS).

For more details on character conversion, see:
- "Conversion Rules for String Comparison" on page 58 for rules on string comparisons
- "Conversion Rules for Operations that Combine Strings" on page 130 for rules on concatenation.

Coded Character Sets and CCSIDs

IBM's Character Data Representation Architecture (CDRA) deals with the differences in string representation and encoding. The *Coded Character Set Identifier* (CCSID) is a key element of this architecture. A CCSID is a 2-byte (unsigned) binary number that uniquely identifies an encoding scheme and one or more pairs of character sets and code pages.

A CCSID is an attribute of strings, just as a length is an attribute of strings. In DB2 Server for VSE & VM databases, different columns can have different CCSID attributes and each string in a column has the CCSID attribute of that column. Note that not all IBM relational database managers support the specification of CCSIDs at the column level.

Character conversion involves the use of a *CCSID Conversion Selection Table*. The Conversion Selection Table, which is stored in the SYSTEM.SYSSTRINGS catalog table, contains a list of valid source and target combinations. For each pair of CCSIDs, the Conversion Selection Table contains information used to perform the conversion from one coded character set to the other. This information includes an indication of whether conversion is required. (In some cases, no conversion is necessary even though the strings involved have different CCSIDs.)

Default CCSID

Every application server and application requester has a default CCSID (or default CCSIDs in installations that support DBCS data). A list of the CCSIDs supported by the DB2 Server for VSE & VM database manager can be found in the CCSID column of the SYSTEM.SYSCCSIDS catalog table.

A *default CCSID* is the CCSID of the default subtype; for example, the USER special register could have either an SBCS or mixed subtype, and its default CCSID would be the subtype's CCSID specified by the application server. The CCSID of the following types of strings is determined at the application server:
- String constants
- Special registers with string values (such as USER and CURRENT SERVER)
- The result of the scalar functions CHAR, DIGITS, HEX, and VARGRAPHIC
- String columns defined by CREATE TABLE and ALTER TABLE statements
- The character representation of datetime values.

The default CCSID of strings stored in host variables is determined at the application requester.

Statements are converted from the default CCSID of the application requester to the default CCSID of the application server.

In VM: When an application server or application requester is initialized with PROTOCOL=SQLDS (see SQLSTART and SQLINIT in the *DB2 Server for VSE & VM Database Administration* manual), the default CCSID of the application requester is the same as that of the application server regardless of the values reported on the application requester when an SQLINIT QUERY is performed.

In VSE: When an application requester accesses a local application server, the default CCSID of the application requester is the same as that of the application server, regardless of the values reported on the application requester when a DSQQ transaction is performed.

Authorization and Privileges

Users can successfully process SQL statements only if they are authorized to perform the specified function. To create a table, a user must be authorized to create tables; to alter a table, a user must be authorized to alter the table; and so forth.

Two forms of permission exist:
- Authority to
  - connect to a specific database manager
  - allocate resources such as private dbspaces and public dbspaces

  - administer the database manager. This is called *DBA* (Database Administrator) *authority*.
- Privilege to
  - access objects in the database
  - create indexes on specific tables.

Authorization, then, refers to who is allowed to access what data, whereas *privileges* refer to how an authorized user can use the data. For example, authorized users can create, modify, and delete tables. These users then have privileges on those tables and can selectively grant and revoke those privileges to other users.

The person or persons holding DBA authority are charged with the task of controlling the database manager and are responsible for the safety and integrity of the data. Those with DBA authority control who will have access to the database manager and the extent of this access.

Chapter 3. Language Elements

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

Characters

The basic symbols of keywords and operators in the SQL language as supported by this product are single-byte EBCDIC characters. Characters of the language are classified as letters, digits, or special characters.

A *letter* is any one of the uppercase characters A through Z, the lowercase letters a through z, plus the three characters reserved as alphabetic extenders for national languages (for example, in code page 37, $ is at 5B, # is at 7B, @ is at 7C).

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

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

	space	/	slash
"	quote or double-quote	:	colon
%	percent	;	semi-colon
&	ampersand	<	less than
'	apostrophe or single quote	=	equals
(	left parenthesis	>	greater than
)	right parenthesis	?	question mark
*	asterisk	_	underscore
+	plus sign	¬	logical NOT *
,	comma	^	caret *
–	minus sign	|	vertical bar *
.	period	!	exclamation point *

* Not supported in IBM-SQL. For portability of programs, consider alternative characters.

A character set composed of characters not found in the default U.S. English EBCDIC character set can be created (thus, for instance, expanding the set of characters defined as letters). Such a character set could be useful in folding, which converts lowercase characters of a character string into uppercase. For information on how to create these characters, refer to the section on defining your own character set in the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual. Also, see "SYSCHARSETS" on page 378.

Tokens

The basic syntactic units of the language are called *tokens*. A token consists of one or more characters, excluding the blank character, and excluding characters within a string constant (for example 'string') or delimited identifier (for example "field1"). These terms are defined later.

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

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

  Examples:

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

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark (?) is also a delimiter token when it serves as a parameter marker, as explained under "PREPARE" on page 313.

  Examples:

  ```
  'string'      "field1"      =     ,     .
  ```

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 can be followed by a space. Every ordinary token must be followed by a delimiter token or a space. If the syntax does not allow an ordinary token to be followed by a delimiter token, that ordinary token must be followed by a space.

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 terminated by the end of the line. For more information, see "SQL Comments" on page 143.

Uppercase and Lowercase

Letters used in an ordinary token other than a C variable must be uppercase letters. Thus, lowercase letters can only be used in string constants, delimited identifiers, and C language host variables. In all other tokens, the lowercase letters will be folded to uppercase.

Identifiers

An *identifier* is a token 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 identifiers* and *delimited identifiers*.

- An *ordinary identifier* is an uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. The database manager “folds” lowercase characters in ordinary identifiers to uppercase (see “Characters” on page 35 for a complete definition of a letter). An ordinary identifier must not be a reserved word. (See the back cover for a list of DB2 Server for VSE & VM reserved words.)

  Most ordinary identifiers may include DBCS characters if the following support is true:
  - For a VM application requester, the CHARNAME parameter of the SQLINIT EXEC must identify a mixed character set; at the application server, the SYSTEM.SYSOPTIONS catalog must have the DBCS option set to YES and the CHARNAME option must be a mixed CHARNAME
  - For VSE, the CHARNAME option in the SYSTEM.SYSOPTIONS catalog table must be a mixed CHARNAME; at the application server, the SYSTEM.SYSOPTIONS catalog must have the DBCS option set to YES and the CHARNAME option must be a mixed CHARNAME

  Note: The ordinary identifiers that cannot include DBCS characters are identified in “Naming Conventions” on page 38.

  See “SYSOPTIONS” on page 393 for more information on the DBCS and CHARNAME options.
- A *delimited identifier* is a sequence of characters enclosed within quotation marks ("). Any character except a quote (") may be used between the quotation marks; however, leading blanks and trailing blanks are not allowed, and periods (.) should be avoided. Lowercase characters are not folded to uppercase, the identifier need not start with a letter, and the value of a delimited identifier may be the same as a reserved word.

Examples:

```
WKLYSAL     WKLY_SAL     "WKLY_SAL"    "UNION"

"wkly sal"  "wkly_sal"   "Dave's Table"

Note that "Dave"s Table" is incorrect.
```

SQL identifiers are also classified according to their maximum length. A *long identifier* has a maximum length of 18 bytes. A *short identifier* has a maximum length of 8 bytes. The length of either a long or short identifier does not include quotation marks around delimited identifiers.

Host Identifiers

A *host_identifier* is a name declared in a host program. The rules for forming a *host_identifier* are the rules of the host language. In addition, a *host_identifier:*

- has a maximum length of 18 characters in Fortran
- in IBM VM systems, host identifiers can contain DBCS characters if the SQLINIT EXEC is invoked with DBCS set to YES and the CHARNAME parameter specifies a mixed character set. Values in the SYSTEM.SYSOPTIONS table do not affect the use of DBCS characters in host identifiers.
- in VSE, can include DBCS characters if, in the SYSTEM.SYSOPTIONS catalog table at the application server, the CHARNAME option specifies a mixed character set and the DBCS option is set to YES.
- should not begin with SQL or RDI
- should not begin with SQ for Fortran.

Naming Conventions

The rules for forming a name depend on the type of the object designated by the name. Though names may include any character, it is not advisable to use special characters, such as #, @, or $, with the DRDA protocol. Special characters may not be supported by all code pages (see "Character Conversion" on page 29 for more information on code pages).

The syntax diagrams use the metavariables in the following list to represent actual object names and values. The list does not include all metavariables used; there are also metavariables whose scope is limited to a particular diagram and these are defined locally.

authorization_name	A short identifier that designates a user or group of users. It must only include SBCS characters. See "Authorization IDs and Authorization-names" on page 41.
collection_id	A short identifier that provides a logical grouping for SQL objects. A *collection_id* used as the qualifier of the name of a table, view, index, or package is an *authorization_name. Collection_id* is the same as *owner_name.* It must not include DBCS characters.
column_name	A qualified or unqualified name that designates a column of a table or view. The unqualified form of a *column_name* is a long identifier. The qualified form is a qualifier followed by a period and a long identifier. The qualifier is a table name, a view name, a synonym, or a correlation name.
constraint_name	A long identifier that designates a referential constraint on a table.
correlation_name	A long identifier that designates a table, a view, or individual rows of a table or view.
cursor_name	In the Positioned UPDATE and Positioned DELETE statements: a long identifier that designates an SQL cursor. In all other statements: a long, ordinary identifier that designates an SQL cursor. *Cursor_names*, in these statements, unlike other ordinary identifiers, can be SQL reserved words (though the use of reserved words in ordinary identifiers is not recommended because that usage is not supported in either IBM-SQL or ISO/ANSI SQL). It must not include DBCS characters.
cursor_variable	A *host_identifier* used to name or identify a cursor in an extended dynamic statement. The *host_identifier*'s attribute must be VARCHAR and its length attribute must be 18.
dbspace_name	A qualified or unqualified name that designates a dbspace. The unqualified form of a *dbspace_name* is a long identifier. An unqualified *dbspace_name* in an SQL statement is implicitly qualified by the

authorization ID of that statement. The qualified form is a *collection_id* followed by a period and a long identifier.

descriptor_name
A *host_identifier*, optionally preceded by a colon, that designates an SQL descriptor area (SQLDA). Note that a *descriptor_name* never includes an indicator variable.

host_variable
A sequence of tokens that designates a host variable. A *host_variable* includes at least one *host_identifier*, as explained in “Host Identifiers” on page 37.

host_variable_list
A list of one or more host variables, host structures, or both, which takes the following form:

```
     ┌─,─────────────┐
     ▼               │
►►───host_variable───┴─────────────────────────►◄
```

In this context, a *host_variable* can also reference a host structure. See “Host Structures and Indicator Arrays” on page 69 for more information on host structures.

index_id
An unqualified *index_name*. It is a long identifier.

index_name
A qualified or unqualified name that designates an index. The unqualified form of an *index_name* is a long identifier. An unqualified *index_name* in an SQL statement is implicitly qualified by the authorization ID of that statement. The qualified form is a *collection_id* followed by a period and a long identifier.

owner_name
A short identifier that designates an owner of a database object such as a table, view, index, or package. *Owner_name* is the same as *collection_id*. It must not include DBCS characters.

package_id
An unqualified *package_name*. It is a short ordinary identifier. It must not include DBCS characters.

package_name
A qualified or unqualified name that designates a package. The unqualified form of a *package_name* is a short ordinary identifier. An unqualified *package_name* in an SQL statement is implicitly qualified by the authorization ID of that statement. The qualified form is a collection_id followed by a period and a short ordinary identifier.

package_spec
A metavariable used to name or identify packages within extended dynamic statements.

```
►►──┬─────────────────────┬────────────────────►
    ├─collection_id──.────┤
    └─host_identifier──.──┘
```

```
►►──┬─package_id──────┬──────────────────────────►◄
    └─host_identifier─┘
```

If a *host_identifier* is used it must be CHAR(8) and, if the name in the *host_identifier* is less than 8 characters, it must be padded to the right with blanks.

In C, the host variable must have a datatype of C NUL-terminated and a length of 9.

password	A short ordinary identifier that designates a password. It must not include DBCS characters.
section_variable	A *host_identifier* used to identify a statement that has been prepared into an extended dynamic package. The *section_variable's* data type must be INTEGER.
server_name	A long identifier that designates an application server. It must not include DBCS characters.
statement_name	A long ordinary identifier that designates a prepared SQL statement. *Statement_names*, unlike other ordinary identifiers, can be SQL reserved words (though the use of reserved words in ordinary identifiers is not recommended because that usage is not supported in either IBM-SQL or ISO/ANSI SQL). It must not include DBCS characters.
synonym	A long identifier that names a synonym, a table, or a view. *Synonym* is a different term to refer to a table or view that must exist at the current server. A *synonym* is named when it is preceded by the keyword SYNONYM; it names a local table or view when it is used in an SQL statement. A qualified name is never interpreted as a *synonym*.
table_id	An unqualified *table_name*. It is a long identifier.
table_name	A qualified or unqualified name that designates a table. The unqualified form of a *table_name* is a long identifier. An unqualified *table_name* in an SQL statement is implicitly qualified by the authorization ID of that statement. The qualified form is a *collection_id* followed by a period and a long identifier.
view_id	An unqualified view_name. It is a long identifier.
view_name	A qualified or unqualified name that designates a view. The unqualified form of a *view_name* is a long identifier. An unqualified *view_name* in an SQL statement is implicitly qualified by the authorization ID of that statement. The qualified form is a *collection_id* followed by a period and a long identifier.

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
- Implicit qualifiers for the names of tables, views, indexes, dbspaces, and packages.

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 preparation. The authorization ID that applies to a dynamic SQL statement is the authorization ID that was obtained by the database manager when a connection was established between the database manager and the process. This is called the *run-time authorization ID*.

An authorization-name specified in an SQL statement should not be confused with the authorization ID of the statement. An *authorization-name* is an identifier that is used in GRANT and REVOKE statements to designate a target of the grant or revoke. It cannot be identical to the authorization ID of the GRANT or REVOKE statement. Note that the premise of a grant of privileges to X is that X will subsequently be the authorization ID of statements which require those privileges.

Examples:

Example 1

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

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

SMITH is the authorization ID of the statement. Thus, the authority to process the statement is checked against SMITH and SMITH is the implicit qualifier of TDEPT.

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

Example 2

Assume SMITH has administrative authority and is the authorization ID of the following statements:

```
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.

Data Types

For information about specifying the data types of columns, see “CREATE TABLE” on page 219.

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

Constants
Columns
Host variables
Functions
Expressions
Special registers.

Figure 6 illustrates the various data types supported by the database manager.

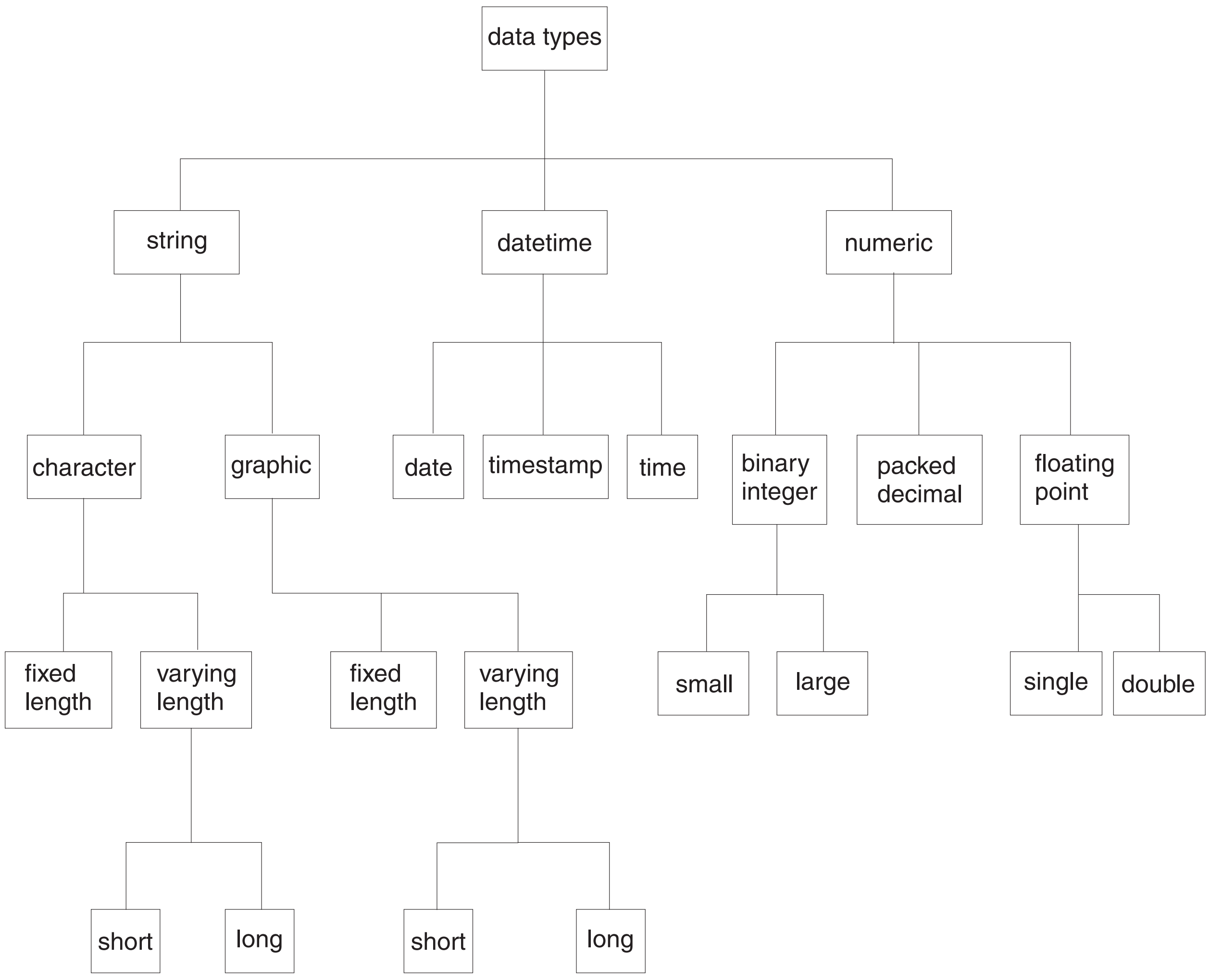


Figure 6. Data Types Supported by the Database Manager

Result Set Locators

This data type is used to identify host variables that are used by the DB2 Server for VSE & VM requester to uniquely indicate a query result set returned by a stored procedure. These host variables are called result set locator variables. They are only supported in client applications written in Assembler, C, COBOL, and PL/I. These variables should be included in the SQL DECLARE section and cannot be array variables.

The syntax used to declare a result set locator variable for each language follows:

RESULT SET LOCATOR

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


├──┬─┤ Assembler ├─┬─────────────────────────────────────────────────────────────┤
   ├─┤ C ├─────────┤
   ├─┤ COBOL ├─────┤
   └─┤ PL/I ├──────┘
```

Assembler:

```
├──variable-name──┬─DC─┬──F──────────────────────────────────────────────────────┤
                  └─DS─┘
```

C:

```
├──┬────────┬──┬──────────┬──────────────────────────────────────────────────────►
   ├─auto───┤  ├─const────┤
   ├─extern─┤  └─volatile─┘
   ├─static─┤
   └─_Packed┘

                                         ┌─,────────────────────────────┐
►──SQL TYPE IS RESULT_SET_LOCATOR VARYING──▼─variable-name──┬──────────────┬─┴──;──────┤
                                                        └─= init-value─┘
```

COBOL:

```
├──01──variable-name──SQL TYPE IS RESULT-SET-LOCATOR VARYING──.─────────────────┤
```

PL/I:

```
├──┬─DECLARE─┬──┬─variable-name───────────┬──────────────────────────────────────►
   └─DCL─────┘  │   ┌─,─────────────┐     │
                └─(───▼─variable-name─┴──)─┘

►──SQL TYPE IS RESULT_SET_LOCATOR VARYING──┬──────────────────────────────────────┬──;──────┤
                                           └─Alignment and/or Scope and/or Storage─┘
```

Nulls

All data types include the null value. The null value is a special value that is distinct from all non-null values; it denotes an unknown value. Although all data types include the null value, columns defined as NOT NULL cannot contain null values.

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*. The empty string should not be confused with the null value.

Fixed-Length Character Strings

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

Varying-Length Character Strings

The values of a varying-length string column can have different lengths. The maximum length is determined by the length attribute of the column. The length attribute must be between 1 and 32,767.

A varying-length character string column with a length attribute greater than 254 is a *long string column*; otherwise it is a *short string column*. A derived character string with a maximum length greater than 254 is a *long string*; otherwise it is a *short string*. Long strings and long string columns cannot be referenced in:
- A function other than SUBSTR or LENGTH. (In SUBSTR and LENGTH, the argument may be a long string column but it may not be a long string host variable.)
- A GROUP BY clause
- An ORDER BY clause
- A CREATE INDEX statement
- A PRIMARY KEY, FOREIGN KEY, or UNIQUE clause
- A SELECT DISTINCT statement's select list
- A subselect of a UNION or UNION ALL
- A subselect of an INSERT
- A predicate, with the exception of the first operand of the LIKE predicate.
- As anything but a host variable on the right side of the SET clause of an UPDATE statement.

The SUBSTR function can be used to convert portions of long strings to short strings. A further restriction on long strings is that, although the argument of the SUBSTR function may be a long string, the result cannot be a long string.

Note also that blocking is turned off for any cursor operation involving a long string.

Character String Host Variables

Fixed-length string variables can be used in all host languages except REXX. Varying-length string variables can be used in all host languages except Fortran. Varying-length string variables in Assembler, C, and COBOL are simulated. In C, varying-length string variables can also be represented by NUL-terminated strings. String variables with values longer than 254 bytes are subject to the same restrictions as long string columns.

Character Subtypes

Each character string is further defined as having one of the following subtypes:

bit data	Data that is not associated with a coded character set and is therefore never converted. The CCSID for bit data is 65535 (X'FFFF').
SBCS data	Data in which every character is represented by a single byte. Each

SBCS string has an associated CCSID. If necessary, an SBCS string is converted before it is used in an operation with a character string that has a different CCSID.

mixed data Data that may contain a mixture of characters from a single-byte character set (SBCS), a double-byte character set (DBCS) and a multi-byte character set (MBCS). Each mixed string has an associated CCSID. If necessary, a mixed string is converted before an operation with a character string that has a different CCSID. If a mixed data string contains a DBCS character, it cannot be converted to SBCS data.

Each database has a default character subtype, either SBCS or mixed. Host variables assume the default character subtype value. This default subtype can be overridden on a per package basis. Different rules for truncation, padding, and concatenation apply to character subtypes.

If mixed data values are used then the following rules apply to ensure proper formation.

1. Two single-byte EBCDIC codes are given special meanings:
   - X'0E', the shift-out character, used to mark the beginning of a sequence of double-byte codes.
   - X'0F', the shift-in character, used to mark the end of a sequence of double-byte codes.
2. Shift-out and shift-in characters must be paired. The following examples are incorrect:
   ```
   'xy< AABB '      ' AABB <xy'
   ```
3. A trailing shift-out character is an error. The following examples are incorrect:
   ```
   'xy<'      '<'
   ```
4. Neither a shift-out nor shift-in character can be nested. It follows that either a shift-in encountered while processing SBCS or a shift-out encountered while processing DBCS is incorrect. The following examples are incorrect:
   ```
   'xy< AABB < CC '      ' GG >       abc>de'
   '>< AA '      '>'      '>xyz'
   ```
5. The substring of data between a shift-out and a shift-in character is always an even number of bytes (or zero bytes - see redundant shift character pairs in the next rule). The following example is incorrect:
   ```
   'xy< AABBC >'
   ```
6. Redundant shift-out and shift-in or shift-in and shift-out pairs are allowed in properly formed mixed data. The following examples are valid:
   ```
   '<>< BB >'      '< AA >< BB >'
   'xy<>z'     '<>xyz<><>'
   ```

   Generally, redundant pairs will not affect “character sensitive” facilities. For example, in the case of the LIKE predicate, specifying:
   ```
   WHERE COL1 LIKE '%A<>'
   ```

   is identical to
   ```
   COL1 LIKE '%A'
   ```

   Note: The basic equal predicate is not “character sensitive.” For example, specifying:
   ```
   WHERE COL1 = 'A<>'
   ```

would not find a match on the column value

```
'A'
```

In order for the database manager to recognize double-byte characters in a mixed data string, two conditions must be present:

1. During DB2 Server for VM installation, the DBCS option must be set to YES on both the application requester and application server. During DB2 Server for VSE installation, the DBCS option must be set to YES. See “SYSOPTIONS” on page 393.
2. Within the string, the double-byte characters must be enclosed between paired shift-out and shift-in characters.

   The pairing is detected as the string is read from left to right. The code X'0E' is interpreted as a shift-out character if X'0F' occurs later; otherwise it is incorrect. The first X'0F' following the X'0E' is the paired shift-in character.

   There must be an even number of bytes between the paired characters, and each pair of bytes is considered to be a double-byte character. There can be more than one set of paired shift-out and shift-in characters in the string.

The length of a mixed data string is its total number of bytes, counting two bytes for each double-byte character and one byte for each shift-out or shift-in character.

Defining Mixed Data for a Distributed Relational Database: The method of representing DBCS characters within a mixed data string differs between ASCII and EBCDIC.

- ASCII reserves a set of code points for SBCS characters and another set as the first half of DBCS characters. Upon encountering the first half of a DBCS character, the system knows that it is to read the next byte in order to obtain the complete character.
- EBCDIC makes use of two special code points, X'0E' and X'0F', to introduce and end a string of DBCS characters respectively.

When defining mixed data, the integer specified in

```
CHAR(integer)
```

indicates the number of bytes to be used for the column. One effect of this is that more double-byte characters can be stored in a mixed ASCII column than in a mixed EBCDIC column.

Examples:

FF r EE d needs at least CHAR(6) in ASCII
< FF >r< EE >d needs at least CHAR(10) in EBCDIC

Because of these differences, mixed data is not transparently portable between DB2 for OS/2 and DB2 Server for VSE & VM. To minimize the effects of these differences, use varying-length strings in applications that require mixed data and operate on both ASCII and EBCDIC systems. In the previous example, this would mean defining the columns as VARCHAR(10).

Extended UNIX Code (EUC) also allows for a form of ASCII mixed data. It is an encoding scheme supported by UNIX in far eastern countries which allows for MBCS characters. Each EUC codepage is made up of three character sets, or planes, denoted by G0, G1, and G2 or four character sets, denoted by G0, G1, G2 and G3. The group in which the data belongs is determined by the range of its first and second bytes. G0 is comprised of single-byte characters and is the ASCII invariant coded character set. G1 characters are double-byte characters within another range. G2 and G3 characters are triple-byte characters, distinguished by the first byte and the range of the last three bytes.

Graphic Strings

A *graphic string* is any sequence of double-byte characters (and does not include shift-out or shift-in characters). The length of the string is the number of its characters. Like character strings, graphic strings can be empty. There is no subtype associated with graphic strings.

Every graphic string has a CCSID that identifies a double-byte coded character set. If necessary, a graphic string is converted before it is used in an operation with a graphic string that has a different CCSID.

Fixed-Length Graphic Strings

All values of a fixed-length graphic column have the same length, given by the length attribute of the column. The length attribute cannot be greater than 127. Therefore, every fixed-length graphic string column is a short string column.

Varying-Length Graphic Strings

The values of a varying-length graphic string column can have different lengths. The maximum length is determined by the length attribute of the column. The length attribute must be between 1 and 16383.

A varying-length graphic string column with a length attribute greater than 127 is a *long string column*; otherwise it is a *short string column*. A derived graphic string with a maximum length greater than 127 is a *long string*; otherwise it is a *short string*. Long graphic strings are subject to the same limitations that apply to long character strings.

Graphic String Host Variables

Graphic variables can be used in COBOL, PL/I, and REXX.

Numbers

All numbers have a sign and a precision. The *precision* of binary integers and decimal numbers is the total number of binary or decimal digits excluding the sign. The *precision* of floating-point numbers is either single or double, referring to the number of hexadecimal digits in the fraction. If a column value is zero, the sign is positive.

Small Integer

A *small integer* is a System/390* binary integer with a precision of 15 bits. The range of small integers is -32768 to 32767.

Large Integer

A *large integer* is a System/390 binary integer with a precision of 31 bits. The range of large integers is -2,147,483,648 to +2,147,483,647.

Single Precision Floating-Point

A *single precision floating-point* number is a System/390 short (32 bits) floating-point number. The range of magnitude is approximately -7.2E75 to -5.4E-79, 0, +5.4E-79 to 7.2E+75.

Double Precision Floating-Point

A *double precision floating-point* number is a System/390 long (64 bits) floating-point number. The range of magnitude is approximately -7.2E75 to -5.4E-79, 0, +5.4E-79 to 7.2E+75.

Decimal

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, can be neither negative nor greater than the precision. The maximum precision is 31 digits.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is *-n* to *+n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale. The maximum range is from 1.0E-31 up to but not including 1.0E+32.

Note: The precision always remains equal to the attribute that defined the precision. For example, a decimal data type defined with a 6,2 attribute cannot have a 7,2 value stored in it.

Numeric Host Variables

Binary integer variables can be used in all host languages. Floating-point variables can be used in all host languages. Decimal variables can be used in all host languages except C and Fortran.

Datetime Values

Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers. However, strings can represent datetime values; see “String Representations of Datetime Values” on page 49.

Date

A *date* is a three-part value (year, month, and day) designating a point in time under the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D. 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* is 28, 29, 30, or 31, depending on the month.

Note: Historical dates do not always follow the Gregorian calendar. For example, dates between 1582-10-04 and 1582-10-15 are accepted as valid dates although they never existed in the Gregorian calendar.

The internal representation of a date is a string of 4 bytes in packed decimal notation. Each byte consists of two 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 catalog is four bytes, representing the internal length. The length of a DATE column as described in the SQLDA is ten bytes, unless your site specified a date installation exit when the database manager was installed. In the latter case, the string format of a date may be up to 254 bytes in length.

Time

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

The internal representation of a time is a string of 3 bytes. Each byte consists of two digits in packed decimal notation. 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 catalog is 3 bytes, representing the internal length. The length of a TIME column as described in the SQLDA is 8 bytes, unless your site specified a time installation exit when the database manager was installed. In the latter case, the string format may be up to 254 bytes in length.

Timestamp

A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined previously, except that the time includes a specification of microseconds. The range of the microsecond part is 000000 to 999999. If the hour is 24, the microsecond value is 000000.

The internal representation of a timestamp is a string of 10 bytes in packed decimal notation. Each byte consists of two 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 catalog is 10 bytes, representing the internal length. The length of a TIMESTAMP column as described in the SQLDA is 26 bytes.

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 of SQL. Dates, times, and timestamps, however, can also be represented by character strings. These representations directly concern the user of SQL because there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must be assigned to a character string variable. The format of the resulting string will depend on how you defined datetime formats using the DATE and TIME preprocessor options, or how your site chose to represent datetime values at the time the database manager was installed.

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp before the operation is performed. If the CCSID of the string is not the same as the default CCSID, the string is first converted to the coded character set identified by the default CCSID before the string is converted to the internal form of the datetime value.

The following sections define the valid string representations of datetime values.

Date Strings: A string representation of a date is a character string that starts with a digit and has a length of at least 8 characters. An input string representation of a date or time value with LOCAL specified can be any short character string. Trailing blanks can be included. Leading zeros can be omitted from the month and day portions.

Valid string formats for dates are listed in Table 1. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. For a site-defined date string format, the format and length must have been specified when the database manager was installed.

Table 1. Formats for String Representations of Dates

Format Name	**Abbreviation**	**Date Format**	**Example**
International Organization for Standardization	ISO	yyyy-mm-dd	1987-10-12
IBM USA standard	USA	mm/dd/yyyy	10/12/1987
IBM European standard	EUR	dd.mm.yyyy	12.10.1987
Japanese industrial standard Christian era	JIS	yyyy-mm-dd	1987-10-12
Site-defined (see "Defining Your Own Datetime Format" in the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual)	LOCAL	Any site-defined form	—

Time Strings: A string representation of a time is a character string that starts with a digit and has a length of at least 4 characters. An input string representation of a date or time value with LOCAL specified can be any short character string. Trailing blanks can be included; a leading zero can be omitted from the hour part of the time and seconds can be omitted entirely. If you choose to omit seconds, an implicit specification of 0 seconds is assumed. Thus 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 2. Each format is identified by name and includes an associated abbreviation (for use by the CHAR function) and an example of its use. In the case of a site-defined time string format, the format and length must have been specified when the database manager was installed.

Table 2. Formats for String Representations of Times

Format Name	**Abbreviation**	**Time Format**	**Example**
International Organization for Standardization	ISO	hh.mm.ss	13.30.05
IBM USA standard	USA	hh.mm AM or PM [1]	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

Table 2. Formats for String Representations of Times (continued)

Format Name	Abbreviation	Time Format	Example
Site-defined (see "Defining Your Own Datetime Format" in the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual)	LOCAL	Any site-defined form	—
Notes:			
[1] A single space must separate the time and the AM or PM, as shown in the example.			

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. Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:

Table 3. USA Format

USA Format	24 Hour Clock
12:01 AM through 12:59 AM	00.01.00 through 00.59.00
01:00 AM through 11:59 AM	01.00.00 through 11.59.00
12:00 PM (noon) through 11:59 PM	12.00.00 through 23.59.00
12:00 AM (midnight)	24.00.00
00:00 AM (midnight)	00.00.00

Timestamp Strings: A string representation of a timestamp is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-xx-dd-hh.mm.ss.zzzzzz*. Trailing blanks can be included. Leading zeros can be omitted from the month, day, and hour part of the timestamp, and trailing zeros can be truncated or omitted entirely from microseconds. If you choose to omit any digit of the microseconds portion, an implicit specification of 0 is assumed. Thus, *1990-3-2-8.30.00.10* is equivalent to *1990-03-02-08.30.00.100000*.

Null Values

Null is a special value used to represent "not applicable" or "undefined". For example:

In the PROJECT sample table, the value for MAJPROJ in the PROJNO 'AD3100' row is null. In this case a value is not applicable because AD3100 is itself a major project.

In the DEPARTMENT sample table, the value for MGRNO in the DEPTNO 'D01' row is null. In this case the value is undefined.

Null is not the same as blank, an empty string, or zero. Columns can be defined to allow or disallow null values. In an application, a null value can be represented in a host variable by assigning a negative value to that host variable's associated indicator variable.

Assigning Null Values Within the Database:

- Either a host variable with a negative indicator variable or the NULL keyword can be used in an INSERT or UPDATE statement to enter a null value into the database.
- Nullable columns omitted from an INSERT statement are also set to NULL.
- In referential constraints with a DELETE rule of SET NULL, when a row is deleted from a parent table, the nullable foreign key columns of the corresponding rows in the dependent table will be set to null values.

Returning Null Values to the Application from the Database:

- Null values returned to the application are the result of: null values in the database, host variables with negative indicator variables or arithmetic errors.

Null Values within Expressions and Predicates

- If there is a null value as an operand of any arithmetic expression or string expression, then the value of that expression is the null value.
- If the first argument of any scalar function except VALUE has a null value, then the value of that function is null.
- The COUNT column function includes rows with a column having the null value but the other column functions ignore NULLs.
- The value of a column function is null if all of the column values are null.
- Search conditions containing null values are evaluated by applying three-valued logic (see Table 5 on page 89) to the truth values of the component predicates. If the truth-value of a search condition is unknown for a row, the row does not satisfy the search condition.
- If a basic predicate contains an expression which has a null value then the truth-value of the predicate is unknown.
- The effect of null values on quantified predicates, BETWEEN predicates, and IN predicates can be determined by applying the rules for: the definition of the predicates in terms of the basic predicates, null values in basic predicates, and three-valued logic.
- The truth-value of the LIKE predicate is unknown if the value of the column, the pattern or the escape character is null.
- The truth-value of the EXISTS predicate is true even if only null values are returned by the subselect.
- The NULL predicate is true if the column value is null.

Equality and Ordering of Null Values:

- Two null values are not considered to be equal when compared. For instance, the truth-value of an '=' predicate is unknown if both operands have a null value.
- In contrast to the previous point, ORDER BY, GROUP BY and DISTINCT treat all null values as if they were equal. Furthermore, the ORDER BY clause handles a null value as if it was larger than any other value.

Checking for a Null Value:

- The correct way to check for null values is with the NULL predicate. For example:

```
WHERE MGRNO IS NOT NULL
```

On the other hand, the predicate:

```
WHERE NOT MGRNO = :HV:IND
```

will never return any rows if IND has a negative value. In order to search for a null value among other values, the null value must be specified in a separate predicate. For example:

```
WHERE NOT MGRNO < 1000 OR MGRNO IS NULL
```

Rows for which MGRNO is null will not satisfy the predicate:

```
WHERE NOT MGRNO < 1000
```

Assignments and Comparisons

Assignment operations are performed during the execution of FETCH, INSERT, PUT, SELECT INTO, and UPDATE statements. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

The basic rule for both operations is that the data type of the operands involved must be compatible. The compatibility rule also applies to UNION, concatenation, and the VALUE scalar function. The compatibility matrix is as follows:

Operand	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp
Binary Integer	Yes	Yes	Yes	No	No	No	No	No
Decimal Number	Yes	Yes	Yes	No	No	No	No	No
Floating Point	Yes	Yes	Yes	No	No	No	No	No
Character String	No	No	No	Yes	No	*	*	*
Graphic String	No	No	No	No	Yes	No	No	No
Date	No	No	No	*	No	Yes	No	No
Time	No	No	No	*	No	No	Yes	No
Time-stamp	No	No	No	*	No	No	No	Yes

Note: * The compatibility of datetime values and character strings is limited to assignment and comparison:

- Datetime values can be assigned to character string columns and to character string variables as explained in “Datetime Assignments” on page 57.
- 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.

A basic rule for assignment operations is that a null value cannot be assigned to a column that cannot contain null values, nor to a host variable that does not have an associated indicator variable. (See “References to Host Variables” on page 68 for a discussion of indicator variables.)

Numeric Assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number cannot be truncated. If necessary, the fractional part of a decimal number is 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.

Because of the added length of double precision floating-point numbers (64 bits rather than the 32 bits of a single precision value), the approximation is more accurate if the receiving column or variable is defined as double precision rather than single precision. Accuracy is lost if the precision of the target is less than that of the assigned value, as would be the case if a number greater than 16,777,216 were assigned to a single precision floating-point column.

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.

Floating-Point to Floating-Point

When a single precision floating-point number is assigned to a double precision floating-point column or variable, the single precision data is padded with eight hex zeros.

When a double precision floating-point number is assigned to a single precision floating-point column or variable, the double precision data is converted and rounded up on the seventh hex digit.

Floating-Point to Decimal

When a single precision floating-point number is converted to decimal, the number is first converted to a temporary decimal number of precision 6 by rounding on the seventh decimal digit. Nine zeros are then appended to the number to bring the precision to 15. Because of the rounding involved, a number less than $0.5*10^{-6}$ is reduced to 0.

When a double precision floating-point number is converted to decimal, the number is first converted to a temporary decimal number of precision 15. Then, if necessary, the number is truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 15 decimal digits. As a result, a number less than $0.5*10^{-15}$ 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.

Example: This example shows the effect of rounding a double precision floating-point number by using a temporary decimal number:

```
The floating-point number       .123456789098765E-05

in decimal notation is:         .00000123456789098765
                                                +5
Rounding adds 5                 ---------------------
in the 16th position            .00000123456789148765

and truncates the result to:    .000001234567891
```

To COBOL Integers

Assignments to COBOL integer variables use the full size of the integer. Thus, the value placed in the COBOL data item may be out of the range of values.

In COBOL, for example, if COL1 contains a value of 12345, the COBOL statements:

```
01  A  PIC  S9999  BINARY.
EXEC SQL SELECT COL1
         INTO :A
         FROM TABLEX
END-EXEC.
```

result in the value 12345 being placed in A, even though A has been defined with only 4 digits.

Notice that the following COBOL statement:

```
MOVE 12345 TO A.
```

results in 2345 being placed in A.

String Assignments

The general rule for string assignments is that the length of a string assigned to a column must not be greater than the length attribute of the column. (Trailing blanks are included in the length of the string.)

Following are exceptions to this rule:

- If the source of the assignment is a column value, the string is truncated, if necessary, to the length attribute of the target column.
- If the source of the assignment is a fixed-length host variable and the target is a short varying-length column, all the trailing blanks of the source string, if any, are always truncated before assignment. Hence, if the host variable's length attribute is greater than the target column's length attribute and all the excess positions in the original source string contained blanks, the assignment is completed without an error being returned.

When a string is assigned to a fixed-length string column or host variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of blanks (SBCS blanks for character strings of all subtypes, DBCS blanks for graphic strings).

For example, the mixed value `'ab<CC>'` padded to a length of 8 becomes `'ab<CC>   '`.

Bit data is padded with blanks, not with X'00''s.

When a string of length *n* is assigned to a varying-length string variable with a maximum length greater than *n*, the characters after the *n*th character of the variable are undefined and might or might not be set to blanks.

When a string is assigned to a variable and the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters. When this occurs, the value 'W' is assigned to the SQLWARN1 field of the SQLCA. Furthermore, if an indicator variable is provided, it is set to the original length of the string.

The truncation rules for character strings are based on the subtype of the target. The rules are as follows:

- If the target is bit or SBCS, blind truncation occurs.
- If the target is mixed, there is compensation for any incomplete DBCS string. For example, with a target

  ```
  CHAR(8) MIXED DATA:
  ```

  the character string

  ```
  'abc < DDEE >fg'
  ```

  becomes

  ```
  'abc < DD > '
  ```

If truncation is to occur on mixed character data but the data does not follow the proper rules regarding mixed data, then the data will be truncated as SBCS and SQLWARN1 will be set to 'Z'.

For a description of the SQLCA, see “SQL Communication Area (SQLCA)” on page 353.

The above rules apply when both the source and the target are strings. When a datetime data type is involved, see “Datetime Assignments” on page 57.

Conversion Rules for String Assignments

A string assigned to a column or host variable is first converted, if necessary, to the coded character set of the target. Character conversion is necessary only if all of the following are true:

- The CCSIDs are different.
- Neither CCSID is 65535 (X'FFFF').
- The string is neither null nor empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

The database manager returns an error for any of the following conditions;

- The CCSID Conversion Selection Table is used but does not contain any information about the pair of CCSIDs.
- A character of the string cannot be converted, and the operation is assignment to a column or assignment to a host variable without an associated indicator variable.

The database manager returns a warning for any of the following conditions:

- A character of the string is converted to the substitution character.
- A character of the string cannot be converted, and the operation is assignment to a host variable with an associated indicator variable. For example, a DBCS

character cannot be converted and placed in a host variable with an SBCS CCSID. In this example, the string is not assigned to the host variable and the associated indicator variable is set to -2.

Datetime Assignments

A value assigned to a DATE column must be a date or a valid string representation of a date. A date can only be assigned to a DATE column, a character string column, or a character string variable. A value assigned to a TIME column must be a time or a valid string representation of a time. A time can only be assigned to a TIME column, a character string column, or a character string variable. A value assigned to a TIMESTAMP column must be a timestamp or a valid string representation of a timestamp. A timestamp can only be assigned to a TIMESTAMP column, a character string column, or a character string variable.

When a datetime value is assigned to a character string variable or column, it is converted to a string representation. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target varies depending on the format of the string representation. If the length of the target is greater than required, 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.

If the target is a column, truncation is not allowed. The length must be at least 10 for a date, 8 for a time, and 26 for a timestamp.

When the target is a host variable, the following rules for DATE, TIME, and TIMESTAMP apply:

DATE
The length of the variable must not be less than 10.

TIME
If the USA format is used, the length of the variable must not be less than 8. This format does not include seconds.
If the ISO, EUR, or JIS format is used, the length of the variable must not be less than 5. If the length is 5, 6, or 7, the seconds part of the time is omitted from the result, and SQLWARN1 is set to 'W'. In this case, the seconds part of the time is assigned to the indicator variable if one is provided, and, if the length is 6 or 7, blank padding occurs so that the value is a valid string representation of a time.

TIMESTAMP
The length of the variable must not be less than 19. If the length is between 19 and 25, the timestamp is truncated like a string, causing the omission of one or more digits of the microsecond part. If the length is 20, the trailing decimal point is replaced by a blank so that the value is a valid string representation of a timestamp.

For further information on string lengths for datetime values, see “Datetime Values” on page 48.

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

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

String Comparisons

Two strings are compared by comparing the corresponding bytes of each string. If the strings do not have the same length, the comparison is made with a temporary copy of the shorter string that has been padded on the right with blanks so that it has the same length as the other string. All character subtypes use an SBCS blank character for padding. However, when a comparison other than simple equals or not equals occurs between two varying-length string values, the pad character is X'00'.

Note: The only place where this will make a difference is when a value being compared contains non-printable characters, that is, characters whose code point is less than X'40'. Examples of places where X'00' is used are: the greater than predicate, sorting in response to an ORDER BY clause, and retrieving ordered data using an index.

Two strings are equal if they are both empty or if all corresponding bytes are equal. An empty string is equal to a blank string. If two strings are not equal, their relationship is determined by the comparison of the first pair of unequal bytes from the left end of the strings. This comparison is made according to the EBCDIC collating sequence of the CCSID under which they are compared. If a field procedure is defined on a column, the comparison will be made according to the EBCDIC collating sequence of the value encoded by the field procedure, if the encoded value is a string.

Conversion Rules for String Comparison

When two strings are compared, one of the strings is first converted, if necessary, to the coded character set of the other string. Character conversion is necessary only if all of the following are true:
- The CCSIDs of the two strings are different.
- Neither CCSID is 65535 (X'FFFF').
- If either string selected for conversion is null or empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

If one string has an SBCS CCSID and the other has a mixed CCSID, the SBCS string is converted. Otherwise, the string selected for conversion depends on the type of each operand. The following table shows which operand is selected for conversion, given the operand types.

Table 4. Selecting the Operand for Character Conversion

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

A host variable containing data in a foreign encoding scheme is always converted to the native form of data before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

An error occurs if a character of the string cannot be converted or the CCSID Conversion Selection Table is used but does not contain any information about the pair of CCSIDs. A warning occurs if a character of the string is converted to the substitution character.

Datetime Comparisons

A DATE, TIME, or TIMESTAMP value can 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 are implied. Therefore the following predicate is true:

```
TIME('03.42.00') = '03.42'
```

Note: `TIME('00.00.00')` does not compare as an equal to `TIME('24.00.00')`.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent. Thus, the following predicate is true:

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

In comparisons of DATE and TIME, two strings which are not identical are considered to be equal if they represent the same date (that is, '1991-1-1' = '1991-01-01').

Constants

A *constant* (also 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. String constants are further classified as character or graphic.

All constants have the attribute NOT NULL. A negative sign in a numeric constant with a value of zero is ignored.

Integer Constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 10 digits that does not include a decimal point. The data type of an integer constant is large integer, and its value must be within the range of a large integer.

Examples: `64     -15     +100     32767     720176`

In syntax diagrams the term *integer* is used for an 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 can include a sign and a decimal point; the second number can include a sign but not a decimal point. 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. Excluding leading zeros, the number of digits in the first number must not exceed 17 and the number of digits in the second must not exceed 2. The data type of a floating-point constant is double precision floating-point.

Examples: `15E1     2.E5     2.2E-1     +5.E+2`

Decimal Constants

A *decimal constant* specifies a decimal number as a signed or unsigned number that includes a decimal point and at most 31 digits. The precision is the total number of digits (including leading and trailing zeros). When precision is greater than 31, and a precision of 31 is possible by eliminating leading zeros, then those zeros are eliminated.

Examples: `25.5     1000.     -15.     +37589.3333333333`

Character String Constants

A *character string constant* specifies a varying-length character string. There are two forms of character string constant:

- A sequence of characters that starts and ends with a string delimiter ('). 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 254. Two consecutive string delimiters represent one string delimiter within the character string. Two consecutive apostrophes not contained within a string represent an empty string.

  Examples: `'Peggy'     '14.12.1985'     '32'     'DON''T CHANGE'     ''`
- An X followed by a sequence of characters that starts and ends with a string delimiter is called a *hexadecimal constant*. Note that hexadecimal constants are just another way of representing character data. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 254. A hexadecimal digit is a digit or any of the letters A through F (uppercase or lowercase). Under the conventions of

hexadecimal notation, each pair of hexadecimal digits represents a character. This form of string constant lets you specify characters that do not have a keyboard representation.

Hexadecimal constants, as character string constants, are converted from the CCSID of the application requester to the CCSID of the application server as part of the CCSID conversion of statements. One example of where this may give unexpected results is the following situation. Consider the case of an application requester which uses CCSID=851 (an ASCII CCSID) and a DB2 Server for VSE & VM application server which uses CCSID=500 (an EBCDIC CCSID). The table T1 has a character column defined as FOR BIT DATA. The statement

```
INSERT INTO T1 VALUES (X'41')
```

will insert the hexadecimal value X'C1' into the column, because CCSID conversion of the statement happens before the statement is interpreted (note that for CCSID=851, 'A' = X'41' and for CCSID=500, 'A' = X'C1'). To have no conversion occur and have the value X'41' inserted into the column, use a host variable instead of the constant.

Examples: `X'FFFF'     X'535164A1'     X'C5C2C3C4C9C3'     X'4153434949'`

The subtype is classified as mixed data if it includes a DBCS substring (that is, it contains properly matched shift control characters) and the application server and application requester both support DBCS characters (that is, the CHARNAME is a mixed CHARNAME). In all other cases, a character string constant is classified as SBCS data. For example, X’0E42C142C2’ would be interpreted as SBCS data because it contains improperly matched shift control characters.

Examples of mixed character string constants:

```
'abc< XXYYZZ >'    '< XXYYZZ >'    'a< XXYYZZ >b'    '< XX >ab< YYZZ >cd'
```

Character constants of the bit subtype cannot be defined.

The CCSID assigned to a constant is the appropriate default CCSID of the application server at bind time. Since a character string constant is always part of a statement, it is converted with the statement from the default CCSID of the application requester to the default CCSID of the application server. For example, an ASCII constant might get converted to an EBCDIC representation. A special case is the above example where a hexadecimal constant which is first converted to its character representation before it is converted from the default CCSID of the application requester to the default CCSID of the application server.

Graphic String Constants

A *graphic string constant* is a short, varying-length string that specifies a graphic string. There are three forms of graphic string constant, two for static SQL statements in PL/I and one for all other contexts. In the following description of these three forms, the string of characters (XXYYZZ) is the actual string, consisting of 0 to 127 double-byte characters. The character G (N may be used as a synonym for G) and the string delimiter (here represented by the apostrophe [']) are required in the positions indicated.

The forms of graphic string constants are:

- In PL/I source programs: < ’XXYYZZ’GG > and '< XXYYZZ >'G.

In the first form, ▌'▌, is the *double-byte* string delimiter X'427D' and, in this case, GG is the double-byte character X'42C7'. To use that character within the double-byte character sequence, it must be doubled.

See "PREPARE" on page 313 for more information on the use of DBCS constants in prepared statements in PL/I Version 2 programs.

- In all other contexts: G'< XXYYZZ >'

  Here, ' is the EBCDIC string delimiter, X'7D'. With this form of graphic string constant, the empty string can be denoted by `G'<>'`. G must be the single-byte character G.

The CCSID assigned to a constant is the appropriate default CCSID of the application server at bind time, assuming that the application server and application requester support DBCS characters (that is, the CHARNAME is a mixed CHARNAME). Since a graphic string constant is always part of a statement, it is converted with the statement from the default CCSID of the application requester to the default CCSID of the application server.

Graphic string constants are only supported in COBOL, PL/I, and REXX.

Special Registers

A *special register* is a storage area that is defined for an application process by the database manager and stores information that can be referenced in SQL statements. A reference to a special register is a reference to a value provided by the application server. If the value is a string, its CCSID is a default CCSID (based on the default subtype value, CHARSUB) of the application server.

USER

The USER special register specifies a CHAR(8) value that identifies the run-time authorization ID. The authorization ID is padded on the right with blanks, if necessary, so that the value of USER is always a fixed-length character string of length 8.

Example

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

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

CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is processed at the application server. The data type is DATE. 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.

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 SERVER

The CURRENT SERVER special register specifies a CHAR(18) value that identifies the current application server. The server-name ID is padded on the right with blanks, if necessary, so that the value of CURRENT SERVER is always a fixed-length character string of length 18.

The CURRENT SERVER can be changed by the CONNECT statement.

Example

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

```
SELECT CURRENT SERVER
  INTO :APPL_SERVE
  FROM ONE_ROW_TABLE
```

CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is processed at the application server. The data type is TIME. 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.

CURRENT TIMESTAMP

The CURRENT TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is processed at the application server. The data type is TIMESTAMP. 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.

CURRENT TIMEZONE

The CURRENT TIMEZONE special register specifies the difference between UTC (Universal Coordinated Time, formerly known as GMT) and local time at the application server. The data type is DECIMAL(6,0). 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.

Example

Using the IN_TRAY table select all the rows from the table and adjust the value from the RECEIVED column to account for timezone.

```
SELECT RECEIVED - CURRENT TIMEZONE, SOURCE,
           SUBJECT, NOTE_TEXT FROM IN_TRAY
```

Column Names

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

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:

– In a *column function*, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. (Groups and intermediate result tables are explained under Chapter 5, "Queries," on page 121.) 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.

Qualified Column Names

A qualifier for a column name can be a table name, a view name, a synonym, or a correlation name.

Whether a column name can be qualified depends on its context:
- In some forms of the COMMENT ON and LABEL ON statements, the column name must be qualified.
- Where the column name specifies values of the column, a column name can 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. See "Column Name Qualifiers to Avoid Ambiguity" on page 66 and "Column Name Qualifiers in Correlated References" on page 67 for details.

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 shown below establishes Z as a correlation name for X.MYTABLE.

```
FROM X.MYTABLE Z
```

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

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

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 correlation name is always an exposed name. All exposed names in the FROM clause must be unique. A table name, view name, or synonym is said to be exposed in that FROM clause if a correlation name is not specified. 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
```

The names are compared after qualifying any unqualified table or view names.

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                                 INCORRECT
   ```

   a reference to either the first or second instance of EMPLOYEE will be incorrect as neither is uniquely identified.
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). This FROM clause is only valid if the authorization ID of the statement is not X.

A correlation name specified in a FROM clause must not be the same as:
- Any other correlation name in that FROM clause

- Any unqualified table name, view name, or synonym exposed in the FROM clause
- The second SQL identifier of any qualified table name, view name, or synonym in the FROM clause.

For example, the following FROM clauses are incorrect:

```
FROM EMPLOYEE E, EMPLOYEE E
FROM EMPLOYEE DEPARTMENT, DEPARTMENT                INCORRECT
FROM X.T1, EMPLOYEE T1
```

The following FROM clause is technically correct, though potentially confusing:

```
FROM EMPLOYEE DEPARTMENT, DEPARTMENT EMPLOYEE
```

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 or view. The tables and views 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.

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, as in this statement:

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

This example illustrates how to establish table designators in the FROM clause:

- A name that follows a table or view name is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ qualifies the first column name in the select list.
- An exposed name is its own table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE qualifies the second column name in the select list.

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.

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.

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 or view name 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 incorrect, 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 *subselect* is a form of a query that can be used as a component of various SQL statements. Refer to Chapter 5, “Queries,” on page 121 for more information on subselects. A subselect used within a search condition of any statement is called a *subquery*.

A subquery can include search conditions of its own, and these search conditions can, in turn, include subqueries. Thus an SQL statement can 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 has a clause that establishes one or more table designators. This is the FROM clause, except in the highest level of an UPDATE or DELETE statement. A search condition of a subquery can reference not only columns of the tables identified by the FROM clause of its own element of the hierarchy, but also columns of tables identified at any level along the path from its own element 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*.

A correlated reference to column C of table T can be of the form C, T.C, or Q.C, if Q is a correlation name defined for T. However, a correlated reference in the form of an unqualified column name is not good practice. The following explanation is based on the assumption that a correlated reference is always in the form of a qualified column name and that the qualifier is a correlation name.

A qualified column name, Q.C, is a correlated reference only if these three conditions are met:
- Q.C is used in a search condition of a subquery.
- Q does not designate a table used in the FROM clause of that subquery.
- Q does designate a table used at some higher level.

Q.C refers to column C of the table or view at the level where Q is used as the table designator of that table or view. Because the same table or view can be identified at many levels, unique correlation names are recommended as table designators. If Q designates a table at more than one level, Q.C refers to the lowest level that contains the subquery that includes Q.C.

In the following statement, Q is used as a correlation name for T1 and T2, but Q.C refers to the correlation name associated with T2, because it is the lowest level that contains the subquery that includes Q.C.

```
SELECT *
  FROM T1 Q
  WHERE A < ALL (SELECT B
                   FROM T2 Q
```

```
WHERE B < ANY (SELECT D
                 FROM T3
                 WHERE D = Q.C))
```

References to Host Variables

A host variable is an Assembler language storage area, C variable, COBOL data item, Fortran variable, PL/I variable, or a REXX variable that is referenced in an SQL statement. Host variables are defined by statements of the host language (that is, they are given a name and a data type), as described in the *DB2 Server for VSE & VM Application Programming* manual. Note that host variables cannot be referenced in dynamic SQL statements; instead, parameter markers must be used (see “Parameter markers” on page 314).

All host variables used in an SQL statement must be declared in an SQL declare section in all host languages other than REXX, where variables do not have to be declared. 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 Metavariable host-variable

The metavariable *host_variable*, as used in the syntax diagrams, is a reference to a host language variable (the *main variable*) and an optional associated host language variable (the *indicator variable*). A *host_variable* in the INTO clause of a FETCH or a SELECT INTO statement is an output variable to which a value is assigned by the database manager. In all other contexts a *host_variable* is an input variable which provides a value to the database manager.

The general form of a *host_variable* reference is:

```
►►──:host_identifier──┬──────────────────────────────┬──────────────►◄
                      │ ┌─INDICATOR─┐                │
                      └─┴───────────┴─:host_identifier┘
```

The first *host_identifier* designates the *main variable*. Depending on the operation, it either furnishes a value to the database manager, or is furnished one. An *input variable* furnishes a value; an *output variable* is furnished one. For example, an input variable can specify a comparand in a WHERE clause or a replacement for a column value in an UPDATE statement. An output variable can receive a column value when a row is fetched from a table. A given *host_variable* can serve as both an input and an output variable in the same program.

The second *host_identifier* designates the associated *indicator variable* and it must be defined as a half-word integer (corresponding to the data type SMALLINT). The indicator variable does one of the following:
- Identifies the null value. When the indicator variable is negative, this signifies that its associated main variable has the null value.
- Records the original length of a truncated string.
- Indicates that a character could not be converted.
- Indicates that there is an error in the arithmetic expression.
- Records the seconds portion of a time if the time is truncated on assignment to its associated main variable.

Indicator variables, including indicator variables in predicates, can be used to identify null values on input to the database manager (UPDATE, INSERT, or PUT statements or predicates of SELECT, DELETE and UPDATE), or on output from the database manager (INTO clause of SELECT and FETCH statements).

If the second *host_identifier* is omitted, the *host_variable* does not have an indicator variable. The value specified by the *host_variable* reference :V1 is always the value of V1, and the null value cannot be assigned to the variable. It is always good practice to include an output indicator variable. Thus, this form should not be used in an INTO clause unless the corresponding result column cannot contain null values. If this form is used and the column contains nulls, the database manager will return 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.

For more information on host variables, see the *DB2 Server for VSE & VM Application Programming* manual.

Example:

Using the PROJECT table:
- Set the host variable PNAME (varchar(26)) to the project name (PROJNAME)
- Set the host variable STAFF (dec(5,2)) to the mean staffing level (PRSTAFF)
- Set 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'
```

Host Structures and Indicator Arrays

Host structures and indicator arrays can be defined in C, COBOL, and PL/I. Host structures are defined by statements of the host language, as explained in the *DB2 Server for VSE & VM Application Programming* manual. As used here, the term "host structure" does not include an SQLCA or SQLDA.

The form of a host structure reference is identical to the form of a *host_variable* reference. The reference `:S1:I1` is a host structure reference if `S1` names a host structure. `I1` must be defined as either an indicator variable or a one-dimensional array of half-word integer variables . `S1` is the *main structure* and `I1` is its indicator variable or *indicator array*.

In the discussion that follows, let `S1` be a structure defined with variables `V1`, `V2`, and `V3`. Let `I1` be a one-dimensional array of three half-word integers. Each `Vn` is a subfield of `S1` as follows:

```
S1
  V1
  V2
  V3
I1(3)
```

A host structure can be used in any context where a *host_variable_list* can be referenced. A host structure reference is equivalent to a reference to each of the subfields contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator array is the indicator variable for the *n*th subfield of the main structure.

Host structures and variable arrays used as indicator arrays must also be declared in an SQL declare section, the same as *host_variables*.

In PL/I, for example, the statement:

```
EXEC SQL FETCH CURSOR1 INTO :S1;
```

is equivalent to:

```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3;
```

If the main structure has *m* more subfields than the indicator array, the last *m* subfields of the main structure do not have associated indicator variables. If the main structure has *m* less subfields than the indicator array, the last *m* variables of the indicator array are ignored. If an indicator variable (rather than an *indicator array*) is specified, only the first subfield of the main structure has an indicator variable. If an indicator array is not specified for a host structure, no subfield of the main structure has an indicator variable.

The following restrictions apply to the use of a host structure in place of a list of *host_variables*. (For language specific information and examples, please refer to the *DB2 Server for VSE & VM Application Programming* manual.)

- A host structure may be any two-level structure or substructure defined within an SQL declare section. For multi-level structures, all of the deepest two-level structures may be used as host structures.
- Elements of indicator arrays cannot be referenced individually in SQL statements as host variables or indicator variables. However, an indicator array may be referenced following a host variable or host structure subfield. In this case, the first element of the indicator array contains the indicator information.

  For example, the statement:

  ```
  EXEC SQL FETCH CURSOR2 INTO :V2:I1;
  ```

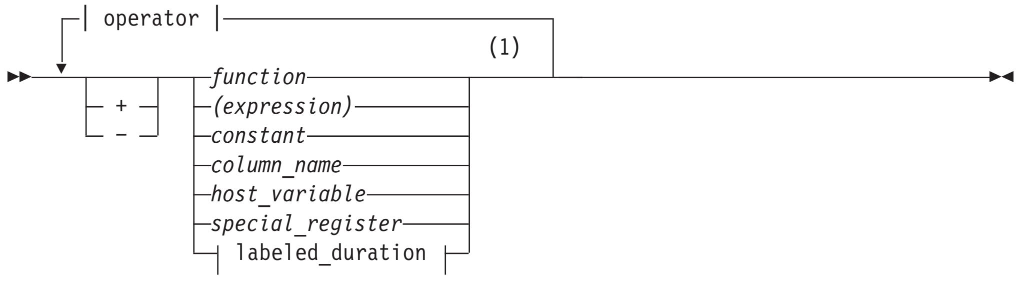
  will result in the first element of I1 receiving the indicator value. The second and third elements of I1 are not affected by this statement.
- If different structures are declared with identical subfield or sub-structure names, any reference to these names must be qualified to the extent needed to ensure that the reference is not ambiguous. In an SQL statement, the format of a qualified name is S1.V1.
- The use of qualified field or subfield names is not supported by the *package_spec* metavariable. Host structure subfields can be used in the *package*us.spec* as host variables, but must be unqualified. (In this case, it is not possible to use a subfield that has been declared with the same name as a subfield in a different host structure.)

A subfield of a structure, including structures which are not valid as host structures, may also be used as a host variable. The only requirement is that the structure must be declared within an SQL declare section.

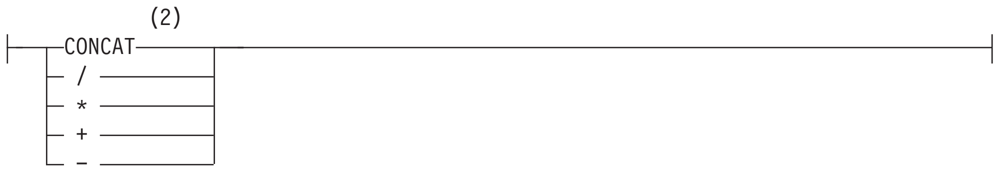
See “Examples” on page 169 for additional coding examples.

Expressions

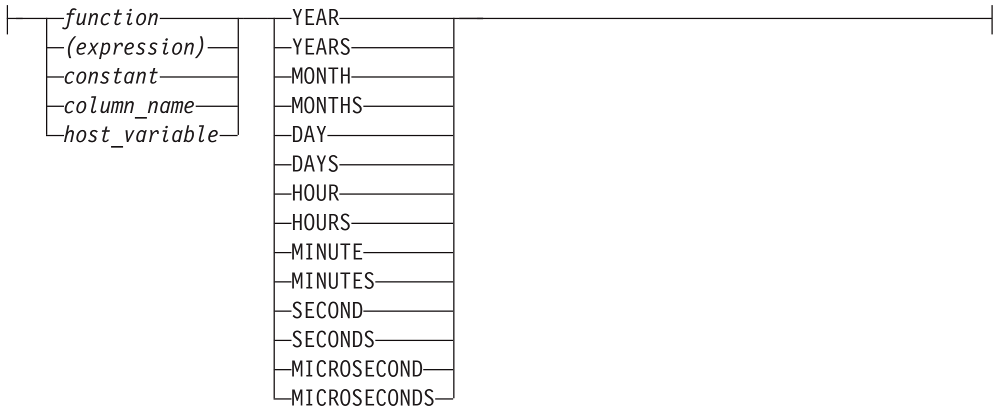
An expression specifies a value.



operator:



labeled_duration:



Notes:

1 Not all combinations of operands and operations are supported.

2 Either || or !! can be used as an alternative to CONCAT in all DB2 Server for VSE & VM-supported code pages. However, !! is not supported in IBM-SQL.

Without Operators

If no operators are used, the result of the expression is the specified value.

Examples:

```
SALARY     :SALARY     'SALARY'    MAX(SALARY)
```

With the Concatenation Operator

The concatenation operator (CONCAT) links two string operands to form a *string expression*.

The operands of concatenation must be compatible strings. Note that datetime data types (including the CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP special registers) can be used as operands in a character string expression because the datetime data types are compatible with the character data type. If both operands are character strings, the sum of their length attributes must not exceed 254; if both are graphic strings, the sum of their length attributes must not exceed 127.

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.

With mixed data this result will not have redundant shift codes “at the seam”. Thus, if the first operand is a string ending with a “shift-in” character, while the second operand is a character string beginning with a “shift-out” character, these two bytes are eliminated from the result. 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, unless redundant shift codes are eliminated, in which case the length is two less than the lengths of the operands.

If both operands are fixed-length character strings (neither of which is mixed data) the result is a fixed-length character string whose length attribute is the sum of the length of the operands. Otherwise, the result is a varying-length character string whose length attribute is the sum of the length attributes of the operands.

If both operands are fixed-length graphic strings, the result is a fixed-length graphic string whose length attribute is the sum of the length of the operands. Otherwise, the result is a varying-length graphic string whose length attribute is the sum of the length attributes of the operands.

The CCSID of the result is determined by the CCSID of the operands as explained under “Conversion Rules for Operations that Combine Strings” on page 130.

If an operand is a string from a column with a field procedure, the operation applies to the decoded form of the value; the result does not inherit the field procedure.

Example 1: `FIRSTNME` **`CONCAT`** `' '` **`CONCAT`** `LASTNAME`

Example 2: Given:

`COLA` defined as VARCHAR(5) with value `'AA'`
`COLB` defined as VARCHAR(5) with value `'BB   '`
`COLC` defined as CHAR(5) with value `'CC   '`
`COLD` defined as CHAR(5) with value `'DDDDD'`

The value of `COLA` **`CONCAT`** `COLB` **`CONCAT`** `COLC` **`CONCAT`** `COLD` is:

```
          'AABB   CC   DDDDD'
```

With Arithmetic Operators

Arithmetic operators (`+, -, *, /`) link two numeric or datetime operands to form a *numeric expression*.

If arithmetic operators are used, the result of the expression is a number derived from the application of the operators to the values of the operands. If any operand can be null, or the expression is used in an outer SELECT list, the result can be null. If any operand has the null value, the result of the expression is the null value. Arithmetic operators must not be applied to character strings. For example, USER+2 is incorrect.

The prefix operator + (*unary plus*) does not change its operand. The prefix operator - (*unary minus*) reverses the sign of a nonzero operand. If the data type of A is *small integer*, the data type of -A is *large integer*. If the data type of A is *small float*, then the data type of -A is *large float*. 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. Either an error or a warning results if the second operand of division has a value of zero.

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. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of large integers.

Integer and Decimal Operands

If one operand is an integer and the other is decimal, the operation is performed in decimal using a temporary copy of the integer that has been converted to a decimal number with zero scale and precision as defined in the following table:

Operand	Precision of Decimal Copy
Column or variable:large integer	11
Column or variable:small integer	5
Constant	Same as the number of digits (including leading zeros) in the constant

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 that has been extended with trailing zeros so that its fractional part has the same number of digits as the other operand.

Unless specified otherwise, all functions and operations that accept decimal numbers allow a precision of up to 31 digits. The result of a decimal operation cannot have a precision greater than 31.

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.

The precision of the result of addition and subtraction is min(31, max(*p*-*s*, *p*'-*s*')+max(*s*, *s*')+1) and the scale is max(*s*, *s*').

The precision of the result of multiplication is min(31, *p*+*p*') and the scale is min(31, *s*+*s*').

The precision of the result of division is 31 and the scale is 31-*p*+*s*-*s*'. If the scale is negative, a negative value is returned in the SQLCODE field of the SQLCA. Precision and scale can be influenced by decimal constants with leading or trailing zeros.

Floating-Point Operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point. If necessary, the operands are first converted to double precision floating-point numbers. 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 that 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 that has been converted to double precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

Datetime Operands

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.

Durations

A *duration* is a number representing an interval of time. There are four types of durations:

Labeled Durations (see diagram on page 71)
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 START_DATE + 2 MONTHS + 14 DAYS is valid, while the expression START_DATE + (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 *yyyyxxdd*, where *yyyy* represents the number of years, *xx* the number of months, and *dd* the number of days. The result of subtracting one DATE value from another, as in the expression END_DATE - START_DATE, 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 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 *yyyyxxddhhmmsszzzzzz,* where *yyyy, xx, dd, hh, mm,* and *ss* represent, respectively, the number of years, months, days, hours, minutes, and seconds, and *zzzzzz* represents the number of 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 215 (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. 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. Here the day portion of the result is set to 28, and the SQLWARN7 condition is set, indicating that an end-of-month adjustment was made to correct an incorrect date.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. (For the purposes of the operation, a month is a calendar page. Adding n months to a date, for example, is like turning n pages of a calendar starting with the page on which the date appears.) The day portion of the date is unchanged unless the result would be incorrect (September 31, for example). Here, the day is set to the last day of the month, and the SQLWARN7 field in SQLCA is set indicating 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 SQLWARN7 is set whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive

DECIMAL(8,0) number, is equivalent to the expression DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years in that order. Thus, DATE1 – X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression DATE1 – DAY(X) DAYS – MONTH(X) MONTHS – YEAR(X) YEARS.

When adding durations to dates, adding one month to a given date gives the same date one month later *unless* that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time Arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting Times: The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0). If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1. If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation RESULT = TIME1 - TIME2.

```
If SECOND(TIME2) <= SECOND(TIME1)
  then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).

If SECOND(TIME2) > SECOND(TIME1)
  then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
  MINUTE(TIME2) is then incremented by 1.

If MINUTE(TIME2) <= MINUTE(TIME1)
  then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).

If MINUTE(TIME2) > MINUTE(TIME1)
  then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
  HOUR(TIME2) is then incremented by 1.

HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
```

For example, the result of TIME('11:02:26') - '00:32:56' is 102930 (a duration of 10 hours, 29 minutes, and 30 seconds).

Incrementing and Decrementing Times: The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order.

For example, `TIME1 + X`, where `X` is a DECIMAL(6,0) number, is equivalent to the expression

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECONDS(X) SECONDS
```

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.
```

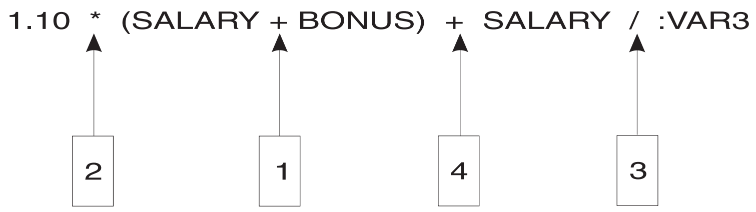
The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

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 are evaluated first. 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.

Example:



Concatenation is performed before datetime arithmetic.

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 the same predicate must be compatible.
- The length attribute of a host variable referenced in a predicate must not be greater than 254 bytes.
- With the exception of the LIKE predicate, a long string column must not be referenced.
- A view column referenced in a predicate must not be derived from a column function.
- If a basic predicate or a LIKE predicate contains an operand with a null value, then it is evaluated as unknown. Other predicates containing negative indicator values are evaluated according to:
  - The rules for defining the predicate in terms of the basic predicates
  - The rules for nulls in basic predicates
  - The truth tables for logic with three values.
- The value of an indicator variable provided at runtime for a parameter marker in a predicate must not be negative.

Basic Predicate

```
>>──expression──┬─ = ────────┬──┬─expression───┬──────────────────────────><
                │      (1)   │  └─(subselect)──┘
                ├─ <> ───────┤
                ├─ < ────────┤
                ├─ > ────────┤
                ├─ <= ───────┤
                └─ >= ───────┘
```

Notes:

1 Either ¬= or ^= may be used as an alternative to the <> operand in all code pages supported by the DB2 Server for VSE & VM database manager. However, these alternatives are not supported in IBM-SQL. Portable applications should use <>.

A *basic predicate* compares two values.

A *subselect* in a basic predicate must specify a single result column and must not return more than one value.

If the value of either operand is null or the subselect returns no value, 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'

  PRTSTAFF <> :VAR1

  SALARY + BONUS + COMM < 20000

SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

Quantified Predicate

```
►►──expression──┬─ = ──────┬──┬─SOME─┬──(subselect)─────────────────────────►◄
                │     (1)  │  ├─ANY──┤
                ├─ <> ─────┤  └─ALL──┘
                ├─ < ──────┤
                ├─ > ──────┤
                ├─ <= ─────┤
                └─ >= ─────┘
```

Notes:

1 Either ¬= or ^= may be used as an alternative to the <> operand in all code pages supported by the DB2 Server for VSE & VM database manager. However, these alternatives are not supported in IBM-SQL. Portable applications should use <>.

A *quantified predicate* compares a value with a set of values.

The *subselect* must specify a single result column and can return any number of values, including null values.

A quantified predicate has the same form as a basic predicate except that the second operand is a *subselect* preceded by SOME, ANY, or ALL.

When ALL is specified, the result of the predicate is:

- **true** if any of the following are true:
  - the subselect returns no values
  - the specified relationship is true for every value returned by the subselect
  - the subselect returns no value and the first operand is null.
- **false** if the specified relationship is false for at least one value returned by the subselect.
- **unknown** if the specified relationship is not false for any of the values returned by the subselect and at least one comparison is unknown because of a null value.

When SOME or ANY is specified, the result of the predicate is:

- **true** if the specified relationship is true for at least one value returned by the subselect.
- **false** if any of the following are true:
  - the subselect returns no values
  - the specified relationship is false for every value returned by the subselect.
- **unknown** if the specified relationship is not true for any of the values returned by the subselect and at least one comparison is unknown because of a null value.

Use the information below when referring to the following examples.

```
TBLA: COLA          TBLB: COLB
      1                   2
      2                   3
      3
      4
```

Example 1

```
SELECT * FROM TBLA WHERE COLA =  ANY(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE
 COLA > ANY(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE  COLA> ALL(SELECT COLB FROM TBLB)
```

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 * FROM TBLA WHERE COLA>  ALL(SELECT COLB FROM TBLB WHERE COLB<0)
```

Results in 1,2,3,4. The subselect returns no values. Thus, the predicate is true for all rows in TBLA.

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 logically equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is logically equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3)
```

that is:

```
value1 < value2 OR value1 > value3
```

If any expression evaluates to a datetime data type, then comparisons will be done with all expressions converted to the appropriate datetime data type.

The values for the expressions in the BETWEEN predicate can have different CCSID values. If a conversion is necessary then it will be based on the above logical equivalence. Conversion is based on the rules for comparisons (see “Conversion Rules for String Comparison” on page 58). If a column’s CCSID is chosen as the final CCSID value then both the other values are converted, if necessary, to that CCSID; this need not be true if the value is not a column’s CCSID.

Example 1

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Example 2

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Example 3

Given the following:

Expression	Type	CCSID
CON_1	constant	00001
HV_2	host variable	00002
HV_3	host variable	00003

When evaluating the predicate:

```
CON_1 BETWEEN :HV_2 AND :HV_3
```

conversion will be based on considering this to be the same as:

```
CON_1 >= :HV_2 AND CON_1 <= :HV_3
```

The values in both HV_2 and HV_3 will be converted to CCSID 00001.

Example 4

Given the following:

Expression	Type	CCSID
CON_1	constant	00001
HV_2	host variable	00002
COL_3	column	00003

When evaluating the predicate:

```
CON_1 BETWEEN :HV_2 AND COL_3
```

conversion will be based on considering this to be the same as:

```
CON_1 >= :HV_2 AND CON_1 <= COL_3
```

Because the CCSID of the column (that is, 00003) is used as the final CCSID value, the values of CON_1 and HV_2 both will be converted to 00003 before any comparisons are done.

Example 5

Given the following:

Expression	Type	CCSID
COL_1	column	00001
HV_2	host variable	00002
CON_3	constant	00003

When evaluating the predicate:

```
COL_1 BETWEEN :HV_2 AND CON_3
```

conversion will be based on considering this to be the same as:

```
COL_1 >= :HV_2 AND COL_1 <= CON_3
```

The values in both HV_2 and CON_3 will be converted to CCSID 00001. (Note the difference in this example's conversion when using a column and example 4.)

Example 6

Given the following:

Expression	Type	Value
COL_1	column CHAR(10)	'01/01/1992'

When evaluating the predicate:

```
COL_1 BETWEEN '07/20/1991' AND '10/22/1992'
```

the comparison will be done as character strings and the predicate will evaluate to false.

When evaluating the predicate:

```
COL_1 BETWEEN DATE('7/20/1991') AND '10/22/1992'
```

the comparison will be done as date types and the predicate will evaluate to true.

EXISTS Predicate

```
►►──┬─────┬──EXISTS──(subselect)─────────────────────────────►◄
    └─NOT─┘
```

The EXISTS predicate tests for the existence of certain rows. The subselect may specify any number of columns, and

- The result is true only if the number of rows specified by the subselect is not zero
- The result is false only if the number of rows specified by the subselect is zero

- The result cannot be unknown.

The values returned by the subselect are ignored.

Example: **EXISTS (SELECT** * **FROM** EMPLOYEE **WHERE** SALARY > 60000**)**

IN Predicate

```
►►──expression──┬─────┬──IN──┬─(subselect)───────────────────────────┬──────────►◄
                └─NOT─┘      │        ┌─,──────────────────────┐     │
                             │        ▼                        │     │
                             └─(──────┬─constant───────────┬───┴─)───┘
                                      ├─host_variable_list─┤
                                      └─special_register───┘
```

The IN predicate compares a value with a set of values.

In the subselect form, the subselect must identify a single result column and may return any number of values, including null values.

An IN predicate of the form:

```
expression IN (subselect)
```

is equivalent to a quantified predicate of the form:

```
expression = ANY (subselect)
```

An IN predicate of the form:

```
expression NOT IN (subselect)
```

is equivalent to a quantified predicate of the form:

```
expression <> ALL (subselect)
```

If any value evaluates to a datetime data type, then comparisons will be done with all values converted to the appropriate datetime data type.

In the non-subselect form of the IN predicate, the second operand is a set of one or more values specified by any combination of constants, host variables, host structures, or special registers. This form of the IN predicate is equivalent to the subselect form except that the second operand consists of the specified values rather than the values returned by a subselect.

The values for the expressions in the IN predicate can have different CCSIDs. Conversion occurs where required based on the assumption that:

```
value1 IN (value2, value3, ...)
```

is logically equivalent to the clause:

```
value1 = value2  OR  value1 = value3  OR ...
```

Conversion is based on the rules for comparisons (see “Conversion Rules for String Comparison” on page 58).

Examples

Example 1

```
DEPTNO IN ('D01', 'B01', 'C01')
```

Example 2

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

Example 3

Given the following:

Expression	Type	CCSID
COL_1	column	00001
HV_2	host variable	00002
HV_3	host variable	00003
CON_4	constant	00004

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

conversion will be based on considering this to be the same as:

```
COL_1 = :HV_2 OR COL_1 = :HV_3 OR COL_1 = CON_4
```

The values in HV_2, HV_3, and CON_4 will be converted to CCSID 00001.

Example 4

Given the following:

Expression	Type	CCSID
HV_1	host variable	00001
CON_2	constant	00002
CON_3	constant	00002
HS_4	host structure	
HV_41	host variable	00003
HV_42	host variable	00004

When evaluating the predicate:

```
:HV_1 IN ( CON_2, CON_3, :HS_4)
```

conversion will be based on considering this to be the same as:

```
:HV_1 = CON_2 OR :HV_1 = CON_3 OR :HV_1 = :HV_41 OR :HV_1 = :HV_42
```

Thus, the value in HV_1 will be converted to CCSID 00002 before it is compared to CON_2 and CON_3, and the values in HV_41 and HV_42 will be converted to CCSID 00001 before they are compared to HV_1.

LIKE Predicate

```
►►──column_name──┬─────┬──LIKE──┬─USER─────────────┬──────────────────────────►
                 └─NOT─┘        ├─host_variable────┤
                                └─string_constant──┘

►──┬─────────────────────────────────────┬─────────────────────────────────────►◄
   └─ESCAPE──┬─host_variable────┬────────┘
             └─string_constant──┘
```

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign have special meanings.

The *column_name* must identify a string column. If a character string column is identified, the other operands must be character strings. If a graphic string column is identified, the other operands must be graphic strings. With character strings, the terms *character, percent sign*, and *underscore* in the following description refer to single-byte characters. With graphic strings, the terms refer to double-byte characters.

Note that trailing blanks in a pattern are usually part of the pattern. The exception to this is that trailing blanks in a pattern that is specified within a fixed-length host variable are ignored when that pattern is compared against a varying-length column.

Simple Description

For character columns, a simple description of the LIKE pattern is as follows:

- The underscore sign (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

Rigorous Description

Let *x* denote a value of a column and *y* denote the string specified by the second operand.

The string *y* is interpreted as a sequence of the minimum number of substring specifiers so each character of *y* 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 *x* or *y* is the null value. Otherwise, the result is either true or false. The result is true if *x* and *y* are both empty strings or if there exists a partitioning of *x* into substrings such that:

- A substring of *x* is a sequence of zero or more contiguous characters and each character of *x* is part of exactly one substring.
- If the *n*th substring specifier is an underscore, the *n*th substring of *x* is any single character.
- If the *n*th substring specifier is a percent sign, the *n*th substring of *x* 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 *x* is equal to that substring specifier and has the same length as that substring specifier.

- The number of substrings of *x* is the same as the number of substring specifiers.

It follows that if `y` is an empty string and `x` is not an empty string, the result is false.

The predicate `x NOT LIKE y` is equivalent to the search condition `NOT(x LIKE y).`

If the CCSID of either the pattern value or the escape value is different than the CCSID of the column, that value is converted to adhere to the CCSID of the column before the predicate is applied.

With Mixed Data

If the column has a mixed subtype, the pattern can include both SBCS and DBCS characters. The special characters in the pattern are interpreted as follows:

- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one DBCS character.
- A percent (either SBCS or DBCS) refers to any number of characters of any type, either SBCS or DBCS.
- Any redundant shifts in either column values or the pattern value are ignored.

With a Field Procedure

If the column has a field procedure, the procedure is invoked to decode the values of the column, and the comparisons are made with the decoded values.

The ESCAPE Clause

This clause allows the definition of patterns intended to match values that contain the actual percent and underscore characters. The following rules govern the use of the ESCAPE clause:

- If a character string column is identified, the escape character must be a character string constant or variable of length 1.
- If a graphic string column is identified, the escape character must be a graphic string constant or variable of length 1.
- If the ESCAPE *host_variable* has a negative indicator variable, the result of the predicate is unknown.
- The *host_variable* or *string_constant* forming the pattern must not contain the escape character except when followed by the escape character, '%' or '_'.

  For example, if '+' is the escape character, any occurrences of '+' other than '++', '+_', or '+%' in the pattern is an error.
- An escape clause cannot be used with a pattern having a mixed subtype.

If both the pattern and the escape character are constants, the entire pattern will always be checked for incorrect occurrences of the escape character.

If either the pattern or the escape character is a *host_variable*, occurrences of the escape character in the pattern will not be validated unless the portion of the pattern proceeding the escape character matches at least one row.

USER as a Pattern

The rules for the LIKE predicate are unchanged with the special register USER. This means that the value of the special register USER is treated as a pattern. USER evaluates to a CHAR(8) string whose value is the user ID of the currently connected user. If the value of USER contains a '_' it will match any character and the result of:

```
WHERE C1 LIKE USER
```

would not be the same as the result of:

```
WHERE C1 = USER
```

It is recommended that the 'equals' predicate be used where user IDs may contain special characters, and the value of USER is not to be treated as a pattern.

Examples:

Example 1: Search for the string ‘SYSTEMS’ appearing anywhere within the PROJNAME column in the PROJECT table.

```
PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

Example 2: Search for a string with a first character of ‘J’ that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

```
EMPLOYEE.FIRSTNME LIKE 'J_'
```

Example 3: In:

```
C1 LIKE 'AAAA+%BBB%' ESCAPE '+'
```

'+' is the escape character and indicates that the search is for a string that starts with 'AAAA%BBB'. The '+%' is interpreted as a single occurrence of '%' in the pattern.

Example 4::

```
both:   WHERE COL1 LIKE 'aaa<AABB%%CC>'
and :   WHERE COL1 LIKE 'aaa<AABB>%<CC>'

would match the value  -->'aaa<AABBDDZZCC>'
as well as the value   -->'aaa<AABB>dzx<CC>'
```

Example 5::

```
WHERE COL1 LIKE 'a%<CC>'

would match the values -->   'a<CC>'    and
                              'ax<CC>'   and
                              'ab<DDEE>fg<CC>'
```

Example 6::

```
WHERE COL1 LIKE 'a_<CC>'

would match the value  -->   'ax<CC>'
but not the value      -->   'a<XXCC>'
```

Example 7::

```
WHERE COL1 LIKE 'a<__CC>'

would match the value  -->   'a<XXCC>'
but not the value      -->   'ax<CC>'
```

Example 8::

```
WHERE COL1 LIKE '<>'
would match the "empty string" value.
```

Example 9::

```
WHERE COL1 LIKE 'ab<CC>_'

would match the values -->   'ab<CC>d and
                             'ab<><CC>d
```

NULL Predicate

```
►►──column_name──IS──┬─────┬──NULL─────────────────────────────────►◄
                     └─NOT─┘
```

The NULL predicate tests for null values.

The result of a NULL predicate cannot be unknown. If the value of the column is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

To search for fields that contain null values, the words IS NULL must be used. 'WHERE PAY IS NULL' is correct, but 'WHERE PAY = NULL' is incorrect.

Examples:

```
EMPLOYEE.PHONE IS NULL

SALARY IS NOT NULL
```

Search Conditions

```
►►──┬─────┬──┬─predicate──────────┬─────────────────────────────────►
    └─NOT─┘  └─(search_condition)─┘
```

```
   ┌───────────────────────────────────────────────┐
   ▼                                               │
►──┴┬───────────────────────────────────────────┬──┴────────────────►◄
    └─┬─AND─┬──┬─────┬──┬─predicate──────────┬─┘
      └─OR──┘  └─NOT─┘  └─(search_condition)─┘
```

A *search condition* specifies a condition that is true, false, or unknown about a given row or group. When the condition is "true," the row or group qualifies for the results. When the condition is "false" or "unknown," the row or group does not qualify.

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 the following table in which P and Q are any predicates:

Table 5. 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

Table 5. Truth Tables for AND and OR (continued)

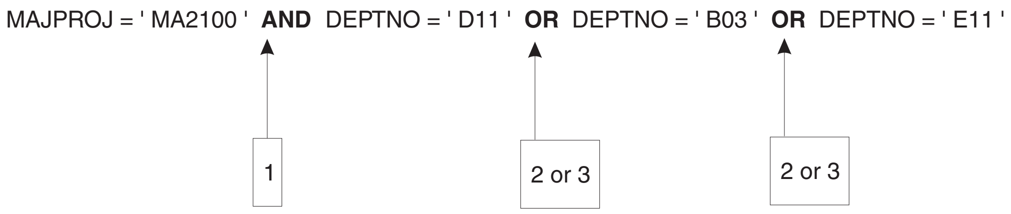
P	Q	P AND Q	P OR Q
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

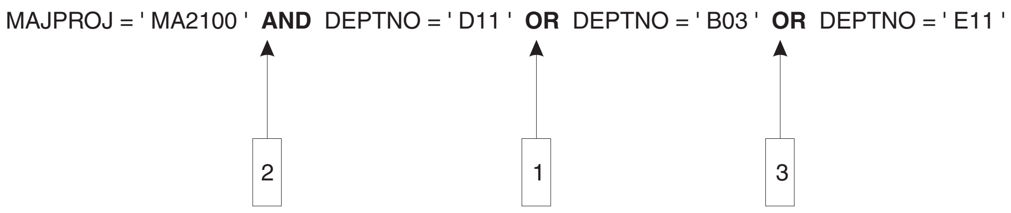
Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

Examples:

Example 1


MAJPROJ = ' MA2100 ' AND DEPTNO = ' D11 ' OR DEPTNO = ' B03 ' OR DEPTNO = ' E11 '
1
2 or 3
2 or 3

Example 2


MAJPROJ = ' MA2100 ' AND DEPTNO = ' D11 ' OR DEPTNO = ' B03 ' OR DEPTNO = ' E11 '
2
1
3

Chapter 4. Functions

A *function* is an operation denoted by a function name followed by one or more operands which are enclosed in parentheses. The operands of functions are called *arguments*. Most functions have a single argument that is specified by an *expression*. The result of a function is a single value derived by applying the function to the result of the expression.

Functions are classified as *column functions* or *scalar functions*. The argument of a column function is a set of values. An argument of a scalar function is a single value. If multiple arguments are allowed, each argument is a single value.

In the syntax of SQL, the term *function* is used only in the definition of an expression. Thus a function can be used only where an expression can be used. Additional restrictions apply to the use of column functions as specified in the following section and in Chapter 5, “Queries,” on page 121.

Column Functions

The following information applies to all column functions, except for the COUNT(*) variation of the COUNT function.

The argument of a column function is a set of values derived from one or more columns. The scope of the set is a group or an intermediate result table as explained in Chapter 5, “Queries,” on page 121. For example, the result of the following SELECT statement is the number of distinct values of JOB for employees in department D01:

```
SELECT COUNT(DISTINCT JOB)
  FROM 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.

The DISTINCT operation can only be applied to values of a column. If DISTINCT is omitted, the values of the arguments are specified by an expression. That expression must not include a column function and must include at least one column-name, a requirement that is not satisfied by a reference to a view column derived from a constant or expression without a column-name. If a *column_name* is a correlated reference (which is allowed in a subquery of a HAVING clause) the expression must not include operators.

AVG

```
           ┌─ALL─┐
►►──AVG──(─┴─────┴──────numeric_expression──┬──)──────────────────────────────►◄
           └─DISTINCT──column_name──────────┘
```

The AVG function returns the average of a set of numbers.

The argument values must be numbers and their sum must be within the range of the data type of the result. The result can be null.

The data type of the result is the same as the data type of the argument values, except that:
- The result is a double-precision floating-point if the argument values are single-precision floating-point.
- The result is a large integer if the argument values are small integers.

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*. Negative scale is not allowed.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are eliminated.

If the function is applied to an empty set, the result is a null value. Otherwise, the result is the average value of the set.

The order in which the summation part of the operation is performed is undefined, but every intermediate result must be within the range of the result data type.

If the type of the result is integer, the fractional part of the average is lost.

Examples

Example 1: Using the PROJECT table, set the host variable AVERAGE (decimal(5,2)) to the average staffing level (PRSTAFF) of projects in department (DEPTNO) ‘D11’.

```
SELECT AVG(PRSTAFF)
  INTO :AVERAGE
  FROM PROJECT
  WHERE DEPTNO = 'D11'
```

Results in AVERAGE being set to 4.25 (that is, 17 / 4) when using the sample table.

Example 2: Using the PROJECT table, set the host variable ANY_CALC 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.

COUNT

```
►►──COUNT──┬─(──DISTINCT──column_name──)─┬──────────────────────────────►◄
           └─(*)─────────────────────────┘
```

The COUNT function returns the number of rows or values in a set of rows or values.

The *column_name* must not identify a long string column. The result of the function is a large integer and must be within the range of large integers. The result cannot be null.

The argument of COUNT(*) is a set of rows. The result is the number of rows in the set.

The argument of COUNT(DISTINCT *column_name*) 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 and duplicate values. The result is the number of values in the set.

Examples

Example 1: Using the EMPLOYEE TABLE, set the host variable FEMALE (int) to the number of rows where the value of the SEX column is ‘F’.

```
SELECT COUNT(*)
  INTO :FEMALE
  FROM EMPLOYEE
  WHERE SEX = 'F'
```

Results in FEMALE being set to 13 when using the sample table.

Example 2: Using the EMPLOYEE table, set the host variable FEMALE_IN_DEPT (int) to the number of departments (WORKDEPT) that have at least one female as a member.

```
SELECT COUNT(DISTINCT WORKDEPT)
  INTO :FEMALE_IN_DEPT
  FROM EMPLOYEE
  WHERE SEX = 'F'
```

Results in FEMALE_IN_DEPT being set to 5 when using the sample table. (There is at least one female in departments A00, C01, D11, D21, and E11.)

MAX

```
               ┌─ALL─┐
►►──MAX──(──┬──┴─────┴──expression──────────┬──)──────────────────────────────►◄
            │              (1)               │
            └─DISTINCT──────────column_name─┘
```

Notes:

1 Although it is allowed, the keyword DISTINCT does not affect the result of the function.

The MAX function returns the maximum value in a set of values.

The argument values can be any values other than long strings.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument. 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 the function is applied to an empty set, the result is a null value. Otherwise, the result is the maximum value in the set.

Examples

Example 1: Using the EMPLOYEE table, set the host variable MAX_SALARY (decimal(7,2)) to the maximum monthly salary (SALARY / 12) value.

```
SELECT MAX(SALARY) /12
INTO :MAX_SALARY
FROM EMPLOYEE
```

Results in MAX_SALARY being set to 4395.83 when using the sample table.

Example 2: Using the PROJECT table, set the host variable LAST_PROJ (char(24)) to the project name (PROJNAME) that comes last in the collating sequence.

```
SELECT MAX(PROJNAME)
  INTO :LAST_PROJ
  FROM PROJECT
```

Results in LAST_PROJ being set to 'WELD LINE PLANNING' when using the sample table.

MIN

```
              ┌─ALL─┐
►►──MIN──(────┴─────┴──expression──────────┬──)──────────────────────────────────►◄
              │        (1)                 │
              └─DISTINCT──────column_name──┘
```

Notes:

1 Although it is allowed, the keyword DISTINCT does not affect the result of the function.

The MIN function returns the minimum value in a set of values.

The argument values can be any values other than long strings.

The data type and length attribute of the result are the same as the data type and length attribute of the argument values. When the argument is a string, the result has the same CCSID as the argument. 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 the function is applied to an empty set, the result is a null value. Otherwise, the result is the minimum value in the set.

Examples

Example 1: Using the EMPLOYEE table, set the host variable COMM_SPREAD (decimal(7,2)) to the difference between the maximum and minimum commission (COMM) for the members of department (WORKDEPT) 'D11'.

```
SELECT MAX(COMM) - MIN(COMM)
  INTO :COMM_SPREAD
  FROM EMPLOYEE
  WHERE WORKDEPT  = 'D11'
```

Results in COMM_SPREAD being set to 1118 (that is, 2580 - 1462) when using the sample table.

Example 2: Using the PROJECT table, set the host variable FIRST_FINISHED (char(10)) to the estimated ending date (PRENDATE) of the first project scheduled to be completed.

```
SELECT MIN(PRENDATE)
  INTO :FIRST_FINISHED
  FROM PROJECT
```

Results in FIRST_FINISHED being set to '1982-09-15' when using the sample table.

SUM

```
          ┌─ALL─┐
►►──SUM──(─┴─────┴──────numeric_expression───┬──)──────────────────────────────►◄
          └─DISTINCT──column_name────────────┘
```

The SUM function returns the sum of a set of numbers.

The argument values must be numbers and their sum must be within the range of the data type of the result.

The 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 and double precision floating-point if the argument values are single precision floating-point. If the data type of the argument values is decimal, the precision of the result is 31 and the scale is the same as the scale of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values by the elimination of null values. If DISTINCT is specified, duplicate values are 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.

Examples

Example 1: 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.

Example 2: Assume that a table SALES has the following columns and values:

Name:	**CUSTOMER**	**SALES**	**MONTHS**
Type:	smallint	int	int
Desc:	Customer number	Total value of purchases by this customer	Number of months that customer has bought something

Name:	CUSTOMER	SALES	MONTHS
Values:	101	1000	5
	102	500	1
	103	300	3

Set the host variable TOT_AVG_SALE (integer) to the sum of the average monthly sales per customer.

```
SELECT SUM(SALES / MONTHS)
  INTO :TOT_AVG_SALE
  FROM SALES
```

Results in TOT_AVG_SALE being set to 800. Note that the expression for each row is calculated before it is added to the sum.

Scalar Functions

A scalar function can be used wherever an expression can be used. The restrictions on the use of column functions do not apply to scalar functions. For example, the argument of a scalar function can be a function; that is, scalar functions can be nested. However, the restrictions that apply to the use of expressions and column functions also apply when an expression or column function is used within a scalar function. For example, the argument of a scalar function can be a column function only if a column function is allowed in the context in which the scalar function is used.

The restrictions on the use of column functions do not apply to scalar functions because a scalar function is applied to a single value rather than a set of values. For example, the result of the following SELECT statement has as many rows as there are employees in department D11:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BIRTHDATE)
  FROM EMPLOYEE
  WHERE WORKDEPT = 'D11'
```

CHAR

```
►►──CHAR──(──┬─┬─date_expression─┬──┬─────────┬─┬──)──────────────────────►◄
             │ └─time_expression─┘  ├─,ISO────┤ │
             │                      ├─,USA────┤ │
             │                      ├─,EUR────┤ │
             │                      ├─,JIS────┤ │
             │                      └─,LOCAL──┘ │
             ├─timestamp_expression─────────────┤
             └─decimal_expression───────────────┘
```

The CHAR function returns a string representation of a datetime value or decimal value.

The first argument must be a decimal number, timestamp, date, or time. The second argument, if applicable, is the name of a string format.

The result of the function is a fixed-length character string. The CCSID of the string is the default CCSID (based on the default subtype value, CHARSUB) of the application server. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The other rules depend on the data type of the first argument:

- If the first argument is a decimal number:

  The second argument must not be specified. The result is the fixed length character string representation of the argument. The first character of the result is a minus sign if the argument is negative; otherwise, the first character is blank.

  The result includes a decimal point, sign, and *p* digits, where *p* is the precision of the argument. The length of the result is 2+*p*.
- If the first argument is a timestamp:

  The second argument is not applicable and must not be specified.

  The result is the character string representation of the timestamp. The length of the result is 26.
- If the first argument is a date:

  Omission of the second argument is an implicit specification of the string format specified at installation time. The installation default for the string format can be overridden by preprocessor options. If LOCAL is implicitly or explicitly specified, a date installation exit must be installed.

  The result is the character string representation of the date in the format specified by the second argument. If LOCAL is specified, the length of the result is the length specified in the SYSOPTIONS catalog during start-up. Otherwise, the length of the result is 10.
- If the first argument is a time:

  Omission of the second argument is an implicit specification of the string format specified at installation time. The installation default can be overridden by preprocessor options. If LOCAL is implicitly or explicitly specified, a time exit must be installed.

  The result is the character string representation of the time in the format specified by the second argument. If LOCAL is specified, the length of the result is the length specified in the SYSOPTIONS catalog during start-up. Otherwise, the length of the result is 8.

Examples

Example 1: Assume the column PRSTDATE has an internal value equivalent to 1988-12-25.

```
CHAR(PRSTDATE, USA)
```

Results in the value '12/25/1988'.

Example 2: Assume the column STARTING has an internal value equivalent to 17.12.30, and the host variable HOUR_DUR (decimal(6,0)) is a time duration with a value of 050000 (that is, 5 hours).

```
CHAR(STARTING, USA)
```

Results in the value '5:12 PM'.

```
CHAR(STARTING + :HOUR_DUR, USA)
```

Results in the value '10:12 PM'.

Example 3: Assume the column RECEIVED (timestamp) has an internal value equivalent to the combination of the PRSTDATE and STARTING columns.

```
CHAR(RECEIVED)
```

Results in the value '1988-12-25-17.12.30.000000'.

DATE

```
►►──DATE──(──expression──)──────────────────────────────────────────►◄
```

The DATE function returns a date from a value.

The argument must be a timestamp, a date, a positive number less than or equal to 3652059, a valid string representation of a date, or a character string of length 7.

If the argument is a character string of length 7, it must represent a valid date in the form *yyyynnn*, where *yyyy* are digits denoting a year, and *nnn* are digits between 001 and 366 denoting a day of that year.

The result of the function is a date. The data type is 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 timestamp:

  The result is the date part of the timestamp.
- If the argument is a date:

  The result is that date.
- If the argument is a number:

  The result is the date that is *n*-1 days after January 1, 0001, where *n* is the number that would occur if the INTEGER function were applied to the argument.
- If the argument is a character string:

  The result is the date represented by the character string.

  Notes:

  1. When a string representation of a date is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a date value.
  2. When a string representation of a date is mixed with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a date value.

Examples

Example 1: Assume that the column RECEIVED (timestamp) has an internal value equivalent to '1988-12-25-17.12.30.000000'.

```
DATE(RECEIVED)
```

Results in an internal representation of '1988-12-25'.

Example 2:

```
DATE('1988-12-25')
```

Results in an internal representation of '1988-12-25'.

```
DATE('25.12.1988')
```

Results in an internal representation of '1988-12-25'.

```
DATE(35)
```

Results in an internal representation of '0001-02-04'.

DAY

```
►►──DAY──(──┬─date_expression───────────────┬──)──────────────────────────────►◄
            ├─timestamp_expression──────────┤
            ├─date_duration_expression──────┤
            └─timestamp_duration_expression─┘
```

The DAY function returns the day part of a value.

The argument must be a date, timestamp, date duration, or timestamp duration. If a decimal number, the argument must be:
- DECIMAL(8,0) for date duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 or a timestamp:

  The result is the day part of the value, which is an integer between 1 and 31.
- If the argument is a date duration or timestamp duration:

  The result is the day part of the value, which is an integer between -99 and 99. A nonzero result has the same sign as the argument.

Examples

Example 1: Using the PROJECT table, set the host variable END_DAY (smallint) to the day that the WELD LINE PLANNING project (PROJNAME) is scheduled to stop (PRENDATE).

```
SELECT DAY(PRENDATE)
  INTO :END_DAY
FROM PROJECT
WHERE (PROJNAME) = 'WELD LINE PLANNING'
```

Results in END_DAY being set to 15 when using the sample table.

Example 2: Assume that the column DATE1 (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.

DAYS

```
►►──DAYS──(──┬─date_expression────────┬──)─────────────────────────────────►◄
             ├─timestamp_expression───┤
             └─date_string_expression─┘
```

The DAYS function returns an integer representation of a date.

The argument must be a date, a timestamp, or a valid string representation of a date.

Notes:

1. When a string representation of a date is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a date value.
2. When a string representation of a date is mixed with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a date value.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is 1 more than the number of days from January 1, 0001 to *D*, where *D* is the date that would occur if the DATE function were applied to the argument.

Examples

Example 1: Using the PROJECT table, set the host variable EDUCATION_DAYS (int) to the number of elapsed days (PRENDATE - PRSTDATE) estimated for the project (PROJNO) 'IF2000'.

```
SELECT DAYS(PRENDATE) - DAYS(PRSTDATE)
  INTO :EDUCATION_DAYS
FROM PROJECT
WHERE (PROJNO) = 'IF2000'
```

Results in EDUCATION_DAYS being set to 396 when using the sample table.

Example 2: Using the PROJECT table, set the host variable TOTAL_DAYS (int) to the sum of elapsed days (PRENDATE - PRSTDATE) estimated for all projects in department (DEPTNO) 'E21'.

```
SELECT SUM(DAYS(PRENDATE) - DAYS(PRSTDATE))
  INTO :TOTAL_DAYS
FROM PROJECT
WHERE (DEPTNO) = 'E21'
```

Results in TOTAL_DAYS being set to 1584 when using the sample table.

DECIMAL

```
►►──DECIMAL──(──numeric_expression──┬──────────────────────────────────────────┬──────►
                                    │                      ┌─,0──────────────┐ │
                                    └─,precision_integer───┴─,scale_integer──┴─┘

►──)─────────────────────────────────────────────────────────────────────────────────►◄
```

The DECIMAL function returns a decimal representation of a number.

numeric_expression
  An expression that returns a value of any numeric data type.

precision_integer
  An integer constant with a value in the range of 1 to 31.

  The default for the *precision_integer* depends on the data type of the *numeric_expression*:
  - 15 for floating-point and decimal
  - 11 for large integer
  - 5 for small integer.

scale_integer
  An integer constant in the range of 0 to the *precision_integer* value.

The result of the function is a decimal number with precision of *p* and scale of *s*, where *p* and *s* are the second and third arguments. If the first argument can be null, the result can be null; if the first argument is null, the result is the null value.

The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *p* and a scale of *s*. An error occurs if the number of significant decimal digits required to represent the whole part of the number is greater than *p-s*.

Examples

Example 1: Use the DECIMAL function in order to force a DECIMAL data type (with a precision of 5 and a scale of 2) to be returned in a select-list for the EDLEVEL column (data type = SMALLINT) in the EMPLOYEE table. The EMPNO column should also appear in the select list.

```
SELECT EMPNO, DECIMAL(EDLEVEL,5,2)
  FROM EMPLOYEE
```

Example 2: Assume the host variable PERIOD is of type INTEGER. Then, in order to use its value as a date duration it must be "cast" as decimal(8,0).

```
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM PROJECT
```

DIGITS

```
►►──DIGITS──(──┬─integer_expression─┬──)──────────────────────────────────────►◄
               └─decimal_expression─┘
```

The DIGITS function returns a character string representation of a number.

The argument is an expression that returns a value of an integer, small integer, or decimal data type.

The result of the function is a fixed-length character string. The CCSID of the string is the default CCSID (based on the default subtype value, CHARSUB) of the application server. 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 a string of digits that represents the absolute value of the argument without regard to its scale. Thus, the result does not include a sign or a decimal point. The result includes any necessary leading zeros so that the length of the string is:
- 5 if the argument is a small integer
- 10 if the argument is a large integer
- *p* if the argument is a decimal number with a precision of *p*.

Examples

Example 1: Using the EMP_ACT table, set the host variable TIME_DISPLAY (char(5)) to the time (EMPTIME) that employee number (EMPNO) '000130' is to spend on an activity (ACTNO) 90.

```
SELECT DIGITS(EMPTIME)
  INTO :TIME_DISPLAY
FROM EMP_ACT
WHERE EMPNO = '000130' AND ACTNO = 90
```

TIME_DISPLAY will be set to '00100' when using the sample table.

Example 2: Return activity number (ACTNO) from the EMP_ACT table as a character string in a select list. The EMPNO and PROJNO columns should also appear in the select list.

```
SELECT EMPNO, PROJNO, DIGITS(ACTNO)
  FROM EMP_ACT
```

FLOAT

```
►►──FLOAT──(──numeric_expression──)──────────────────────────────────►◄
```

The FLOAT function returns a floating-point representation of a number.

The argument is an expression that returns a value of any 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.

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, FLOAT is applied to SALARY so that the division is carried out in floating point:

```
SELECT EMPNO, FLOAT(SALARY)/COMM
  FROM EMPLOYEE
  WHERE COMM > 0
```

HEX

```
►►──HEX──(expression)──────────────────────────────────────────────►◄
```

The HEX function returns a hexadecimal representation of a value.

The argument is an expression that returns a value of any data type other than a long string.

The result of the function is a character string. The CCSID of the string is the default CCSID (based on the default subtype value, CHARSUB) of the application server. 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 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, the result is the hexadecimal representation of the internal form of the argument.

If the argument is a single-byte character string (SBCS), the length of the argument must be 127 or less, and the length of the result is twice the defined (maximum) length of the argument. If the argument is a double-byte character string (DBCS), the length of the argument must be 63 or less, and the length of the result is four times the defined (maximum) length of the argument.

The result is fixed-length if the argument is fixed length. If the argument is varying-length, the result is also varying-length.

Examples

Example 1: Using the DEPARTMENT table set the host variable HEX_MGRNO (char(12)) to the hexadecimal representation of the manager number (MGRNO) for the 'PLANNING' department (DEPTNAME).

```
SELECT HEX(MGRNO)
  INTO :HEX_MGRNO
FROM DEPARTMENT
WHERE DEPTNAME = 'PLANNING'
```

HEX_MGRNO will be set to 'F0F0F0F0F2F0' when using the sample table.

Example 2: Suppose COL_1 is a column with a data type of char(1) and a value of 'B'. The hexadecimal representation of the letter 'B' is X'C2'. HEX(COL_1) returns a two-character string 'C2'.

Example 3: Suppose COL_3 is a column with a data type of decimal(6,2) and a value of 40.1. HEX(COL_3) returns the internal representation, an eight-character string '0004010C'.

HOUR

```
►►──HOUR──(──┬─time_expression──────────────┬──)──────────────────────────►◄
             ├─timestamp_expression─────────┤
             ├─time_duration_expression─────┤
             └─timestamp_duration_expression┘
```

The HOUR function returns the hour part of a value.

The argument must be a time, timestamp, time duration, or timestamp duration. If a decimal number, the argument must be:
- DECIMAL(6,0) for time duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 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
```

INTEGER

```
►►──INTEGER──(──numeric_expression──)─────────────────────────────────►◄
```

The INTEGER function returns an integer representation of a number.

The argument is an expression that returns a value of any numeric 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 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.

Example

Using the EMPLOYEE table, select a list containing salary (SALARY) divided by education level (EDLEVEL). Truncate any decimal in the calculation. The list should also contain the values used in the calculation and 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
```

LENGTH

```
►►──LENGTH──(──expression──)──────────────────────────────────────►◄
```

The LENGTH function returns the length of a value.

The argument is an expression that returns a value of any data type. The expression cannot be a long string host variable.

The result of the function is a large integer. If the argument can be null, the result can be null; if the argument is null, the result is the null value.

The result is the length of the argument. The length of strings includes blanks. 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
- The integer part of (*p*/2)+1 for packed decimal numbers with precision *p*
- 4 for single-precision float
- 8 for double-precision float
- The length of the string for character strings
- 4 for date
- 3 for time
- 10 for timestamp

Note that no special consideration is given for mixed character strings. Shift-in, shift-out, and each byte of a DBCS character within a mixed string are all considered to be single bytes.

Examples

Example 1: 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.

Example 2: Assume that START_DATE is a column of type DATE.

```
LENGTH(START_DATE)
```

Returns the value 4.

Example 3: Assume that START_DATE is a column of type DATE.

```
LENGTH(CHAR(START_DATE, EUR))
```

Returns the value 10.

MICROSECOND

```
►►──MICROSECOND──(──┬─timestamp_expression──────────┬──)────────────────────────►◄
                    └─timestamp_duration_expression─┘
```

The MICROSECOND function returns the microsecond part of a value.

The argument must be a timestamp or timestamp duration. If a decimal number, the argument must be DECIMAL(20,6) for timestamp duration to be properly interpreted.

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:

  The result is the microsecond part of the value, which is an integer between 0 and 999 999.
- If the argument is a duration:

  The result is the microsecond part of the value, which is an integer between –999 999 and 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)
```

MINUTE

```
►►──MINUTE──(──┬─time_expression───────────────┬──)─────────────────────────────►◄
               ├─timestamp_expression──────────┤
               ├─time_duration_expression──────┤
               └─timestamp_duration_expression─┘
```

The MINUTE function returns the minute part of a value.

The argument must be a time, timestamp, time duration, or timestamp duration. If a decimal number, the argument must be:

- DECIMAL(6,0) for time duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 or a 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
```

MONTH

```
►►──MONTH──(──┬─date_expression──────────────┬──)──────────────────────────►◄
              ├─timestamp_expression─────────┤
              ├─date_duration_expression─────┤
              └─timestamp_duration_expression┘
```

The MONTH function returns the month part of a value.

The argument must be a date, timestamp, date duration, or timestamp duration. If a decimal number, the argument must be:
- DECIMAL(8,0) for date duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 or a 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
```

SECOND

```
►►──SECOND──(──┬─time_expression──────────────┬──)─────────────────────────►◄
               ├─timestamp_expression─────────┤
               ├─time_duration_expression─────┤
               └─timestamp_duration_expression┘
```

The SECOND function returns the seconds part of a value.

The argument must be a time, timestamp, time duration, or timestamp duration. If a decimal number, the argument must be:

- DECIMAL(6,0) for time duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 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

Example 1: Assume that the host variable TIME_DUR (decimal(6,0)) has the value 153045.

```
SECOND(:TIME_DUR)
```

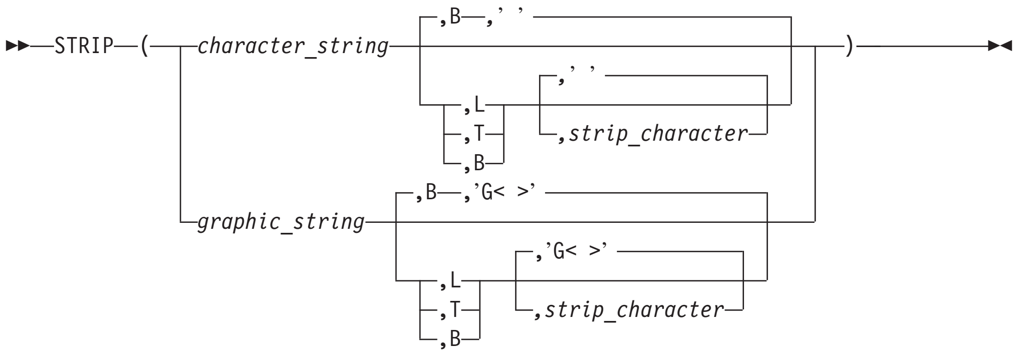
Returns the value 45.

Example 2: 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.

STRIP



The STRIP function returns a value in which blanks, or another specified character, have been removed from the end or the beginning of a string.

character_string or *graphic_string*
: Either a character-compatible expression (CHAR, VARCHAR, TIME, DATE, TIMESTAMP) or a graphic expression (GRAPHIC, VARGRAPHIC). The argument cannot be a long string.

  Note that the argument cannot have a subtype of mixed.

L
T

B One of L, T, or B (not in quotation marks) to remove leading, trailing, or both leading and trailing characters from a string. If a value other than L, T, or B is specified, an error will occur.

The default value is B.

strip_character
A character constant indicating the character to be stripped from *string*.

The default is either a single character space or a graphic double character space (X'4040') depending on the data type of *string*.

The data type and CCSID of the result depends on the data type of the *string* argument. The possible data types and CCSIDs are shown in the following table:

Input Data Type	Output Data Type	Output CCSID
CHAR(n)	VARCHAR(n)	same as that of string
VARCHAR(n)	VARCHAR(n)	same as that of string
GRAPHIC(n)	VARGRAPHIC(n)	same as that of string
VARGRAPHIC(n)	VARGRAPHIC(n)	same as that of string
DATE	VARCHAR(n)[1]	CCSID of the default subtype
TIME	VARCHAR(n)[1]	CCSID of the default subtype
TIMESTAMP	VARCHAR(26)	CCSID of the default subtype

1 For DATE and TIME data types, the value of n is determined by the SYSTEM.SYSOPTIONS values for datetime formats. If a LOCAL format datetime value is used, then n is the LOCAL length specified in SYSTEM.SYSOPTIONS, otherwise 8, 10, and 26 will be used for TIME, DATE, and TIMESTAMP respectively.

The defined length of the result is identical to the defined length of the *string* argument.

If the first argument can be null, then the result can be null. If the first argument is null, the result is the null value.

Examples

Example 1: Assume the host variable HELLO (char(9)) has a value of ' Hello '.

```
STRIP(:HELLO)
```

Returns the value 'Hello'.

```
STRIP(:HELLO, T)
```

Returns the value ' Hello'.

Example 2: Assume the host variable BALANCE (char(9)) has a value of '000345.50'.

```
STRIP(:BALANCE,L,'0', )
```

Returns the value '345.50'.

Example 3: This example shows the treatment of a graphic string.

```
STRIP(G'< XXLLMMNNXXXX >',   T,G'< XX >')
```

Returns the value G'< XXLLMMNN >'

Example 4: This example shows that spaces are treated like any other character. Therefore, if spaces precede the character that is to be stripped from the start of the string, then nothing is stripped.

```
STRIP('  00123.400',B,'0')
```

Returns the value ' 00123.4'.

SUBSTR

```
►►──SUBSTR──(──string_expression,──start_integer_expression──────────────────►

►──┬──────────────────────────┬──)──────────────────────────────────────────►◄
   └─,length_integer_expression─┘
```

The SUBSTR function returns a substring of a string. 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. The CCSID of the result is the same as that of *string*.

string
: Denotes an expression that specifies the string from which the result is derived. *string* must be a character string or a graphic string. If a long *string* is specified, it must be a column, not a host variable, and the resulting string must have a length attribute of not more than 254 bytes (127 characters for graphic data).

  A substring of *string* is zero or more contiguous characters of *string*. If *string* is a graphic string, a character is a DBCS character. If *string* is a character string, a character is a byte.

  The SUBSTR function accepts mixed data strings. However, because SUBSTR operates on a strict byte-count basis, the result will not necessarily be a properly formed mixed data string.

  The SUBSTR function also accepts a datetime argument type for extended flexibility in extracting datetime substring values.

  Note: TIMESTAMP expressions always have an implicit length of 26 and datatype of CHAR. If the statement is a dynamically prepared one, DATE and TIME expressions each have an implicit length of 254 and a data type of VARCHAR. If the statement is not a dynamically prepared one, DATE and TIME expressions each have an implicit data type of CHAR and a length which is determined by the value for datetime formats in the SYSTEM.SYSOPTIONS catalog table. If the datetime format is LOCAL, the length is the LOCAL length in the SYSTEM.SYSOPTIONS catalog table; otherwise, it is 8 for TIME and 10 for DATE.

start
: Denotes an expression that specifies the position of the first character of the result. It must be a positive binary integer that is not greater than the length attribute of *string*. (The length attribute of a varying-length string is its maximum length.)

length
: Denotes an expression that specifies the length of the result. If specified, *length-expression* must evaluate to a binary integer in the range 0 to *n*, where *n*

is the length attribute of *string* - *start* + 1. It must not, however, be the integer constant 0. (SUBSTR(col,1,1-1) is valid; SUBSTR(col,1,0) is not).

If *length* is explicitly specified, *string* is effectively padded on the right with the necessary number of blank characters so that the specified substring of *string* always exists.

The default for *length* is the number of characters from the character specified by the *start* to the last character of *string*. However, if *string* is a varying-length string with an actual length less than *start* (for example, SUBSTR('abcde', 7), the default is zero and the result is the empty string.

If *string* is a character string:

- If *length* is explicitly specified by an integer constant less than or equal to 254, the result is a fixed-length character string with a length attribute of *length*.
- If *length* is not explicitly specified, but *string* is a fixed-length character string and *start* is an integer constant, the result is a fixed-length character string with a length attribute of:

  LENGTH(*string*) - *start* + 1
- If *length* is not explicitly specified, but *string* is a varying-length character string or *start* is not an integer constant, the result is a varying-length character string with a length attribute that is the same as the length attribute of *string*. (Remember, that if the actual length of the string is less than the start position, the actual length of the substring is zero.)
- The maximum length attribute of the result is 254.

If *string* is a graphic string:

- If *length* is explicitly specified by an integer constant less than or equal to 127, the result is a fixed-length graphic string with a length attribute of *length*.
- If *length* is not explicitly specified, but *string* is a fixed-length graphic string and *start* is an integer constant, the result is a fixed-length graphic string with a length attribute of:

  LENGTH(*string*) - *start* + 1
- If *length* is not explicitly specified, but *string* is a varying-length graphic string or *start* is not an integer constant, the result is a varying-length graphic string with a length attribute that is the same as the length attribute of *string*. (Remember, that if the actual length of the string is less than the start position, the actual length of the substring is zero.)
- The maximum length attribute of the result is 127.

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

Example 1: Assume the host variable NAME (varchar(50)) has a value of 'KATIE AUSTIN' and the host variable SURNAME_POS (int) has a value of 7.

```
SUBSTR(:NAME, :SURNAME_POS)
```

Returns the value 'AUSTIN'

```
SUBSTR(:NAME, :SURNAME_POS, 1)
```

Returns the value 'A'.

Example 2: Select all rows from the PROJECT table for which the project name (PROJNAME) starts with the word 'OPERATION '.

```
SELECT * FROM PROJECT
  WHERE SUBSTR(PROJNAME,1,10) = 'OPERATION '
```

The space at the end of the constant is necessary to preclude initial words such as 'OPERATIONS'.

Example 3: Assume there is a host variable VC300 (VARCHAR(300)), the host variable START (int) has a value of 30, and the host variable LNGTH (int) has a value of 250. Obtain a substring of VC300 starting at START with a length of LNGTH.

Attempt 1:

```
SUBSTR(:VC300, :START, :LNGTH)
```

This is not allowed because LNGTH is a host variable and the resulting size is assumed to be that of the source string (300). Thus the size of the host variable exceeds the maximum allowed size of 254.

Attempt 2:

```
SUBSTR(SUBSTR(:VC300, :START, 254), 1, :LNGTH)
```

This attempt is successful.

1. The inner substring, that is:
   ```
   SUBSTR(:VC300, :START, 254)
   ```
   produces a CHAR(254) result whose value is taken from position 30 to position 273 of VC300 (and contains trailing blanks if VC300 is less than 273 bytes long).
2. The outer substring, that is:
   ```
   SUBSTR(inner_result, 1, :LNGTH)
   ```
   produces a VARCHAR(254) result whose value is taken from position one to position 250 of the inner_result. The length of the result is 250.

Attempt 3:

```
SUBSTR(:VC300, 299)
```

This is not allowed because the result is a varying-length string with length attribute 300 which is longer than 254.

TIME

```
►►──TIME──(──┬─time_expression──────────┬──)──────────────────────────────►◄
             ├─timestamp_expression─────┤
             └─time_string_expression───┘
```

The TIME function returns a time from a value.

The argument must be a timestamp, a time, or a valid string representation of a time.

The result of the function is a time. The data type is 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 timestamp:

  The result is the time part of the timestamp.
- If the argument is a time:

  The result is that time.
- If the argument is a character string:

  The result is the time represented by the character string.

  Notes:

  1. When a string representation of a time is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a time value.
  2. When a string representation of a time is mixed with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a time value.

Example

Select all notes from the IN_TRAY sample table that were received at least one hour later in the day (any day) than the current time.

```
SELECT * FROM IN_TRAY
WHERE TIME(RECEIVED) >= CURRENT TIME + 1 HOUR
```

TIMESTAMP

```
►►──TIMESTAMP──(──┬─┬─timestamp_expression─────────────────┬──────────────┬─)──────►◄
                  │ ├─timestamp_str_expression─────────────┤              │
                  │ ├─yyyymmddhhmmss_character_expression──┤              │
                  │ └─390_storeclock_character_expression──┘              │
                  └─┬─date_expression──────┬──┬─,time_expression──────┬───┘
                    └─date_str_expression──┘  └─,time_str_expression──┘
```

The TIMESTAMP function returns a timestamp from a value or a pair of values.

The rules for the arguments depend on whether the second argument is specified.

- If only one argument is specified:

  It must be an expression that returns a timestamp, a valid string representation of a timestamp, a character string of length 8, or a character string of length 14.

  A character string of length 8 is assumed to be a System/390 Store Clock value.

  A character string of length 14 must be a string of digits that represents a valid date and time in the form yyyyxxddhhmmss, where yyyy is the year, xx is the month, dd is the day, hh is the hour, mm is the minute, and ss is the seconds.
- If both arguments are specified:

  The first argument must be an expression that returns either a date or a valid string representation of a date. The second argument must be an expression that returns either a time or a valid string representation of a time.

The result of the function is a timestamp. The data type is TIMESTAMP. If either argument can be null, the result can be null; if either argument is null, the result is the null value.

The other rules depend on whether the second argument is specified:

- If both arguments are specified:

  The result is a timestamp with the date specified by the first argument and the time specified by the second argument. The microsecond part of the timestamp is zero.
- If only one argument is specified and it is a timestamp:

  The result is that timestamp.
- If only one argument is specified and it is a character string:

  The result is the timestamp represented by that character string. The interpretation of a character string as a Store Clock value will yield a timestamp with a year between 1900 to 2042 as described in the *IBM System/370 Principles of Operation* manual.

  Notes:

  1. When a string representation of a timestamp or date and time is SBCS with a CCSID that is not the same as the default CCSID for SBCS data, that value is converted to adhere to the default CCSID for SBCS data before it is interpreted and converted to a timestamp value.
  2. When a string representation of a timestamp or date and time is mixed with a CCSID that is not the same as the default CCSID for mixed data, that value is converted to adhere to the default CCSID for mixed data before it is interpreted and converted to a timestamp value.

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’.

TRANSLATE

```
>>──TRANSLATE──(──┬─char_string_exp──┤ char_string_exp options ├─┬──)──────────><
                  └─┤ graphic_string_exp ├──────────────────────┘
```

char_string_exp options:

```
   ┌─,'ABC...XYZ','abc...xyz'────────────────────────────────┐
├──┼─────────────────────────────────────────────────────────┴──────────────┤
   │                      ┌─,X'000102...FDFEFF'──────────────┐  │
   └─,to_string_exp───────┼──────────────────────────────────┤──┘
                          │                     ┌─,' '─────┐ │
                          └─,from_string_exp────┼──────────┤─┘
                                                └─,pad_char┘
```

graphic_string_exp:

```
                                                               ┌─,G'< >'──┐
├──graphic_string_exp──,to_string_exp──,from_string_exp────────┼──────────┤──────┤
                                                               └─,pad_char┘
```

The TRANSLATE function returns a value in which one or more characters in a string expression may have been translated into other characters.

char_string_exp or *graphic_string_exp*
A short string expression that has either a character data type (CHAR, VARCHAR, DATE, TIME, or TIMESTAMP) or a graphic data type (GRAPHIC or VARGRAPHIC). This argument cannot be a long string.

when the first argument is *char_string_exp:*

to_string_exp
Is a short string expression that has a character data type.

If the length attribute of *to_string_exp* is less than the length attribute of *from_string_exp*, then *to_string-exp* is padded to the longer length using either the *pad_char* or a space.

If the length attribute of *to_string_exp* is more than the length attribute of *from_string_exp*, the extra characters in *to_string_exp* are ignored, without a warning.

from_string_exp
Is a short string expression that has a character data type.

If there are duplicate characters in *from_string_exp*, the first one scanning from the left is used. No warning is issued.

The default value for *from_string_exp* is a string of 256 characters starting with the character X'00' and ending with the character X'FF' (decimal 255).

pad_char
Is a CHAR(1) constant used to pad *to_string_exp* if it is shorter than *from_string_exp*. If the length is not equal to one, an error will occur.

The default *pad_char* is a space.

Note: None of the arguments can have a subtype of mixed.

If *to_string_exp* is not supplied, then *from_string_exp* must not be supplied. In this case, *char_string_exp* is simply translated to upper case. This is done based on the folding rules specified in the SYSCHARSETS catalog table.

when the first argument is *graphic_string_exp***:**

to_string_exp
: Is a short string expression that also returns a graphic data type.

  If the length attribute of *to_string_exp* is less than the length attribute of *from_string_exp*, then *to_string_exp* is padded to the longer length using either the *pad_char* or a graphic space.

  If the length attribute of *to_string_exp* is more than the length attribute of *from_string_exp*, the extra characters in *to_string_exp* are ignored, without a warning.

from_string_exp
: Is a short string expression that has a graphic data type.

  If there are duplicate characters in *from_string_exp*, the first one scanning from the left is used. No warning is issued.

pad_char
: Is a GRAPHIC(1) constant that is used to pad *to_string_exp* if it is shorter than *from_string_exp*. If the length is not equal to 1, an error will occur.

  The default *pad_char* is a graphic space.

Translation Process

The result string is built character by character from *char_string_exp*, translating characters in *from_string_exp* to the corresponding character in *to_string_exp*. For each character in *char_string_exp*, the same character is searched for in *from_string_exp*. If the character is found to be the *n*th character in *from_string_exp*, the resulting string will contain the *n*th character from *to_string_exp*. If *to_string_exp* is less than *n* characters long, the resulting string will contain the pad character. If the character is not found in *from_string_exp*, it is moved to the result string untranslated.

The data type and CCSID of the result depends on the data type of the *string* argument. The possible data types and CCSIDs are shown in the following table:

Input Data Type	**Output Data Type**	**Output CCSID**
CHAR(n)	VARCHAR(n)	same as that of string
VARCHAR(n)	VARCHAR(n)	same as that of string
GRAPHIC(n)	VARGRAPHIC(n)	same as that of string
VARGRAPHIC(n)	VARGRAPHIC(n)	same as that of string
DATE	VARCHAR(n)^1	CCSID default of the subtype
TIME	VARCHAR(n)^1	CCSID default of the subtype
TIMESTAMP	VARCHAR(26)	CCSID default of the subtype

1 For DATE and TIME data types, the value of n is determined by the SYSTEM.SYSOPTIONS values for datetime formats. If a LOCAL format datetime value is used, then n is the LOCAL length specified in SYSTEM.SYSOPTIONS, otherwise 8, 10, and 26 will be used for TIME, DATE, and TIMESTAMP respectively.

The length of the result is identical to the length of the string argument. If any argument can be null, the result can be null; if any argument is null, the result is the null value.

The use of an argument expression (for example, column or host variable) defined as mixed character is not allowed.

Examples

Example 1: Assume the host variable SITE (varchar(30)) has a value of 'Pivabiska Lake Place'.

```
TRANSLATE(:SITE)
```

Returns the value 'PIVABISKA LAKE PLACE'.

```
TRANSLATE(:SITE,'$','L')
```

Returns the value 'Pivabiska $ake Place'.

```
TRANSLATE(:SITE,'$$','Ll')
```

Returns the value 'Pivabiska $ake P$ace'.

```
TRANSLATE(:SITE,'pLA','Place','.')
```

Returns the value 'pivAbiskA LAk. pLA..'.

Example 2: Produce a list that includes the first three columns from all rows in the IN_TRAY sample table and order the list on SUBJECT in a case insensitive manner.

```
SELECT SUBJECT, RECEIVED, SOURCE, TRANSLATE(SUBJECT)
  FROM IN_TRAY
  ORDER BY 4
```

Example 3: This shows the treatment of a graphic string.

```
TRANSLATE(G'  < JJOOHHNN >',G'  < AACCKK >',G'  < OOHHNN >')
```

Returns the value G'< JJAACCKK >'.

VALUE

```
                       ┌──────────────┐
                       ▼              │
►►──VALUE──(──expression────,expression──┴──)──────────────────────────►◄
```

The VALUE function returns the first non-null result in a series of expressions.
- None of the expressions can be long strings.
- The data types of the arguments must be compatible.

If any argument is numeric, all arguments must be numeric (SMALLINT, INTEGER, DECIMAL, and FLOAT). If any argument is a character string, all arguments must be character-compatible strings (CHAR, VARCHAR, DATE, TIME, TIMESTAMP). If any argument is a graphic string, all arguments must be graphic strings (GRAPHIC and VARGRAPHIC).

The arguments are evaluated in the order in which they are specified, and the result value of the function is equal to the first argument that is not NULL. If the arguments can be null, the result can be null; if all the arguments are null, the result is the null value.

- If all arguments are dates, the result data type is a date; if all arguments are times, the result data type is a time; and if all arguments are timestamps, the result data type is a timestamp.
- If the arguments are strings, the CCSID of the result is calculated using the rules given in “Conversion Rules for Operations that Combine Strings” on page 130.
- If all arguments are fixed-length strings, the result is a fixed-length string of length *n*, where *n* is the length of the longest argument.
- If any argument is a varying-length string, the result is a varying-length string with length attribute *n*, where *n* is the length attribute of the argument with the greatest length attribute. The actual length of the result is the actual length of the selected argument.
- If the arguments are numeric, the result data type is the strongest of the argument data types (FLOAT > DECIMAL > INTEGER > SMALLINT).
- If the resultant data type is DECIMAL, precision and scale values are determined as follows: scale is the largest result scale of any data type, and precision is 'MIN(31,SCALE+n)' where 'n' is the largest integral part (precision minus scale) result of any argument. For example, in VALUE(A, B, C), where A is DECIMAL(10, 4), B is DECIMAL(8, 5), C is DECIMAL(5, 2), the resulting scale is MAX(4, 5, 2) = 5 and the resulting precision is

  ```
  MIN(31, 5 + MAX(10-4, 8-5, 5-2)) = 11
  ```

  If scale + 'n' is greater than 31, then there is a potential for a conversion error because there may be a value whose integer part will not fit and an error occurs. For example, in VALUE(X, Y) where X is DECIMAL(27, 2) and Y is DECIMAL(14, 9) the resultant data type is DECIMAL(31,9). If the value of 'X' is 1000000000000000000000000.02 and the value of 'Y' is 30000.000000004, then a decimal overflow occurs because 'X' will not fit in DECIMAL(31,9).

Examples

Example 1: Select all the values from all the rows in the DEPARTMENT table. If the value for department manager (MGRNO) is missing (that is, null) then return a value of ‘ABSENT’.

```
SELECT DEPTNO, DEPTNAME, VALUE(MGRNO, 'ABSENT'), ADMRDEPT
  FROM DEPARTMENT
```

Example 2: Select 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, VALUE(SALARY,0)
  FROM EMPLOYEE
```

VARGRAPHIC

```
►►──VARGRAPHIC──(──expression──)──────────────────────────────────────►◄
```

The VARGRAPHIC function returns a graphic string representation of a character string.

The argument must be a character string (CHAR, VARCHAR) or a character compatible string (DATE, TIME, TIMESTAMP). If varying-length, the maximum length must not be greater than 127. If the argument is a character string constant that is to be interpreted as mixed data, it must contain properly paired shift control characters.

The result of the function is a varying-length graphic string. If the subtype of the argument is SBCS then the CCSID of the result is the system default CCSID for graphic data. If the subtype of the argument is mixed, then the CCSID of the result is the graphic CCSID which makes up the DBCS portion of the mixed argument CCSID. For details, see the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual for the table showing mixed CCSIDs and their corresponding DBCS (and SBCS) component CCSIDs.

If the argument can be null, the result can be null; if the argument is null, the result is the null value. The result includes all DBCS characters of the argument and the DBCS equivalent of all single-byte characters of the argument. The first character of the result is the first logical character of the argument, the second character of the result is the second logical character of the argument, and so on. The result does not include X'0E' or X'0F'.

The DBCS equivalent of X'40' is X'4040'. The DBCS equivalent of every single-byte character (nn) other than X'40' is X'42nn'.

The length of the result depends on the number of logical characters in the argument. If the length or maximum length of the argument is *n* bytes, the maximum length of the result is *n* (DBCS characters).

VARGRAPHIC will convert a mixed data value to GRAPHIC in the following manner:

- DBCS characters from the source will be placed into the target unchanged
- SBCS characters from the source will have their DBCS equivalent placed into the target
- No shift-in or shift-out characters will be transferred to the target.

Example

Using the EMPLOYEE table, set the host variable VAR_DESC (vargraphic(24)) to the VARGRAPHIC equivalent of the first name (FIRSTNME) for employee number (EMPNO) ‘000050’.

```
SELECT VARGRAPHIC(FIRSTNME)
  INTO :VAR_DESC
  FROM EMPLOYEE
  WHERE EMPNO = '000050'
```

VAR_DESC will be set to the GRAPHIC form of 'JOHN' when using the sample table. The hex representation of this is: X'42D142D642C842D5'.

YEAR

```
►►──YEAR──(──┬─date_expression───────────────┬──)──────────────────────────────►◄
             ├─timestamp_expression──────────┤
             ├─date_duration_expression──────┤
             └─timestamp_duration_expression─┘
```

The YEAR function returns the year part of a value.

The argument must be a date, timestamp, date duration, or timestamp duration. If a decimal number, the argument must be:
- DECIMAL(8,0) for date duration
- DECIMAL(20,6) for timestamp duration

to be properly interpreted.

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 or a 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

Example 1: Select all the projects in the PROJECT table that are scheduled to start (PRSTDATE) and end (PRENDATE) in the same calendar year.

```
SELECT * FROM PROJECT
  WHERE YEAR(PRSTDATE) = YEAR(PRENDATE)
```

Example 2: Select all the projects in the PROJECT table that are scheduled to take less than one year to complete.

```
SELECT * FROM PROJECT
  WHERE YEAR(PRENDATE - PRSTDATE) < 1
```

Chapter 5. Queries

A *query* specifies a result table.

In a program, a query is a component of certain SQL statements. There are three forms of a query:
- The *subselect*
- The *fullselect*
- The *select-statement*.

There is another form of select, described under "SELECT INTO" on page 336.

Authorization

For any form of a query, the privileges held by the authorization ID of the statement must include at least one of the following for each of the tables or views identified in the statement:
- Ownership of the table or view
- The SELECT privilege on the table or view
- DBA authority.

subselect

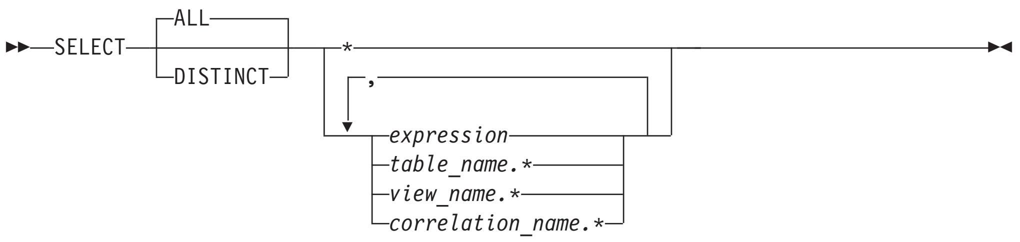
```
►►─select_clause─from_clause──┬──────────────┬──┬─────────────────┬──┬───────────────┬──────────────►◄
                              └─where_clause─┘  └─group_by_clause─┘  └─having_clause─┘
```

The *subselect* is a component of the fullselect, the CREATE VIEW statement, and the INSERT statement. It is also a component of certain predicates which, in turn, are components of a subselect. A subselect that is a component of a predicate is called a *subquery*. If more than one table or view is identified in the FROM clause, the subselect is called a *join*. (See the *DB2 Server for VSE & VM Application Programming* manual for information on joining tables.)

A subselect specifies a result table derived from the tables or views 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 sequence of the (hypothetical) operations is:
1. FROM clause
2. WHERE clause
3. GROUP BY clause
4. HAVING clause
5. SELECT clause.

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. 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. DISTINCT must not be used more than once in a subselect. This restriction includes SELECT DISTINCT and the use of DISTINCT in a column function of the select list or HAVING clause, but does not include subqueries of the subselect.

Two rows are duplicates of one another only if each value in the first row is equal to the corresponding value in the second row. (For determining duplicate rows, 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 statement containing the SELECT clause is prepared. Hence, * does not identify any columns that have been added to a table after the statement has been prepared.

expression
Can be any expression of the type described in Chapter 3. Each column name used in the select list must unambiguously identify a column of R. The operand of an operator must not be a column function that includes the keyword DISTINCT.

*name.**
Represents a list of names that identify the columns of *name*. The *name* can be a table name, view name, or correlation name, and must designate a table or view named in the FROM clause. The first name in the list identifies the first column of the table or view, the second name in the list identifies the second column of the table or view, and so on.

The list of names is established when the statement containing the SELECT clause is prepared. Hence, * does not identify any columns that have been added to a table after the statement has been prepared.

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 during program preparation) and must not exceed 255. The result table of a subquery must be a single column, unless the subquery is used in the EXISTS predicate.

Limitation on Long String Columns

No column in the list may be a long string column if:
- SELECT DISTINCT is used
- The subselect is a subquery
- The subselect is an operand of UNION or UNION ALL.

Note:

The restriction does not apply to a subquery of an EXISTS predicate because an EXISTS subquery does not return values.

Applying the Select List

Some of the results of applying the select list to R depend on whether GROUP BY or HAVING is used. Those results are described separately.

If GROUP BY or HAVING is used::
- Each *column-name* in the select list must either identify a grouping column or be specified within a column function.
- 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:
- The select list must not include any column functions, or it must be entirely a list of column functions.
- If the select list does not include column functions, it 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, 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 attribute of result columns

Result columns allow null values if they are derived from:
- Any column function but COUNT
- A column that allows null values
- An arithmetic expression in an outer select list
- An arithmetic expression that allows nulls
- A scalar function or string expression that allows null values
- A host variable that has an indicator variable
- A result of a UNION if at least one of the corresponding items in the select list is nullable.

Names of result columns

A result column derived from a column name acquires the unqualified name of that column. All other result columns have no names.

Data type 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 or floating-point constant	the same as the data type of the constant, with the same precision and scale for decimal constants. For floating-point constants, the data type is DOUBLE PRECISION.
the name of any numeric host variable	the same as the data type of the variable, with the same precision and scale for decimal variables. If the data type of the variable is not identical to an SQL data type (for example, DISPLAY SIGN LEADING SEPARATE in COBOL), the result column is decimal.
an arithmetic or string expression	the same as the data type of the result. Decimal results have the same precision and scale as described under "Expressions" on page 71.
any function	(see Chapter 4 to determine the data type of the result.)
the name of any string column	the same as the data type of the column, with the same length attribute.
the name of any string host variable	the same as the data type of the variable, with a length attribute equal to the length of the variable. 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.

from-clause

```
           ┌─,──────────────────────────────────────┐
           ▼                                         │
►►──FROM────┬─table_name─┬──┬──────────────────┬─────┴──────────────────►◄
            └─view_name──┘  └─correlation_name─┘
```

The FROM clause specifies an intermediate result table. If a single table or view is identified, the intermediate result table is simply that table or view. If more than one table or view is identified, the intermediate result table consists of all possible combinations of the rows of the identified tables or views. Each row of the result is a row from the first table or view concatenated with a row from the second table or view, 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 named tables or views.

The following rules apply to the names specified in a FROM clause:

- Each table_name or view_name must identify an existing table or view at the application server.
- Each correlation_name is defined as a designator of the table or view identified by the immediately preceding table_name or view_name.
- The exposed names must be unique. An exposed name is a correlation_name, a table_name that is not followed by a correlation_name, or a view_name that is not followed by a correlation_name.

If a correlation_name is specified for a table or view, any qualified reference to a column of that table or view in the subselect must use that correlation_name. For more information about the FROM clause, see "Correlation Names" on page 64.

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 or view identified 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 processed 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 processed for each row of R only if it includes a correlated reference. In fact, a subquery with no correlated references is processed just once, whereas a subquery with a correlated reference may have to be processed once for each row.

group-by-clause

```
                ┌─,───────────┐
                ▼             │
>>──GROUP BY────column_name───┴───────────────────────────────────────────><
```

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.

Each *column_name* must unambiguously identify a column of R other than a long string column. Each identified column is called a *grouping column*.

The result of GROUP BY is a set of groups of rows. The rows within each group are in an arbitrary order. In each group of more than one row, all values of each grouping column are equal; and all rows with the same set of values of the grouping columns are in the same group. For grouping, all null values within a grouping column are considered equal.

Because every row of a group contains the same value of any grouping column, the name of a grouping column can be used in a search condition in a HAVING clause or an expression in a SELECT clause; in each case, the reference specifies only one value for each group.

If the grouping column 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 column 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.

If a field procedure is not involved, the collating sequence depends on the CCSID of the application server. With the DRDA protocol, the application server could be using an ASCII CCSID, producing an unexpected result to an application program assuming an EBCDIC CCSID and code page (see the *IBM SQL Reference* manual for details).

GROUP BY is ignored if used in a subquery of a basic predicate.

having-clause

```
►►──HAVING──search_condition─────────────────────────────────────────►◄
```

The HAVING clause specifies an intermediate result table that consists of those groups of R for which the search_condition is true. R is the result of the previous clause of the subselect. If this clause is not GROUP BY, R is considered a single group with no grouping columns.

Each column name in the search_condition must:

- unambiguously identify a grouping column of R, or
- be specified within a column function, or
- be a correlated reference. A column name is a correlated reference if it identifies a column of a table or view identified 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 processed 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 processed for each group only if it contains a correlated reference. For an illustration of the difference, see examples 6 and 7 under “Examples of a subselect” on page 127.

A correlated reference to a group of R must either identify a grouping column or be contained within a column function.

The HAVING clause must not be used in a subquery of a basic predicate.

Examples of a subselect

Example 1

Select all columns and rows from the EMPLOYEE table.

```
  SELECT * FROM EMPLOYEE
```

Example 2

Join the EMP_ACT and EMPLOYEE tables, select all the columns from the EMP_ACT table and add the employee's surname (LASTNAME) from the EMPLOYEE table to each row of the result.

```
SELECT EMP_ACT.*, LASTNAME
  FROM EMP_ACT, EMPLOYEE
  WHERE EMP_ACT.EMPNO = EMPLOYEE.EMPNO
```

Example 3

Using a join-condition on the EMPLOYEE and DEPARTMENT tables, select the employee number (EMPNO), employee name (FIRSTNME concatenated with MIDINIT concatenated with 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, FIRSTNME CONCAT ' ' CONCAT MIDINIT CONCAT ' ' CONCAT LASTNAME,
    WORKDEPT, DEPTNAME
    FROM EMPLOYEE, DEPARTMENT
    WHERE WORKDEPT = DEPTNO
    AND YEAR(BIRTHDATE) < 1930
```

Example 4

Select the job (JOB) and the minimum and maximum salaries (SALARY) for each group of rows with the same job code in the EMPLOYEE table, but only for groups with more than one row and with a maximum salary greater than or equal to 27 000.

```
  SELECT JOB, MIN(SALARY), MAX(SALARY)
    FROM EMPLOYEE
    GROUP BY JOB
    HAVING COUNT(*) > 1 AND MAX(SALARY) >= 27000
```

Example 5

Select all the rows of EMP_ACT table for employees (EMPNO) in department (WORKDEPT) 'E11'. (Employee department numbers are shown in the EMPLOYEE table.)

```
  SELECT * FROM EMP_ACT
    WHERE EMPNO IN (SELECT EMPNO FROM EMPLOYEE
                         WHERE WORKDEPT = 'E11')
```

Example 6

From the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary for all employees.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                               FROM EMPLOYEE)
```

The subquery in the HAVING clause would only be processed once in this example.

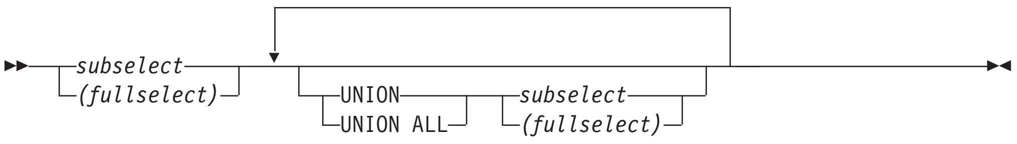
Example 7

Using the EMPLOYEE table, select the department number (WORKDEPT) and maximum departmental salary (SALARY) for all departments whose maximum salary is less than the average salary in all other departments.

```
SELECT WORKDEPT, MAX(SALARY)
  FROM EMPLOYEE EMP_COR
  GROUP BY WORKDEPT
  HAVING MAX(SALARY) < (SELECT AVG(SALARY)
                               FROM EMPLOYEE
                               WHERE NOT WORKDEPT = EMP_COR.WORKDEPT)
```

Note: In contrast to example 6, the subquery in the HAVING clause would need to be executed for each group.

fullselect



A *fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

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 duplicate rows eliminated. In either case, each row of the UNION table is either a row from R1 or a row from R2. The columns of the result are not named in the SQLDA.

Two rows are duplicates if each value in the first is equal to the corresponding value of the second. (For determining duplicates, two null values are considered equal.)

UNION and UNION ALL are associative operations. However, when UNION and UNION ALL are used in the same statement, the result depends on the order in which the operations are performed. Operations within parentheses are performed first. When the order is not specified by parentheses, operations are performed in left-to-right order.

Rules for columns

R1 and R2 must have the same number of columns, and the data type of the nth column of R1 must be compatible with the data type of the nth column of R2.

R1 and R2 must not include long string columns.

The nth column of the result of UNION and UNION ALL is derived from the nth columns of R1 and R2. The following table shows all valid combinations of operand columns and, for each combination, the data type of the result column.

If one operand column is...	And the other operand is...	The data type of the result column is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = max(x,y)
bit data	mixed, SBCS, or bit data	bit data
mixed data	mixed or SBCS data	mixed data
SBCS data	SBCS data	SBCS data
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)
DATE	DATE	DATE
TIME	TIME	TIME
TIMESTAMP	TIMESTAMP	TIMESTAMP
FLOAT (double)	any numeric type	FLOAT (double)
FLOAT (single)	FLOAT (single)	FLOAT (single)
FLOAT (single)	DECIMAL, NUMERIC, INTEGER, or SMALLINT	FLOAT (double)
DECIMAL(w,x)	DECIMAL(y,z) or NUMERIC(y,z,)	DECIMAL(p,s) where p = max(x,z)+max(w-x,y-z) s = max(x,z) (see note below)
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where p = x+max(w-x,11) (see note below)
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = x+max(w-x,5) (see note below)
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
SMALLINT	SMALLINT	SMALLINT
Note: A decimal result column must not have a precision greater than 31.		

If neither operand column allows nulls, the result column does not allow nulls. Otherwise, the result column allows nulls. If the description of any operand column is not the same as the description of the result column, its values are converted to conform to the description of the result column.
The conversion operation is exactly the same as if the values were assigned to the result column. For example, if one operand column is CHAR(10), and the

other operand column is CHAR(5), the result column is CHAR(10), and the values derived from the CHAR(5) column are padded on the right with five blanks.

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) begins with either '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'.

```
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')
```

Conversion Rules for Operations that Combine Strings

The operations that combine strings are concatenation, UNION, and UNION ALL. These rules also apply to the VALUE scalar function. In each case, the CCSID of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the coded character set identified by that CCSID.

The CCSID of the result is determined by the CCSIDs of the operands. The CCSIDs of the first two operands determine an intermediate result CCSID, this CCSID and the CCSID of the next operand determine a new intermediate result CCSID, and so on. The last intermediate result CCSID and the CCSID of the last operand determine the CCSID of the result string or column. For each pair of CCSIDs, the result CCSID is determined by the sequential application of the following rules:

- If the CCSIDs are equal, the result is that CCSID.
- If either CCSID is 65535 (X'FFFF'), the result is 65535.
- If one CCSID denotes SBCS data and the other denotes mixed data, the result is the CCSID for mixed data.

- Otherwise, the result CCSID is determined by the following table:

Table 6. Selecting the CCSID of the Intermediate Result

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

However, a host variable containing data in a foreign encoding scheme is always converted to the native form of data before it is used in any operation. The above rules are based on the assumption that this conversion has already occurred.

Note that an intermediate result is considered to be a derived value operand. For example, assume COLA, COLB, and COLC are columns with CCSIDs 37, 278, and 500, respectively. The result CCSID of COLA CONCAT COLB CONCAT COLC would be determined as follows:

– The result of the CCSID of COLA CONCAT COLB is first determined to be 37, because both operands are columns, so the CCSID of the first operand is chosen.
– The result CCSID of “intermediate result” CONCAT COLC is determined to be 500, because the first operand is a derived value and the second operand is a column, so the CCSID of the second operand is chosen.

An operand of concatenation or the selected argument of the VALUE scalar function is converted, if necessary, to the coded character set of the result string. Each string of an operand of UNION or UNION ALL is converted, if necessary, to the coded character set of the result column. Character conversion is necessary only if all of the following are true:
- The CCSIDs are different.
- Neither CCSID is 65535 (X'FFFF').
- The string is neither null nor empty.
- The CCSID Conversion Selection Table indicates that conversion is necessary.

An error occurs if a character of a string cannot be converted or if the CCSID Conversion Selection Table is used but does not contain any information about the CCSID pair. A warning occurs if a character of a string is converted to the substitution character.

Examples

Example 1

Given the following:

Expression	**Type**	**CCSID**
COL_1	column	00001
HV_2	host variable	00002
COL_3	column	00003

When evaluating the predicate:

```
COL_1 CONCAT :HV_2 CONCAT COL_3
```

The resulting CCSID of the first two operands is 00001. Because the result of the first concatenation is a derived string, the second concatenation will have a result CCSID of 00003 (the column CCSID is chosen over the CCSID of the derived string). The final CCSID is 00003.

Example 2

Using the information from the previous example, when evaluating the predicate:

```
VALUE(COL_1, :HV_2, COL_3)
```

The resulting CCSID of the first two operands is 00001. However, the expression type is column, not derived string, since the two operands are not combined as in the concatenation example. Using the rules for the intermediate CCSID and the third operand's CCSID, 00001 (the intermediate CCSID) is chosen as the final CCSID. This is because the CCSID of the first column is chosen over the CCSID of the second column.

select-statement

```
>>──fullselect──┬───────────────────────┬──┬─────────────┬──────────────────────────><
                ├─order_by_clause───────┤  └─with_clause─┘
                ├─for_fetch_only_clause─┤
                ├─for_read_only_clause──┤
                │                (1)    │
                └─update_clause─────────┘
```

Notes:

1 The update-clause cannot be specified if the fullselect contains an order-by-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 interactively, causing a result table to be displayed at your terminal. In either case, the table specified by a *select-statement* is the result of the fullselect.

order-by-clause

```
                 ┌─,─────────────────────────────┐
                 V                  ┌─ASC──┐     │
>>──ORDER BY───────┬─column_name─┬──┼──────┼─────┴──────────────────────────><
                   └─integer─────┘  └─DESC─┘
```

The ORDER BY clause specifies an ordering of the rows of the result table. If a single column is identified, the rows are ordered by the values of that column. If more than one column is identified, the rows are ordered by the values of the first identified column, then by the values of the second identified column, and so on. A long string column must not be identified.

A named column may be identified by an *integer* or a *column_name*. An unnamed column must be identified by an integer. A column is unnamed if it is derived from a constant, an arithmetic expression, or a function. If the fullselect includes a UNION operator, every column of the result table is unnamed.

column_name
: Must unambiguously identify a column of the result table. Although columns not included in the result table cannot be referenced in the ORDER BY clause, the rules for unambiguous column references are the same as in the other clauses of the fullselect. See “Column Name Qualifiers to Avoid Ambiguity” on page 66 for more information

integer
: Must be greater than 0 and not greater than the number of columns in the result table. The integer *n* identifies the *n*th column of the result table.

ASC
: Uses the values of the column in ascending order. This is the default.

DESC
: Uses the values of the column in descending order.

Ordering is performed in accordance with the comparison rules described in Chapter 3. The null value is higher than all other values. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified column have an arbitrary order. If the ORDER BY clause is not specified, the rows of the result table have an arbitrary order.

If a field procedure is not involved, the collating sequence depends on the CCSID of the application server. With the DRDA protocol, the application server could be using an ASCII CCSID, producing an unexpected result to an application program assuming an EBCDIC CCSID and code page (see the *IBM SQL Reference* for details).

for_fetch_only/for_read_only clause
: The FOR FETCH/READ ONLY clause forces blocking on read only cursors without the use of BLOCK as a preprocessing option.

  If the application has been preprocessed with the NOBLOCK option (no blocking) then the FOR FETCH ONLY will be ignored.

  FOR READ ONLY is a synonym of the FOR FETCH ONLY.

update-clause

```
                 ┌─,───────────┐
                 ▼             │
►►──FOR UPDATE OF───column_name─┴──────────────────────────────►◄
```

The 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. The clause must not be specified if the result table of the fullselect is read-only.

If an UPDATE clause is specified, only the columns identified in that clause can be updated in subsequent Positioned UPDATE statements.

If a dynamically prepared select-statement does not include an UPDATE clause, its associated cursor is not updateable.

The use of this clause for statically prepared select-statements and statically prepared Positioned UPDATE statements depends on whether the NOFOR preprocessor option is in effect. If the UPDATE clause is not specified:

- If NOFOR is in effect, all updateable columns can be updated in subsequent Positioned UPDATE statements.
- If NOFOR is not in effect, the cursor associated with the select-statement is not updateable.

See the *DB2 Server for VSE & VM Application Programming* manual for information on preprocessing and running programs.

If blocking is not in effect, a row may be deleted from a non-read-only table if the FOR UPDATE OF clause is specified.

If blocking is in effect and you intend to perform a positioned delete operation, blocking must be explicitly turned off by specifying the FOR UPDATE OF clause in the DECLARE statement.

Notes:

1. There is no corresponding FOR DELETE OF clause.
2. The order-by clause is not allowed with the update-clause.

with-clause

```
►►──WITH──┬─RR─┬─────────────────────────────────────────────────►◄
          ├─CS─┤
          └─UR─┘
```

The WITH clause specifies the isolation level at which the statement is executed.

RR
Repeatable read

CS
Cursor stability

UR
Uncommitted read

WITH UR can be specified only if the result table is read-only.

The isolation level specified on the SELECT statement will override any other isolation level specification; for example, in ISQL, if SET ISOLATION CS has been specified, and a SELECT statement WITH UR is executed, that statement will use an isolation level of uncommitted read. A SELECT statement without the WITH clause will use an isolation level of CS, as defined by the SET ISOLATION statement. As another example, a statement specifying WITH UR in a package prepped with ISOL(CS) will use an isolation level of uncommitted read.

If a SELECT statement specifying WITH UR is used with a cursor that is not read-only, SQLCODE -173 will be returned indicating that WITH UR cannot be specified on a select statement used in a non-read-only cursor.

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 PL/I 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;
```

Chapter 6. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the following table.

Table 7. SQL Statements

SQL Statement	Function	Refer to Page
ACQUIRE DBSPACE	Obtains and names a dbspace.	"ACQUIRE DBSPACE" on page 144
ALLOCATE CURSOR	Defines a cursor and associates it with a result set locator variable.	"ALLOCATE CURSOR" on page 146
ALTER DBSPACE	Alters the percentage of free space. Also alters the lock size of a PUBLIC dbspace.	"ALTER DBSPACE" on page 148
ALTER PROCEDURE	Alters the definition of an existing stored procedure.	"ALTER PROCEDURE" on page 150
ALTER PSERVER	Alters the definition of an existing stored procedure server.	"ALTER PSERVER" on page 155
ALTER TABLE	Adds a column to a table or manages referential constraints.	"ALTER TABLE" on page 157
ASSOCIATE LOCATORS	Obtains the RESULT SET LOCATOR value for each result set returned by a stored procedure.	"ASSOCIATE LOCATORS" on page 166
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section.	"BEGIN DECLARE SECTION" on page 169
CALL	Invokes a stored procedure.	"CALL" on page 171
CLOSE	Closes a cursor.	"CLOSE" on page 175
Extended CLOSE	Closes a cursor defined by an Extended DECLARE CURSOR statement.	"Extended CLOSE" on page 177
COMMENT ON	Replaces or adds a comment to the description of a table, view, or column.	"COMMENT ON" on page 178
COMMENT ON PROCEDURE	Replaces or adds a comment to the description of a stored procedure identified.	"COMMENT ON PROCEDURE" on page 180
COMMIT	Terminates a logical unit of work and commits the database changes made by that logical unit of work.	"COMMIT" on page 182
CONNECT	Connects to an application server.	"CONNECT (for VM)" on page 185
CREATE INDEX	Defines an index on a table.	"CREATE INDEX" on page 198
CREATE PACKAGE	Creates a package.	"CREATE PACKAGE" on page 201
CREATE PROCEDURE	Defines a stored procedure.	"CREATE PROCEDURE" on page 208
CREATE PSERVER	Defines a stored procedure server.	"CREATE PSERVER" on page 216
CREATE SYNONYM	Defines an alternate name for a table or view.	"CREATE SYNONYM" on page 218
CREATE TABLE	Defines a table.	"CREATE TABLE" on page 219

Table 7. SQL Statements (continued)

SQL Statement	Function	Refer to Page
CREATE VIEW	Defines a view of one or more tables or views.	“CREATE VIEW” on page 231
DECLARE CURSOR	Defines an SQL cursor.	“DECLARE CURSOR” on page 235
Extended DECLARE CURSOR	Defines a cursor that is to be associated with a statement that was prepared using an Extended PREPARE statement.	“Extended DECLARE CURSOR” on page 240
DELETE	Deletes zero or more rows from a table.	“DELETE” on page 242
DESCRIBE	Describes the result columns of a prepared statement.	“DESCRIBE” on page 247
Extended DESCRIBE	Describes the result columns of a SELECT statement that was prepared using an Extended PREPARE statement.	“Extended DESCRIBE” on page 251
DESCRIBE CURSOR	Obtains information about the result set that is associated with the cursor and puts that information into a descriptor.	“DESCRIBE CURSOR” on page 252
DESCRIBE PROCEDURE	Obtains information about the result sets returned by a stored procedure and puts that information into a descriptor.	“DESCRIBE PROCEDURE” on page 254
DROP	Deletes a dbspace, index, package. synonym, table, or view	“DROP” on page 257
DROP PROCEDURE	Deletes the definition of a stored procedure.	“DROP PROCEDURE” on page 260
DROP PSERVER	Deletes the definition of a stored procedure server.	“DROP PSERVER” on page 261
DROP STATEMENT	Deletes a statement from a package created with CREATE PACKAGE.	“DROP STATEMENT” on page 262
END DECLARE SECTION	Marks the end of a host variable declaration section.	“END DECLARE SECTION” on page 263
EXECUTE	Executes a prepared SQL statement.	“EXECUTE” on page 264
Extended EXECUTE	Executes an SQL statement prepared using an Extended PREPARE statement.	“Extended EXECUTE” on page 268
EXECUTE IMMEDIATE	Prepares and executes an SQL statement.	“EXECUTE IMMEDIATE” on page 270
EXPLAIN	Obtains information about the structure and execution performance of a DELETE, INSERT, UPDATE, or SELECT statement.	“EXPLAIN” on page 273
FETCH	Assigns values of a row of a result table to host variables.	“FETCH” on page 283
Extended FETCH	Assigns values of a row in a result table to host variables using a cursor defined by an Extended DECLARE CURSOR statement.	“Extended FETCH” on page 287
GRANT (Package Privileges)	Grants privilege to execute statements in a package	“GRANT (Package Privileges)” on page 288
GRANT (System Authorities)	Grants system authorities.	“GRANT (System Authorities)” on page 290
GRANT (Table Privileges)	Grants privileges on a table or view.	“GRANT (Table Privileges)” on page 293
INCLUDE	Inserts declarations into a source program.	“INCLUDE” on page 296

Table 7. SQL Statements (continued)

SQL Statement	Function	Refer to Page
INSERT	Inserts zero or more rows into a table.	"INSERT" on page 298
LABEL ON	Replaces or adds a label on the description of a table, view, or column.	"LABEL ON" on page 303
LOCK DBSPACE	Either prevents concurrent processes from changing a dbspace or prevents concurrent processes from using a dbspace.	"LOCK DBSPACE" on page 305
LOCK TABLE	Either prevents concurrent processes from changing a table or prevents concurrent processes from using a table.	"LOCK TABLE" on page 306
OPEN	Opens a cursor.	"OPEN" on page 307
Extended OPEN	Opens a cursor defined by an Extended DECLARE CURSOR statement.	"Extended OPEN" on page 312
PREPARE	Prepares an SQL statement (with optional parameters) for execution within the same logical unit of work.	"PREPARE" on page 313
Extended PREPARE	Prepares an SQL statement into a package created with CREATE PACKAGE.	"Extended PREPARE" on page 317
PUT	Inserts (a row of) data into a table.	"PUT" on page 322
Extended PUT	Inserts (a row of) data into a table using a cursor defined by an Extended DECLARE CURSOR statement.	"Extended PUT" on page 325
REVOKE (Package Privileges)	Revokes the privilege to execute statements in a package.	"REVOKE (Package Privileges)" on page 327
REVOKE (System Authorities)	Revokes system authorities.	"REVOKE (System Authorities)" on page 328
REVOKE (Table Privileges)	Revokes privileges on a table or view.	"REVOKE (Table Privileges)" on page 330
ROLLBACK	Terminates a logical unit of work and backs out the database changes made by that unit of work.	"ROLLBACK" on page 334
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables.	"SELECT INTO" on page 336
UPDATE	Updates the values of one or more columns in zero or more rows of a table.	"UPDATE" on page 338
UPDATE STATISTICS	Update statistics on tables and indexes in system catalogs.	"UPDATE STATISTICS" on page 344
WHENEVER	Defines actions to be taken on the basis of SQL return codes.	"WHENEVER" on page 346

How SQL Statements Are Invoked

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The *Invocation* section in the description of each statement indicates whether the statement is executable.

An *executable statement* can be invoked in three ways:
- Embedded in an application program
- Dynamically prepared and processed
- Issued interactively.

Depending on the statement, you can use some or all of these methods. The *Invocation* section in the description of each statement tells you which methods can be used.

A *nonexecutable statement* can only be embedded in an application program.

In addition to the statements described in this chapter, there is one more SQL statement construct: the *select-statement*. (See "select-statement" on page 133.) It is not included in this chapter because it is used differently from other statements.

A *select-statement* can be invoked in three ways:
- Included in DECLARE CURSOR and implicitly processed by OPEN
- Dynamically prepared, referenced in DECLARE CURSOR, and implicitly processed by OPEN
- Entered interactively.

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

The different methods of invoking an SQL statement are discussed below in more detail. For each method, the discussion includes the mechanism of execution, interaction with host variables, and testing if the execution was successful.

Embedding a Statement in an Application Program

You can include SQL statements in a source program that will be submitted to the preprocessor. Such statements are said to be *embedded* in the program. An embedded statement can be placed where a similar host language statement is allowed in the program. You must precede each embedded statement with EXEC SQL.

Executable statements

An executable statement embedded in an application program is run every time a statement of the host language would be processed if specified in the same place. (Thus, for example, a statement within a loop is run every time the loop is processed, and a statement within a conditional construct is run only when the condition is satisfied.)

An embedded statement can contain references to host variables. A host variable referenced in this way can be used in two ways:
- As input (the current value of the host variable is used in the execution of the statement)
- As output (the variable is assigned a new value as a result of executing the statement).

In particular, all references to host variables in expressions and predicates are effectively replaced by current values of the variables, that is, the variables are used as input. The treatment of other references is described individually for each statement.

All executable statements should be followed by a test of an SQL return code (see "SQL Return Codes" on page 142). Alternatively, you can use the WHENEVER statement (which is itself nonexecutable) to change the flow of control immediately after the execution of an embedded statement.

If the program is prepared with the NOEXIST option (see the *DB2 Server for VSE & VM Application Programming* manual), then objects referenced in SQL statements need not exist when the statements are prepared.

Nonexecutable statements

An embedded nonexecutable statement is processed only by the preprocessor. The preprocessor reports any errors encountered in the statement. The statement is *never* processed, and acts as a no-operation if placed among executable statements of the application program. Therefore, you should not follow such statements by a test of an SQL return code.

Dynamic Preparation and Execution

Your application program can dynamically build an SQL statement in the form of a character string placed in a host variable. In general, the statement is built from some data available to the program (for example, input from a terminal). The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE and processed by means of the (embedded) statement EXECUTE. Alternatively, you can use the (embedded) statement EXECUTE IMMEDIATE to prepare and process a statement in one step.

A statement that is going to be dynamically prepared must not contain references to host variables. It can instead contain parameter markers. (See "PREPARE" on page 313 for rules concerning the parameter markers.) When the prepared statement is processed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. (See "EXECUTE" on page 264 for rules concerning this replacement.) After prepared, a statement can be processed several times with different values of host variables. Note that parameter markers are not allowed in EXECUTE IMMEDIATE.

The successful or unsuccessful execution of the statement is indicated by the setting of an SQL return code in the SQLCA after the EXECUTE (or EXECUTE IMMEDIATE) statement. You should check the SQL return code as described above for embedded statements. See "SQL Return Codes" on page 142 for more information.

Static Invocation of a select-statement

You can include a *select-statement* as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is processed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table a row at a time by successive executions of the FETCH statement.

The *select-statement* used in this way may contain references to host variables. These references are effectively replaced by the values that the variables have at the moment of executing OPEN.

Dynamic Invocation of a select-statement

Your application program can dynamically build a *select-statement* in the form of a character string placed in a host variable. In general, the statement is built from some data available to the program (for example, a query obtained from a terminal). The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. The statement is then processed every time you open the cursor by means of the (embedded) statement OPEN. After the cursor is open, you can retrieve the result table one row at a time by successive executions of the FETCH statement.

The *select-statement* used in that way must not contain references to host variables. It can instead contain parameter markers. (See "PREPARE" on page 313 for rules concerning the parameter markers.) The parameter markers are effectively replaced by the values of the host variables specified in the OPEN statement. (See "OPEN" on page 307 for rules concerning this replacement.)

Interactive Invocation

A capability for entering SQL statements from a terminal is part of the architecture of the database manager. This product provides ISQL and the Database Services utility for this facility. An associated product, Query Management Facility (QMF), also provides interactive access to DB2 Server for VSE & VM databases. A statement entered in this way is said to be issued interactively. See the *DB2 Server for VSE & VM Interactive SQL Guide and Reference* manual and the *DB2 Server for VSE & VM Database Services Utility* manual for more information and examples.

A statement issued interactively must be an executable statement that does not contain parameter markers or references to host variables. These make sense only in the context of an application program.

SQL Return Codes

An application program containing executable SQL statements must either provide a structure named SQLCA or a stand-alone integer variable named SQLCODE (SQLCOD in Fortran and RPG). An SQLCA is provided automatically in REXX and RPG. In other languages, an SQLCA can be obtained by using the INCLUDE SQLCA statement. INCLUDE SQLCA must not be used if a stand-alone SQLCODE is provided.

The SQLCA includes an integer variable named SQLCODE (SQLCOD in Fortran and RPG). The option of providing a stand-alone SQLCODE instead of an SQLCA allows for conformance with the ISO/ANSI SQL standard. This option can be requested with either the STDSQL(89) or NOSQLCA preprocessor option as described in the *DB2 Server for VSE & VM Application Programming* manual.

SQLCODE

Regardless of whether the application program provides an SQLCA or a stand-alone variable, SQLCODE is set by the database manager after each SQL statement is processed. All IBM database managers conform to the ISO/ANSI SQL standard, as follows:

- If SQLCODE = 0 and SQLWARN0 is blank, execution was successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data, because the cursor was positioned after the last row of the result table.
- If SQLCODE > 0 and not = 100, execution was successful with a warning.
- If SQLCODE = 0 and SQLWARN0 = 'W', execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

The meaning of SQLCODE values other than 0 and 100 is usually product-specific.

SQLSTATE

SQLSTATE is also set by the database manager after execution of each SQL statement. Thus, application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE (SQLSTT in Fortran and RPG) is a character string variable in the SQLCA.

SQLSTATE provides application programs with common codes for common error conditions. Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors. The coding scheme is the same for all database managers and is based on the proposed ISO/ANSI SQL2 standard. See "SQLSTATEs" in the *DB2 Server for VM Messages and Codes* or the *DB2 Server for VSE Messages and Codes* manual for more information and a complete list of the possible values of SQLSTATE.

SQL Comments

Static SQL statements can include host language or SQL comments. SQL comments are introduced by two hyphens.

These rules apply to the use of SQL comments:

- The two hyphens must be on the same line, not separated by a space.
- Comments can be started wherever a space is valid (except within a delimiter token or before or between 'EXEC' and 'SQL').
- Comments are terminated by the end of line.
- Comments are not allowed within statements that are dynamically prepared (using PREPARE or EXECUTE IMMEDIATE) or prepared using any of the extended dynamic PREPARE statements.
- In COBOL, the hyphens must be preceded by a space.

For host language rules regarding the use of SQL comments, see the *DB2 Server for VSE & VM Application Programming* manual.

Example

This example shows how to include comments in a statement:

```
CREATE VIEW PRJ_MAXPER -- projects with most support personnel
  AS SELECT PROJNO, PROJNAME -- number and name of project
  FROM PROJECT
  WHERE DEPTNO = 'E21' -- systems support dept code
  AND PRSTAFF > 1
```

ACQUIRE DBSPACE

The ACQUIRE DBSPACE statement causes the database manager to find and name an available dbspace.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- DBA authority to acquire either a public dbspace or a dbspace for another user
- RESOURCE authority to acquire a private dbspace.

Syntax

```
►►──ACQUIRE──┬─PUBLIC──┬──DBSPACE NAMED──dbspace_name──────────────────────────►
             └─PRIVATE─┘

►──┬─────────────────────────────────────────────┬─────────────────────────────►◄
   │    ┌─,─────────────────────────────────┐    │
   │    ▼  (1)                              │    │
   └─(────────┬─NHEADER = ──┬─8───────┬─────┬─┴─)─┘
              │             └─integer─┘     │
              │          ┌─128─────┐        │
              ├─PAGES = ─┴─integer─┴────────┤
              │             ┌─33──────┐     │
              ├─PCTINDEX = ─┴─integer─┴─────┤
              │            ┌─15──────┐      │
              ├─PCTFREE = ─┴─integer─┴──────┤
              │         ┌─PAGE────┐         │
              ├─LOCK = ─┼─DBSPACE─┼─────────┤
              │         └─ROW─────┘         │
              └─STORPOOL = ──integer────────┘
```

Notes:

1 If any of these clauses is specified more than once, the value with the first specification is used.

Description

PUBLIC/PRIVATE
Is the type of dbspace requested. If the dbspace is PUBLIC, its owner becomes PUBLIC; if the type is PRIVATE, its owner becomes the authorization ID of the statement.

NAMED *dbspace-name*
Provides a name for the dbspace. The name must be a valid SQL identifier. It must be unique within all the dbspaces owned by the same user, but may duplicate the name of a dbspace owned by another user.

If the dbspace name of a private dbspace is qualified, the qualifier is the owner of the dbspace. Otherwise, the authorization ID of the statement is the owner of the dbspace. The owner has all privileges on the dbspace. The privileges can be granted by the owner and cannot be revoked from the owner.

If the dbspace name of a public dbspace is qualified, the qualifier must be "PUBLIC".

NHEADER
Is the number of 4096-byte logical pages in the dbspace that the database manager reserves for header pages. Header pages record information about the contents of the dbspace. NHEADER cannot be larger than eight pages.

PAGES
Is the minimum number of 4096-byte logical pages required for this dbspace. The database manager determines the page number by rounding the number you specify to the **next higher** multiple of 128.

PCTINDEX
Is the percentage of **all** pages in the dbspace that the database manager is to reserve for the construction of indexes.

PCTFREE
Is the percentage of space on **each** page that the database manager is to keep free when data is inserted into the dbspace.

LOCK
Is the lock size, applicable to public dbspaces only. The lock size determines the extent of locking that the database manager acquires when a user reads or updates data. If ROW is specified, only a row in the table is locked; PAGE or DBSPACE cause the smallest lockable unit to be a page (4096 bytes) or the dbspace, respectively.

STORPOOL
Is the storage pool number. This parameter tells the database manager to acquire the dbspace from a specified storage pool. If a dbspace of the specified type and size is not available in the storage pool, the ACQUIRE DBSPACE is not successful and the database manager returns an error. If STORPOOL is not specified, the database manager acquires a dbspace of the correct size and type from any **recoverable** storage pool. For more information, see the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual.

Examples

Acquire a private dbspace in storage pool number 3 and call it FCPSPACE. Leave 25% of the space free on each page.

```
ACQUIRE PRIVATE DBSPACE NAMED FCPSPACE
  (STORPOOL=3, PCTFREE=25)
```

ALLOCATE CURSOR

The ALLOCATE CURSOR statement defines a cursor and associates it with a result set locator variable.

Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot by issued interactively.

Authorization

None required.

Syntax

```
►►──ALLOCATE──cursor-name──CURSOR FOR RESULT SET──rs-locator-variable──────────►◄
```

Description

cursor-name
Identifies a cursor name, which must be unique within the logical unit of work in which it is used. It is an ordinary identifier.

CURSOR FOR RESULT SET *rs–locator–variable*
Identifies a result set locator variable that has been declared in the application program according to the rules for declaring result set locator variables. The result set locator variable must contain a valid result set locator value, as is returned by the ASSOCIATE LOCATORS or DESCRIBE PROCEDURE SQL statement.

Notes

1. **Dynamically prepared ALLOCATE CURSOR statements:**

   One restriction is that a statement identifier cannot be used for an ALLOCATE CURSOR statement if the same statement identifier has been used for a DECLARE CURSOR statement. For example, the following SQL statements are not valid because the PREPARE statement uses STMT1 as an identifier for the ALLOCATE CURSOR statement when it has already been used for a DECLARE CURSOR statement:

   ```
   DECLARE C1 CURSOR FOR STMT1;

   PREPARE STMT1 FROM
       'ALLOCATE C2 CURSOR FOR RESULT SET ?'; INVALID
   ```

   If an ALLOCATE CURSOR statement is dynamically prepared, the DYNALC prep option must be used for the preprocessor to successfully process any FETCH statements issued against the allocated cursor. If the prep option is not used, the preprocessor returns SQLCODE -504 for these FETCH statements because the cursor was not identified by the prep.

2. **Rules for using an allocated cursor:**

   The following rules apply when you use an allocated cursor:

   - You cannot open an allocated cursor by using the SQL OPEN cursor statement.
   - You can close an allocated cursor by using the SQL CLOSE cursor statement. This closes the cursor in the stored procedure as well.

- You can allocate only one cursor to each result set.

3. **Mortality of an allocated cursor:**

   A rollback and an implicit and explicit close will destroy allocated cursors. A commit destroys allocated cursors that are not defined WITH HOLD by the stored procedure. However, note that DB2 Server for VSE & VM does not support CURSOR WITH HOLD. Destroying an allocated cursor closes the associated cursor in the stored procedure.
4. For the ALLOCATE CURSOR statement to be successful, the application must be connected to the site at which the stored procedure was executed.

Examples

The statement in the following example is assumed to be in a PL/I program.

Define and associate cursor C1 with the result set locator variable :loc1 and the related result set returned by the stored procedure:

```
EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1
```

ALTER DBSPACE

The ALTER DBSPACE statement lets you change the amount of free space that the database manager reserves on each data page, and lets you change the type of a lock on a public dbspace.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- For a private dbspace:
  Ownership of the dbspace or
  DBA authority.
- For a public dbspace:
  DBA authority.

Syntax

```
                                          ┌─,──────────────────────────────────┐
                                          │  (1)                               │
►►──ALTER DBSPACE──dbspace_name──(────────▼──────────┬─PCTFREE = integer───┬──┴──)───────►◄
                                                     └─LOCK =──┬─PAGE──────┤
                                                               ├─DBSPACE───┤
                                                               └─ROW───────┘
```

Notes:

1 If either of these clauses is specified more than once, the value with the first specification is used.

Description

dbspace_name
Identifies the dbspace to be changed. It must be a dbspace that exists at the application server.

PCTFREE
Is the percentage of space on each page that the database manager is to keep empty when inserting data into the dbspace's tables. A common practice is to set PCTFREE to a higher value when a dbspace is acquired, load the data, and create an index defined in the same order as the data was loaded. After this process is complete, the PCTFREE is lowered. Some or all of the free space is now available for inserts. The judicious use of reserved free space may result in a more favorable placement of data on pages and, therefore, improve access time.

LOCK
Alters the lock size of a public dbspace. The valid lock sizes are DBSPACE, PAGE, and ROW. If DBSPACE is specified, the system locks the whole dbspace. Page causes the smallest lockable unit to be a page (4096 bytes); ROW causes this unit to be a row.

Examples

Example 1

Alter your private dbspace named FCPSPACE so that no space is reserved on any of the pages.

```
ALTER DBSPACE FCPSPACE  (PCTFREE=0)
```

Example 2

Alter a public dbspace named SPACE so that the pages are locked and the amount of free space is reduced to 3%.

```
ALTER DBSPACE PUBLIC.SPACE
  (PCTFREE=3, LOCK=PAGE)
```

ALTER PROCEDURE

The ALTER PROCEDURE statement is used to alter the definition of an existing stored procedure. It updates the catalog and the corresponding cached information.

The STOP PROC command must be issued with the REJECT option before the ALTER PROCEDURE statement will be accepted.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer of the ALTER PROCEDURE must have DBA authority.

Syntax

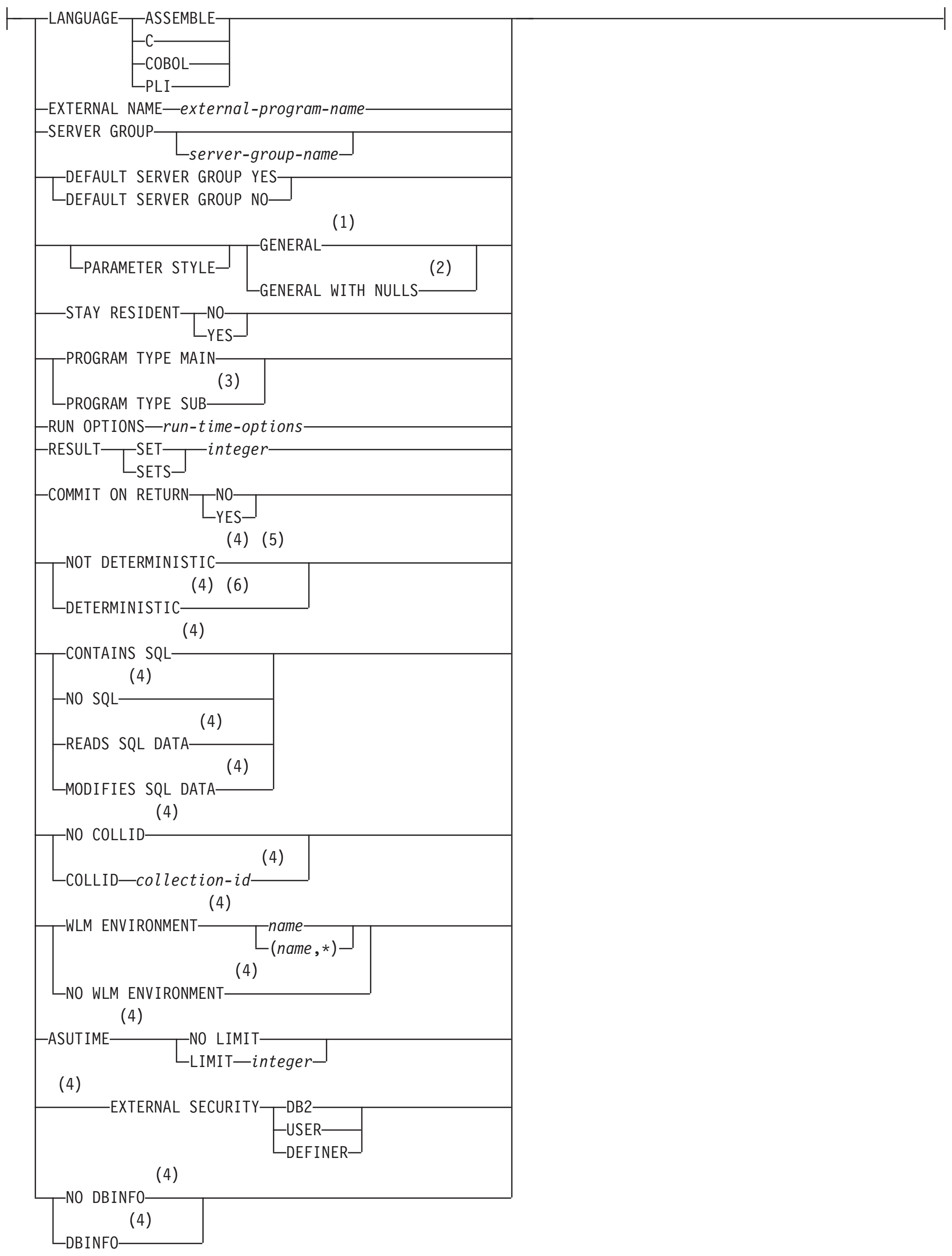
ALTER PROCEDURE

```
►►──ALTER PROCEDURE──procedure-name──┬─────────────────┬──────────────────────►
                                     └─AUTHID──authid──┘


    ┌─,───────────┐
    ▼             │ (1)
►───┬─┤ options ├─┴───────────────────────────────────────────────────────────►◄
```

Notes:

1 One or more clauses may be specified, however each clause may be specified at most once.

options:



Notes:

1 SIMPLE CALL may be used as an alternative to GENERAL. This is for compatibility within the DB2 family.

2 SIMPLE CALL WITH NULLS may be used as an alternative to GENERAL WITH NULLS. This is for compatibility within the DB2 family.

3 Currently, DB2 Server for VSE & VM supports stored procedures written as main programs only.

4 This parameter is included for compatibility with the DB2 family. If specified, it is ignored.

5 VARIANT may be specified as an alternative to NOT DETERMINISTIC. This is for compatibility within the DB2 family.

6 NOT VARIANT may be specified as an alternative to DETERMINISTIC. This is for compatibility within the DB2 family.

Only the parameters that are meaningful to DB2 Server for VSE & VM are described here. If a parameter is not specified on the ALTER PROCEDURE statement, its value is unchanged.

Description

procedure-name
Names the stored procedure. For DB2 Server for VSE & VM, the name must be an ordinary identifier of 18 characters or less.

authid
The authorization ID for the stored procedure. The *authid* must be an ordinary identifier of 8 characters or less. If specified, then only the version of *procedure-name* that is accessible only by *authid* will be altered.

LANGUAGE
Specifies the programming language used to create the stored procedure. All stored procedure programs must be designed to run in the IBM Language Environment.

ASSEMBLE
Specifies that the stored procedure is written in Assembler.

C Specifies that the stored procedure is written in C.

COBOL
Specifies that the stored procedure is written in COBOL.

PLI Specifies that the stored procedure is written in PLI.

EXTERNAL NAME *external-program-name*
Identifies the load module or phase associated with the stored procedure. The *external-program-name* must be an ordinary identifier of 8 characters or less. The load module or phase does not need to exist when the ALTER PROCEDURE statement is issued. However, when a CALL for the stored procedure is issued, the load module must exist and be accessible to the stored procedure server.

SERVER GROUP *server-group-name*
Identifies the group of stored procedure servers in which this stored procedure will run. If specified, *server-group-name* must be an ordinary identifier of 18 characters or less. *server-group-name* must be defined in SYSTEM.SYSPSERVERS.

The SERVER GROUP clause can be specified without a server group name. This provides the ability to take a stored procedure out of a named group and move it to the default group. If *server-group-name* is not specified, the stored procedure must be able to run in the default group. The DEFAULT SERVER GROUP clause determines whether the stored procedure can run in the default stored procedure server group.

DEFAULT SERVER GROUP
Specifies whether the stored procedure can run in the default server group.

YES The stored procedure can run in the default server group.

NO The stored procedure cannot run in the default server group. If NO is specified, the SERVER GROUP clause must have been provided on the CREATE PROCEDURE statement, or it must be provided on the ALTER PROCEDURE statement.

PARAMETER STYLE
Identifies the linkage convention used to pass parameters to the stored procedure. All of the linkage conventions provide arguments to the stored procedure containing the parameters specified on the SQL CALL statement. See the *DB2 Server for VSE & VM Database Administration* manual for more information. The following parameter styles options are valid for DB2 Server for VSE & VM:

GENERAL
If the GENERAL linkage convention is used:

- the SQL CALL statement must provide a parameter for each parameter expected by the stored procedure
- input parameters cannot be null
- nulls can be passed for output parameters only
- the stored procedure cannot return nulls for output parameters

GENERAL WITH NULLS
If the GENERAL WITH NULLS linkage convention is used:

- the SQL CALL statement must provide a parameter for each parameter expected by the stored procedure. When the database manager invokes the stored procedure, it sends it the parameters specified on the SQL CALL statement, as well as an array of indicator variables (with one indicator variable for each parameter). The stored procedure must contain a declaration for this array.
- input parameters can be null. This is achieved through the use of indicator variables, or by specifying the keyword null.
- the stored procedure can return nulls for output parameters, by using indicator variables.

STAY RESIDENT
Specifies whether the stored procedure load module or phase remains loaded in memory after the stored procedure ends. Possible values are:

NO The load module or phase is deleted from memory after the stored procedure ends.

YES The load module or phase remains loaded in memory after the stored procedure ends.

PROGRAM TYPE
Specifies whether the stored procedure runs as a MAIN routine or as a SUB routine. Currently, DB2 Server for VSE & VM supports only stored procedures written as MAIN routines. If PROGRAM TYPE SUB is specified, DB2 Server for VSE & VM will override it with PROGRAM TYPE MAIN.

RUN OPTIONS
Specifies the Language Environment run-time options to be passed to the stored procedure. The options must be specified as a character string up to 254 bytes enclosed in single quotation marks. If this option is not specified, or an empty string is passed, then DB2 Server for VSE & VM passes no run-time options to the Language Environment, and Language Environment uses its installation defaults. Note that DB2 Server for VSE & VM does not do any

checking of the options provided. For a complete description of Language Environment run-time options, see *Language Environment for MVS & VM Programming Reference*.

RESULT SETS or RESULT SET
Specifies the maximum number of query result sets that can be returned by this stored procedure. The largest value that can be specified is 32767.

COMMIT ON RETURN
Indicates whether the unit of work should be committed immediately upon return from the stored procedure.

NO The database manager should not issue COMMIT when the stored procedure returns.

YES The database manager should issue COMMIT when the stored procedure returns when the following statements are true:
- The SQLCODE returned by the CALL statement is not negative
- The stored procedure is not in a *must abort* state

The COMMIT operation includes the work performed by the calling application as well as the stored procedure. Any cursors that are open when the COMMIT occurs will be closed during COMMIT processing.

Examples

Example 1

```
ALTER PROCEDURE MYPROC STAY RESIDENT NO
```

ALTER PSERVER

The ALTER PSERVER statement alters the definition of an existing stored procedure server.

The STOP PSERVER command must be issued with the NOIMPLICIT option before the ALTER PSERVER statement will be accepted.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer of the ALTER PSERVER statement must have DBA authority.

Syntax

```
                                        ┌,─────────────────────────────────────┐
                                        │ (1)                                  │
►►──ALTER PSERVER──procedure-server─────▼──────┬─GROUP──┬──────────────┬──┬────┴─────────►◄
                                               │        └─group-name───┘  │
                                               ├──AUTOSTART NO──┬─────────┤
                                               │ └AUTOSTART YES─┘         │
                                               └─DESCRIPTION──description─┘
```

Notes:

1 One or more clauses may be specified, however each clause may be specified at most once.

Description

procedure-server
The name of the stored procedure server. This must be an ordinary identifier of 8 characters or less.

GROUP
The name of the group that this stored procedure server will be in after the ALTER PSERVER statement has been executed. If a group name is specified, it must be an ordinary identifier of 1 to 18 characters. If the GROUP clause is specified without *group-name*, the stored procedure server will be put in the default group.

group-name
The name of the stored procedure group. It cannot be any of the following:
GROUP
IMPLICIT
NOIMPLICIT
NORMAL
QUICK

AUTOSTART
Determines whether the database manager will issue a START PSERVER command for this stored procedure server when the database is started.

NO START PSERVER will not be issued when the database is started.

YES START PSERVER will be issued when the database is started.

DESCRIPTION
This field provides the database administrator with a place to provide information about this stored procedure server, such as virtual storage requirements, other servers in the group, and so on. *Description* can be up to 254 characters and must be enclosed in single quotation marks.

Examples

Example 1

```
ALTER PSERVER SRV1 GROUP GRP2, AUTOSTART NO
```

ALTER TABLE

The ALTER TABLE statement adds a single column to an existing table, and adds, drops, activates, or deactivates primary and foreign keys.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table
- The ALTER privilege for the table
- DBA authority.

To create, drop, activate, or deactivate a foreign key, the authorization ID of the statement must also hold at least one of the following on the parent table:
- Ownership of the table
- The REFERENCES privilege for the table
- Administrative authority.

To drop, activate, or deactivate a primary key, the authorization ID of the statement must also hold at least one of the following on each table that has a foreign key referencing the primary key that is being dropped.
- Ownership of the table
- The ALTER privilege for the table
- DBA authority.

Syntax

```
►►──ALTER TABLE──table_name──┬─ADD──┤ column-definition-block ├─────────────┬──────────────────►◄
                             │ ┌─ADD─┐                                      │
                             ├─┴─────┴──┬─┤ primary-key-block ├─────────┬───┤
                             │          ├─┤ referential-constraint-block ├┤  │
                             │          └─┤ unique-block ├──────────────┘   │
                             ├─DROP───┬─PRIMARY KEY───────────────────────────┤
                             │        ├─FOREIGN KEY──constraint_name─┤       │
                             │        └─UNIQUE──constraint_name──────┘       │
                             ├─ACTIVATE───┬─ALL─────────────────────────────┤
                             │            ├─PRIMARY KEY─────────────────┤    │
                             │            ├─FOREIGN KEY──constraint_name─┤   │
                             │            └─UNIQUE──constraint_name──────┘   │
                             ├─DEACTIVATE───┬─ALL───────────────────────────┤
                             │              ├─PRIMARY KEY─────────────────┤  │
                             │              ├─FOREIGN KEY──constraint_name─┤ │
                             │              └─UNIQUE──constraint_name──────┘ │
                             └─DATA CAPTURE───┬─NONE────┬───────────────────┘
                                              └─CHANGES─┘
```

column-definition-block:

```
├──column_name──┤ data-type ├──┬──────────────────────────────┬───────────────────────┤
                               │        (1)                   │
                               └───────────┤ fieldproc-block ├┘
```

data-type:

```
├──┬─INTeger──────────────────────────────────────────────────┬────────────────────────┤
   ├─SMALLINT─────────────────────────────────────────────────┤
   │       ┌─(53)──────┐                                       │
   ├─┬─FLOAT─┴─(integer)─┴──┬──────────────────────────────────┤
   │ ├─REAL────────────────┤                                   │
   │ └─DOUBLE PRECISION────┘                                   │
   │                 ┌─(5,0)───────────────────────────┐       │
   ├─┬─DECimal─┬─────┴─(──integer──┬───────────┬──)──┴───────┤
   │ └─NUMERIC─┘                   └─,integer──┘               │
   │               ┌─(1)─────────┐          (1)                │
   ├─┬─CHARacter───┴─(integer)──┴─┬──────┬─────────────────┬───┤
   │ ├─VARCHAR──(integer)─────────┤      ├─FOR SBCS DATA───┤   │
   │ └─LONG VARCHAR───────────────┘      ├─FOR MIXED DATA──┤   │
   │                                     ├─FOR BIT DATA────┤   │
   │                                     └─CCSID──integer──┘   │
   │             ┌─(1)─────────┐                               │
   ├─┬─GRAPHIC───┴─(integer)──┴─┬──┬─────────────────────┬─────┤
   │ ├─VARGRAPHIC──(integer)────┤  │ (1)                 │     │
   │ └─LONG VARGRAPHIC──────────┘  └───────CCSID──integer─┘     │
   ├─DATE──────────────────────────────────────────────────────┤
   ├─TIME──────────────────────────────────────────────────────┤
   └─TIMESTAMP─────────────────────────────────────────────────┘
```

Notes:

1 These clauses may be specified in any order.

fieldproc-block:

```
├──FIELDPROC──program_name──┬──────────────────────┬────────────────────────────────────────┤
                            │    ┌─,──────────┐     │
                            └─(──▼─constant───┴──)─┘
```

primary-key-block:

```
                   ┌─,─────────────────────────────────┐
                   ▼   (1)                   ┌─ASC──┐    │      ┌─PCTFREE = 10──────┐
├──PRIMARY KEY──(──────────column_name──┼──────┼────┴──)──┼───────────────────┼──────────┤
                                             └─DESC─┘           └─PCTFREE = integer─┘
```

Notes:

1 There can be up to 16 columns in a primary key.

referential-constraint-block:

```
                                                ┌─,────────────┐
├──FOREIGN KEY──┬─────────────────┬──(──▼─column_name─┴──)──REFERENCES──table_name──────────►
                └─constraint_name─┘

►──┬──────────────────────────────┬─────────────────────────────────────────────────────┤
   │              ┌─RESTRICT─┐    │
   └─ON DELETE────┼─CASCADE──┼────┘
                  └─SET NULL─┘
```

unique-block:

```
                                          ┌─,─────────────────────────────────┐
                                          ▼   (1)                   ┌─ASC──┐    │      ┌─PCTFREE = 10──────┐
├──UNIQUE──┬─────────────────┬──(──────────column_name──┼──────┼────┴──)──┼───────────────────┼──────┤
           └─constraint_name─┘                                      └─DESC─┘           └─PCTFREE = integer─┘
```

Notes:

1 There can be up to 16 columns on a unique constraint.

Description

table_name
: Identifies the table to be changed. It must be a table that exists at the application server and must not be a view or a catalog table. If the *table_name* is qualified, the qualifier is the owner of the table. Otherwise, the authorization ID of the statement is the owner of the table.

ADD
: Adds a column to the table. All column values are NULL and the column is the *last* table column on the rightmost side. That is, if initially there are *n* columns, the added column is column *n*+1. The value of *n* cannot be greater than 254.

Adding the new column must not make the total byte count of all columns exceed the maximum record size of approximately 4072 bytes. For more information, see “Notes” on page 228.

column_definition_block

column_name
Names the column to be added to the table. The name cannot already be used by an existing column of the table.

data_type
Is one of the data types in the descriptions listed under “CREATE TABLE” on page 219.

fieldproc_block

FIELDPROC *program_name*
Names a field procedure for the column. A field procedure may be used only with a short string column. If FIELDPROC is omitted, the column has no field procedure.

constant
Is a parameter passed to the field procedure when the ALTER TABLE statement invokes it. A parameter list is optional. The number of parameters and the data type of each are determined by the field procedure. The maximum length of the parameter list is 254 bytes, including commas, but excluding insignificant blanks and excluding the delimiting parentheses after blank compression takes place.

primary_key_block

PRIMARY KEY
Is a set of column values in the table that enforces a unique constraint. Only one primary key is allowed in a parent table. Primary key values must be unique and must be defined as NOT NULL.

Defining a primary key on a table sets up the table to be referenced by another table’s foreign key to establish a referential constraint.

column_name
Identifies the column or columns that comprise the primary keys. Each *column_name* must be an unqualified name that identifies a column of the table. No column in a primary key can contain a long string. The same column cannot be specified more than once.

ASC
Creates the primary key such that the values from this column are arranged in ascending order. This is the default.

DESC
Creates the primary key such that the values from this column are arranged in descending order.

PCTFREE
Is the percentage of space in each index page reserved for later insertions and updates of primary keys. The integer may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take up more space, but reduces the time required to insert or update primary key rows of the indexed table.

referential_constraint_block

FOREIGN KEY
Defines a foreign key composed of the identified columns. Consists of one or more columns in this dependent table that together must take on a value that exists in the primary key of the referenced parent table. The columns in the dependent table may contain nulls. If any of the columns contain a null value, the foreign key is considered null.

constraint_name
Provides a name for the referential constraint. A *constraint_name* cannot be used more than once in the same table. Although the database manager generates a constraint_name if one is not specified, a *constraint_name* should be explicitly chosen to make it easier for a user to drop, activate, and deactivate the foreign key.

column_name
Identifies the column or columns that comprise the foreign key. Each *column_name* must be an unqualified name that identifies a column of the table. The data type and length of foreign key columns must match the data type and length of the primary key columns. Only the null attribute of a foreign key column may be different. The same column cannot be specified more than once.

REFERENCES *table_name*
Specifies the name of the parent table involved in the referential constraint. The *table_name* cannot identify the table that is being altered.

ON DELETE
Defines the delete rule to be followed when a row is deleted from the parent table in a relationship.

RESTRICT
Prevents deletion of a parent row until all the dependent rows have been deleted. This is the default.

CASCADE
Causes all dependent rows to be deleted also.

SET NULL
Sets to null all columns of the foreign key values in each dependent row that can contain nulls. At least one column of the foreign key in the dependent table must be able to contain nulls.

The following restrictions for ON DELETE are checked when a table is altered.

- If a table has more than one referential constraint referencing the same parent, all the delete rules on those constraints must be the same and must not be SET NULL.
- If a table is delete-connected to the same parent through multiple paths, all of the delete rules on the paths, except for the last one, must be CASCADE. The last delete rule on all paths must be the same and must not be SET NULL.
- A referential cycle involving two or more tables must not cause a table to be delete-connected to itself.

For additional information and examples of application restrictions see "Definition Restrictions" on page 16.

unique_block

UNIQUE
Adds a unique index automatically for the column or columns specified. If there are duplicates in the values of the columns, then a unique constraint is not added.

constraint_name
Provides a name for the unique constraint. A *constraint_name* cannot be used more than once in the same table. Although the database manager generates a constraint_name if one is not specified, a *constraint_name* should be explicitly chosen to make it easier for a user to drop, activate, and deactivate the unique constraint.

column_name
Identifies the column or columns that comprise the unique key. Each *column_name* must be an unqualified name that identifies a column of the table. No column in a unique constraint can be nullable. No column in a unique constraint can contain a long string. The same column cannot be specified more than once. These columns should not be the same as that of a primary key in the same table.

ASC
Creates the unique key such that the values from this column are arranged in ascending order. This is the default.

DESC
Creates the unique key such that the values from this column are arranged in descending order.

PCTFREE
Is the percentage of space in each index page reserved for later insertions and updates of unique keys. The integer may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take up more space, but reduces the time required to insert or update unique keys.

DROP PRIMARY KEY
Drops the definition of the primary key, thereby removing all referential constraints in which the table is a parent. Dropping a primary key causes the foreign keys that reference the parent table to be dropped.

DROP FOREIGN KEY *constraint_name*
Drops the definition of the foreign key, thereby removing the named referential constraint.

DROP UNIQUE *constraint_name*
Drops the unique index associated with the constraint and the information in the system catalog tables.

ACTIVATE ALL
Causes all the referential constraints defined for a primary key to be enforced automatically. ACTIVATE ALL is equivalent to activating the primary key, then activating all the explicitly inactive foreign keys and unique constraints.

ACTIVATE PRIMARY KEY
Causes the primary key to be enforced automatically. If the primary key is already active, this clause drops and re-creates the primary key index. If the primary key is inactive, then the primary key index is re-created first. If any dependent foreign keys are deactivated implicitly when the primary

key is made inactive, those foreign keys are verified against the primary key. If the primary key index is created successfully and the dependent foreign key values are found in the primary key of the object table, then the primary key and the dependent foreign keys are activated. None of the keys are activated if an error occurs.

ACTIVATE FOREIGN KEY *constraint_name*
Causes the referential constraint defined by the named foreign key to be enforced automatically. If the primary key of the parent table referenced by this foreign key is inactive, the foreign key is not activated. If the associated primary key is active, the foreign key values are verified against the values in the primary key. If all values are found in the parent primary key, the dependent foreign key is activated.

ACTIVATE UNIQUE *constraint_name*
Activates a unique key on an existing table.

DEACTIVATE ALL
Suspends the restrictions imposed by the referential constraints and makes the parent and dependent tables involved in a referential constraint unavailable to users other than the DBA and the owner of the table. All primary and foreign keys become inactive. DEACTIVATE ALL is equivalent to deactivating the primary key, all active foreign keys in the table, and all unique constraints.

DEACTIVATE PRIMARY KEY
Suspends the restrictions imposed by the referential constraints and makes the parent and dependent tables involved in a referential constraint unavailable to users other than the DBA and the owner of the table. Deactivating a primary key drops the primary key index from the object table and implicitly deactivates all active dependent foreign keys.

DEACTIVATE FOREIGN KEY *constraint_name*
Suspends the restrictions imposed by the referential constraints and makes the parent and dependent tables involved in a referential constraint unavailable to users other than the DBA and the owner of the table.

DEACTIVATE UNIQUE *constraint_name*
Deactivates a unique key on an existing table.

DATA CAPTURE
Specifies if log records for this table should contain the full before image (DATA CAPTURE CHANGES) or the partial before image (DATA CAPTURE NONE) for UPDATE operations. If this option is not specified, it defaults to DATA CAPTURE NONE. If DataPropagator Capture is being used to capture changes to this table, DATA CAPTURE CHANGES must be specified. If DataPropagator Capture is not being used to capture updates to this table, DATA CAPTURE NONE should be specified to reduce the amount of data logged for updates to this table.

NONE
Include the partial before image in log records for UPDATE operations. If DataPropagator Capture is not being used to capture updates to this table, DATA CAPTURE NONE should be specified to reduce the amount of data logged for updates to this table.

CHANGES
Include the full before image in log records for UPDATE operations. If DataPropagator Capture is being used to capture changes to this table, DATA CAPTURE CHANGES must be specified.

Notes

It is not possible to:
- Use NOT NULL. All values in a new column are NULL when created.
- See an added column in any existing view of the table.
- Change the name of a column unless the table is dropped and recreated with the new column name.
- Add a PRIMARY KEY, FOREIGN KEY, or UNIQUE constraint on any catalog table.

It is not a good practice to:
- Duplicate a referential constraint in the same table (that is, to have two foreign keys with the same column list referencing the same table).
- Duplicate a unique constraint in the same table (that is, to have two unique constraints with the same column list in the same table).
- Have a unique constraint with the same columns as the primary key of the same table.

In these cases, a warning is issued but the duplicate specification is accepted.

Adding, dropping, activating, or deactivating keys invalidates the packages that access tables affected by these changes in the keys. When an SQL statement attempts to invoke an incorrect package, the database manager tries to dynamically rebind the package.

The characteristics of a primary key or foreign key cannot be directly altered. All specifications of the key must first be dropped and then respecified.

Examples

Example 1

Add a new column named RATING, which is one character long, to the DEPARTMENT table.

```
ALTER TABLE DEPARTMENT
  ADD RATING CHAR
```

Example 2

Add a new column named SITE_NOTES to the PROJECT table. Create SITE_NOTES as a varying-length column with a maximum length of 1000 characters. The values of the column do not have an associated character set and therefore should not be translated.

```
ALTER TABLE PROJECT
  ADD SITE_NOTES  VARCHAR(1000) FOR BIT DATA
```

Example 3

Assume a new table EQUIPMENT has been created with the following columns:

Column Name	Data Type
EQUIP_NO	INT
EQUIP_DESC	VARCHAR(50)
LOCATION	VARCHAR(50)
EQUIP_OWNER	CHAR(3)

Add a referential constraint to the EQUIPMENT table so that the owner (EQUIP_OWNER) must be a department number (DEPTNO) that is present in the DEPARTMENT table. If a department is removed from the DEPARTMENT table, the owner (EQUIP_OWNER) values for all equipment owned by that department should become unassigned (or set to null). Give the constraint the name DEPT_EQUIP.

```
ALTER TABLE EQUIPMENT
  ADD FOREIGN KEY DEPT_EQUIP (EQUIP_OWNER)
    REFERENCES DEPARTMENT
    ON DELETE SET NULL
```

Example 4

Add a constraint to the PROJECT table to ensure that there are not two entries in the table with the same value for project name (PROJNAME).

```
ALTER TABLE PROJECT
  ADD UNIQUE (PROJNAME)
```

See example 1 in “CREATE INDEX” on page 198 for an alternate method of ensuring unique project names.

Example 5

Alter a table to create log records with the partial before image for UPDATE operations where DataPropagator Capture is not capturing updates for the table:

```
ALTER TABLE SALARY1
  DATA CAPTURE NONE
```

Example 6

Alter a table to create log records with the full before image for UPDATE operations because DataPropagator Capture requires this information for update log records:

```
ALTER TABLE SALARY2
  DATA CAPTURE CHANGES
```

ASSOCIATE LOCATORS

The ASSOCIATE LOCATORS statement obtains the RESULT SET LOCATOR value for each result set data type returned by a stored procedure.

Invocation

This statement can be embedded in an application program. It is an executable statement that can be dynamically prepared. It cannot by issued interactively.

Authorization

None required.

Syntax

```
►►──ASSOCIATE──┬──────────────┬──┬─LOCATOR──────┬──────────────────────────────►
               └─┤ RESULT SET ├─┘  │         (1) │
                                   └─LOCATORS─────┘

       ┌─,────────────────────┐
       ▼                      │
►─(────rs-locator-variable────┴──)──WITH PROCEDURE──┬─host-variable───┬───────────►◄
                                                    └─procedure-name──┘
```

Notes:

1 RESULT SET LOCATOR variables are only supported in client applications written in Assembler, C, COBOL, and PL/I.

Description

rs-locator-variable
: Identifies a result set locator variable that has been declared according to the rules for declaring result set locator variables. One result set locator variable is required for each result set that is returned by a stored procedure. If a stored procedure returns fewer result sets than the number of result set locator variables specified, then the extra variables are assigned a value of zero.

WITH PROCEDURE *host-variable* **or** *procedure-name*
: Identifies the stored procedure that returns result set locators. The procedure name may be specified either directly or within a host variable.

If a *host-variable* is specified, it must be a character-string variable and it must not include an indicator variable. Note that the value is not converted to uppercase.

If *procedure-name* is specified, it must be an ordinary identifier, which implies that it cannot contain blanks or special characters, and the value is converted to uppercase. Therefore, if it is necessary to use a lowercase name that contains blanks or special characters, then the name must be specified in a host-variable. The procedure name must be left-justified. The form in which a procedure name exists varies according to the server where the procedure is stored.

DB2 Server for VSE & VM:
The name of the procedure to execute. The name can be up to 18

characters long and must match a value in the NAME column of the SYSTEM.SYSROUTINES catalog table.

DB2 Common Server/DB2 Universal Database (except OS/390 and OS/400):

procedure-name
The name (with no extension) of the procedure to execute. This is used both as the name of the stored procedure library and the function name within that library.

procedure-library!function-name
The exclamation point character acts as a delimiter between the library name and the function name of the stored procedure.

absolute-path!function-name
The absolute-path specifies the complete path to the stored procedure library.

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.

DB2 Universal Database Server for OS/390:
An implicit or explicit three-part name. The parts are as follows:

high order
The location name of the server where the procedure is stored.

middle
SYSPROC

low order
Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.

DB2 Universal Database Server for OS/400:
The external program name is assumed to be the same as the procedure name.

For portability, the procedure name should be specified as a single token no larger than eight bytes.

The ASSOCIATE LOCATORS statement can only be executed against a stored procedure that has already been invoked by the program using the SQL CALL statement.

Notes

1. More than one locator can be assigned to a result set. The same ASSOCIATE LOCATORS statement can be issued more than once with different result set locator variables.
2. If the number of result set locator variables listed in the ASSOCIATE LOCATORS statement is less than the number of result sets returned by the stored procedure, all variables in the statement are assigned a value, and a warning is issued.

   If the number of result set locator variables listed in the ASSOCIATE LOCATORS statement is more than the number of locators returned by the stored procedure, then the extra variables are assigned a value of zero.
3. The ASSOCIATE LOCATORS statement assigns result set locator values to result set locator variables from the SQLVAR sections of the SQLDA. The first SQLDATA field is assigned to the first locator variable, the second SQLDATA field to the second locator variable, and so on.

4. For the ASSOCIATE LOCATORS statement to be successful, the application must be connected to the site at which the stored procedure was executed.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1

Use :loc1 and :loc2 to obtain the result set locator values for the two result sets returned by stored procedure P1:

```
EXEC SQL ASSOCIATE RESULT SET LOCATORS (:loc1, :loc2)
   WITH PROCEDURE P1;
```

Example 2

Use :loc1 and :loc2 to obtain the result set locator values for the two result sets returned by the stored procedure named by host variable :hv1:

```
EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2)
   WITH PROCEDURE :hv1;
```

BEGIN DECLARE SECTION

The BEGIN DECLARE SECTION statement marks the beginning of an SQL declare section where host variables must be defined.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It is not supported in REXX.

Authorization

None required.

Syntax

```
►►──BEGIN DECLARE SECTION──────────────────────────────────────────────────►◄
```

Description

The BEGIN DECLARE SECTION statement can be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. An SQL declare section ends with an END DECLARE SECTION statement, described on page “END DECLARE SECTION” on page 263.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

SQL statements (other than the 'INCLUDE *text-file-name*' form of the INCLUDE statement) cannot be specified within an SQL declare section.

In programs other than REXX, all variables referenced in SQL statements must be declared in one or more SQL declare sections. With the exception of Assembler, the SQL declare section must appear before the first reference to the variable. In REXX, host variables are declared without the use of these statements; meaning they are implicitly declared.

Variables declared outside an SQL declare section must not have the same name as variables declared within an SQL declare section.

Examples

Example 1

In an **Assembler** program, define the host variables HVSMINT (smallint), HVVCHR24 (varchar(24)), and HVDEC72 (dec(7,2)).

```
     EXEC SQL BEGIN DECLARE SECTION
HVSMINT  DS    H
HVVCHR24 DS    H,CL24
HVDEC72  DS    PL4'12345.67'
     EXEC SQL END DECLARE SECTION
```

Example 2

In a **C** program, define the host variables hv_smint (smallint), hv_vchar24 (varchar(24)), hv_double (float), and host structure name_structure (char(9),char(9)).

```
EXEC SQL  BEGIN DECLARE SECTION;
  static short                          hv_smint;
  static struct hv_char {
```

```
            short hv_vchar24_len;
            char  hv_vchar24_value[24];
          }                              hv_vchar24;
   static double                         hv_double;
   static struct name_struct {
            char  lname[9];
            char  fname[9];
          }                              name_structure;
 EXEC SQL  END DECLARE SECTION;
```

Example 3

In a **COBOL** program, define the host variables HV-SMINT (smallint), HV-VCHAR24 (varchar(24)), HV-DEC72 (dec(7,2)), and host structure NAME-STRUCTURE (char(9),char(9)).

```
 WORKING-STORAGE SECTION.
       EXEC SQL BEGIN DECLARE SECTION  END-EXEC.
 01  HV-SMINT              PIC S9(4)       COMP-4.
 01  HV-VCHAR24.
     49 HV-VCHAR24-LENGTH  PIC S9(4)       COMP-4.
     49 HV-VCHAR24-VALUE   PIC X(24).
 01  HV-DEC72              PIC S9(5)V9(2)  COMP-3.
 01  NAME-STRUCTURE.
     05 FNAME              PIC X(9).
     05 LNAME              PIC X(9).
       EXEC SQL END DECLARE SECTION  END-EXEC.
```

Example 4

In a **Fortran** program, define the host variables HVSMINT (smallint), HVCHAR24 (char(24)), and HVDOUBLE (float).

```
 EXEC SQL  BEGIN DECLARE SECTION
   INTEGER*2      HVSMINT
   CHARACTER*24   HVCHAR24
   REAL*8         HVDOUBLE
 EXEC SQL  END DECLARE SECTION
```

Note: Because varying-length character strings are not supported in Fortran, a character host variable large enough to use the largest expected value must be used.

Example 5

In a **PL/I** program, define the host variables HV_SMINT (smallint), HV_VCHAR24 (varchar(24)), HV_DEC72 (dec(7,2)), and host structure NAME_STRUCTURE (char(9),char(9)).

```
 EXEC SQL  BEGIN DECLARE SECTION;
   DCL  HV_SMINT     BINARY    FIXED(15);
   DCL  HV_VCHAR24   CHAR(24)  VARYING;
   DCL  HV_DEC72     FIXED     DECIMAL(7,2);
   DCL  01 NAME_STRUCTURE,
           05 FNAME     CHAR(9),
           05 LNAME     CHAR(9);
 EXEC SQL  END DECLARE SECTION;
```

CALL

The CALL statement invokes a stored procedure. The database manager uses the cached information from SYSTEM.SYSROUTINES, SYSTEM.SYSPARMS, and SYSTEM.SYSPSERVERS to process the statement.

Invocation

This statement must be embedded in an application program. It is an executable statement that cannot be dynamically prepared. However, a host variable can be specified for the procedure-name, enabling the procedure name to be resolved at run time.

Authorization

The privileges required to execute the CALL statement are determined by the application server and must be held by the owner of the package containing the CALL statement. If the server is DB2 Server for VSE & VM, that authorization ID must have at least one of the following for each of the packages associated with the stored procedure:

- Run privilege on the package
- Ownership of the package
- DBA authority

Syntax

```
►►──CALL──┬─procedure-name─┬──┬─────────────────────────────────────┬──────────►◄
          └─host-variable──┘  ├─(──┬──────────────────────────┬──)──┤
                              │    │  ┌─,──────────────────┐  │     │
                              │    │  ▼                    │  │     │
                              │    └────┬─host-variable─┬──┴──┘     │
                              │         ├─constant──────┤           │
                              │         └─NULL──────────┘           │
                              └─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.

  If procedure-name is specified it must be an ordinary identifier, which implies that it cannot contain blanks or special characters, and that the value is converted to upper case. If it is necessary to use lower case names, blanks, or special characters, the name must be specified in a host-variable.

  If a host-variable is specified, it must be a character-string variable and it must not include an indicator variable. Note that the value is not converted to upper case. Procedure-name must be left-justified.

  The procedure name can take one of several forms. The forms supported vary according to the server at which the procedure is stored.

  - DB2 Server for VSE & VM:

    The name of the procedure to execute. The name can be up to 18 characters long, and must match a value in the NAME column of the SYSTEM.SYSROUTINES catalog table.

- DB2 common server / DB2 Universal Database (except OS/390 and OS/400):

  procedure-name
  The name (with no extension) of the procedure to execute. This is used both as the name of the stored procedure library and the function name within that library.

  procedure-library!function-name
  The exclamation character (!) acts as a delimiter between the library name and the function name of the stored procedure.

  absolute-path!function-name
  The absolute-path specifies the complete path to the stored procedure library.

  In all these cases, the total length of the procedure name including its implicit or explicit full path must not be longer than 254 bytes.
- DB2 Universal Database Server for OS/390:

  An implicit or explicit three part name. The parts are as follows:

  high order
  The location name of the server where the procedure is stored.

  middle
  SYSPROC

  low order
  Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.
- DB2 Universal Database Server for OS/400:

  The external program name is assumed to be the same as the procedure-name

For portability, procedure-name should be specified as a single token no larger than 8 bytes. Note that when the SQL CALL statement is preprocessed, the database manager does not check whether the procedure is defined, or whether the caller is authorized to invoke it. This checking is done at run time only.

Parameters (host variable, constant, or NULL)
Identifies a list of values to be passed as parameters to the procedure.

Each specification of a host-variable, constant, or NULL is a parameter of the CALL. If USING DESCRIPTOR is specified, each host variable described by the identified SQLDA is a parameter of the CALL. The nth parameter of the CALL corresponds to the nth parameter of the stored procedure. When the CALL statement is executed, the number of parameters of the CALL must be the same as the number of parameters expected by the stored procedure, and each pair of corresponding parameters must be consistent as explained below.

Each parameter of the stored procedure is defined at the server. In addition to attributes such as data type and length, the description of each parameter indicates how it is used by the stored procedure:
- IN means the parameter is used only as an input value
- OUT means the parameter is used only as an output value
- INOUT means the parameter is used as both an input and an output value

DB2 Server for VSE & VM gets the parameter descriptions from the cached information from the new catalog table SYSTEM.SYSPARMS.

Other servers might acquire parameter descriptions from other sources such as the SQL DECLARE PROCEDURE statement.

When the CALL statement is executed, the value of each parameter of the CALL defined as IN or INOUT is assigned to the corresponding parameter of the stored procedure in accordance with the DB2 Server for VSE & VM rules for assigning values to host variables. Control is then passed to the stored procedure in accordance with the calling conventions of the host language. When execution of the stored procedure is complete, the value of each parameter defined as OUT or INOUT is assigned to the corresponding parameter of the CALL in accordance with the DB2 Server for VSE & VM rules for assigning values to host variables.

Note: DB2 Server for VSE & VM does not support the use of structures or arrays for stored procedure parameters.

host-variable
The parameter of the CALL is the identified host variable. *Host-variable* must identify a host variable (not a structure) described in the program according to the rules for declaring host variables and the data type of the variable must be compatible with the data type of the corresponding parameter of the stored procedure. If an indicator variable is specified, its value must not be negative unless

- the parameter style for the stored procedure (as defined in SYSTEM.SYSROUTINES) is GENERAL WITH NULLS
- the parameter style for the stored procedure (as defined in SYSTEM.SYSROUTINES) is GENERAL and the corresponding parameter of the stored procedure is defined as OUT.

constant
The parameter of the CALL is the specified value. The data type of the constant must be compatible with the datatype of the corresponding parameter of the stored procedure and that parameter must be defined as IN.

NULL
The parameter of the CALL is the null value. The corresponding parameter of the stored procedure must be defined as IN and the description of the stored procedure must allow for null parameters.

USING DESCRIPTOR descriptor-name
Identifies an SQLDA that must contain a valid description of host variables (unless the stored procedure has no parameters in which case the SQLDA is not used). In C, the descriptor-name can be a pointer to an SQLDA.

Before the CALL statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA (this number must not be less than SQLD)
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA (this number must be not less than SQLN*44+16)
- SQLD to indicate the number of variables used in the SQLDA when processing the statement (this number must be the same as the number of parameters defined for the stored procedure).
- SQLVAR occurrences to indicate the attributes of the variables

Notes

1. The capability of calling stored procedures is provided to improve the performance of distributed operations, but the capability is not limited to distributed operations. Thus, the application server can be the local DB2 Server for VSE & VM.
2. The values of all parameters are passed from the application requester to the application server. To improve the performance of this operation, host variables that correspond to OUT parameters and have lengths of more than a few bytes should be set to null before the CALL statement is issued.
3. If accounting is active, the activity done and resources used by the database manager on behalf of the stored procedure will be included in the accounting records of the userid that issued the SQL CALL.

Examples

Example 1

A package for a PL/I application exists on DB_A. A package for the stored procedure REPORT1 exists on DB_B. The SYSTEM.SYSROUTINES table on DB_B describes the procedure REPORT1 which allows nulls and has two parameters. The first parameter is defined as IN and the second as OUT. Here are some of the statements in the PL/I application that runs at DB_A:

```
EXEC SQL CONNECT TO DB_B;
VAR1  = 920176;
IVAR2 = -1;
EXEC SQL
  CALL REPORT1(:VAR1, :VAR2 INDICATOR :IVAR2);
```

CLOSE

The CLOSE statement closes a cursor. In doing so, it stops the usage of the group of rows pointed to by the named cursor. Closing the cursor permits the database manager to release the resources associated with maintaining an open cursor.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required. See “DECLARE CURSOR” on page 235 for the authorization required to use a cursor.

Syntax

```
►►──CLOSE──cursor_name──────────────────────────────────────────────►◄
```

Description

cursor_name
Is an ordinary identifier that identifies the cursor to be closed. The *cursor_name* must identify a cursor defined in a DECLARE statement of your program.

When the CLOSE statement is processed, the cursor must be in the open state. When the CLOSE statement is processed, the indicated cursor leaves the open state, and its active set becomes undefined. No FETCH or PUT statement can be processed on the cursor, and no DELETE or UPDATE statement can refer to its current position, until the cursor is reopened by an OPEN statement.

Notes

Explicitly closing cursors as soon as possible can improve performance.

When a CLOSE statement is processed in a program that is blocking PUTS, the remaining rows in an incomplete block are inserted. SQLERRD(3) contains the number of rows that were successfully inserted.

Note that both the COMMIT and ROLLBACK statements automatically close all cursors (except when blocking an insert cursor - a COMMIT or ROLLBACK statement issued when there is an OPEN with a blocked insert cursor results in an error). CLOSE, however, does not cause a commit or rollback operation; these operations must be coded separately.

Examples

In a COBOL program, use the cursor C1 to fetch the values from the first four columns of the EMP_ACT table a row at a time and put them in the following host variables:
- EMP (char(6))
- PRJ (char(6))
- ACT (smallint)
- TIM (dec(5,2)).

Finally, close the cursor.

```
EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
  77 EMP              PIC X(6).
  77 PRJ              PIC X(6).
  77 ACT              PIC S9(4) COMP-4.
  77 TIM              PIC S9(3)V9(2) COMP-3.
EXEC SQL  END DECLARE SECTION  END-EXEC.
   .
   .
   .

EXEC SQL  DECLARE C1 CURSOR FOR
             SELECT EMPNO, PROJNO, ACTNO, EMPTIME
               FROM EMP_ACT                        END-EXEC.

EXEC SQL  OPEN C1  END-EXEC.

EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM  END-EXEC.

IF SQLSTATE = '02000'
  PERFORM DATA-NOT-FOUND
ELSE
  PERFORM GET-REST-OF-ACTIVITY UNTIL SQLSTATE IS NOT EQUAL TO '00000'.

EXEC SQL  CLOSE C1  END-EXEC.
   .
   .
   .

GET-REST-OF-ACTIVITY.
EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM  END-EXEC.
   .
   .
   .
```

Extended CLOSE

The Extended CLOSE statement “closes” the *cursor_name* which was opened by an Extended OPEN statement.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──CLOSE──cursor_variable──────────────────────────────────────────►◄
```

Description

cursor_variable
: Identifies the cursor that is to be closed. The cursor must have been defined by a preceding Extended DECLARE CURSOR statement in the same logical unit of work.

When the cursor is closed, its active set becomes undefined. No FETCH or PUT statement can be processed on the cursor, and no DELETE or UPDATE statement can refer to its current position, until the cursor is reopened by an Extended OPEN statement.

Notes

CLOSE permits the database manager to release the resources associated with maintaining an open cursor.

In most respects, the Extended CLOSE statement is identical to the CLOSE statement (“CLOSE” on page 175). However, in the Extended CLOSE statement, the cursor_variable is a host variable, thereby making it possible for a user to provide the cursor_variable when the program is run and to CLOSE the cursor in a logical unit of work or program other than the one in which the statement was prepared.

Examples

```
CLOSE :CURSOR1
```

COMMENT ON

The COMMENT ON statement adds or replaces comments (also called remarks) in the catalog descriptions of tables, views, or columns.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include one of the following:
- Ownership of the table or view
- DBA authority.

Syntax

```
►►──COMMENT ON──┬─┤ options_a ├──IS──str_constant──────┬──────────────────────────►◄
                └─┬─table_name─┬──(──┤ options_b ├──)──┘
                  └─view_name──┘
```

options_a

```
├──┬─TABLE──┬─table_name─┬──────────────┬─────────────────────────────────────────┤
   │        └─view_name──┘              │
   └─COLUMN──┬─table_name.column_name─┬─┘
             └─view_name.column_name──┘
```

options_b

```
   ┌─,───────────────────────────────┐
   ▼                                 │
├────column_name──IS──str_constant───┴──────────────────────────────────────────┤
```

Description

TABLE
: Indicates that the comment applies to a table or view.

table_name or *view_name*
: Identifies a table or view to which the comment applies. The name must identify a table or view that exists at the application server.

The comment is placed in the REMARKS column of the SYSTEM.SYSCATALOG catalog table for the row that describes the table or view.

COLUMN
: Indicates that the comment applies to a column.

table_name.column_name or *view_name.column_name*
: Identifies the column, qualified by the name of the table or view in which it appears. The *column_name* must identify a column of the specified table or view that exists at the application server.

The comment is placed into the REMARKS column of the SYSTEM.SYSCOLUMNS catalog table, for the row that describes the column.

Multiple comments
To comment on more than one column in the same table or view within the same statement, follow the table or view name with a list of one or more column names and string constant pairs in parentheses. The *column_name* must identify a column of the specified table or view that exists at the application server.

IS Introduces the comment that you want to make.

string_constant
Can be any SQL character string constant of up to 254 characters. The constant may contain mixed double-byte and single-byte characters.

Examples

Example 1

Insert a comment for the EMPLOYEE table into the catalog.

```
COMMENT ON TABLE EMPLOYEE
  IS 'Reflects first quarter 1981 reorganization'
```

Example 2

Insert a comment for the EMP_VIEW1 view into the catalog.

```
COMMENT ON TABLE EMP_VIEW1
  IS 'View of the EMPLOYEE table without salary information'
```

Example 3

Insert a comment for the EDLEVEL column of the EMPLOYEE table into the catalog.

```
COMMENT ON COLUMN EMPLOYEE.EDLEVEL
  IS 'highest grade level passed in school'
```

Example 4

Insert two comments into the catalog for two different columns of the EMPLOYEE table.

```
COMMENT ON EMPLOYEE
(WORKDEPT IS 'see DEPARTMENT table for names',
EDLEVEL IS 'highest grade level passed in school ')
```

COMMENT ON PROCEDURE

The COMMENT ON PROCEDURE statement adds or replaces comments to the REMARKS column of the SYSTEM.SYSROUTINES catalog table for the row that describes the stored procedure identified.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer must have DBA authority.

Syntax

```
►►──COMMENT ON PROCEDURE──procedure_name──┬─────────────────────┬──────────────────►
                                          └─AUTHID──authid──────┘

►──IS──string_constant──────────────────────────────────────────────────────────►◄
```

Description

PROCEDURE
Indicates that the comment applies to a stored procedure.

procedure_name
Identifies a stored procedure that has been defined (meaning, a CREATE PROCEDURE has been processed successfully for it).

AUTHID
Indicates that *authid* is specified.

authid
Identifies the authorization ID for the stored procedure. If specified, the comment will only be added or updated for the version of *procedure_name* that is accessible only by *authid*.

IS Introduces the comment that you want to make.

string_constant
Can be any SQL character string constant of up to 254 characters. The constant may contain mixed double-byte and single-byte characters. The comment is placed into the REMARKS column of the SYSTEM.SYSROUTINES catalog table, for the row that describes the stored procedure.

Examples

Example 1

Insert a comment for the STORPRC1 stored procedure into the catalog.

```
COMMENT ON PROCEDURE STORPRC1
  IS 'Calculates project cost for the current month in person-hours'
```

Example 2

Insert a comment for the STORPRC2 stored procedure with AUTHID USER1 into the catalog.

```
COMMENT ON PROCEDURE STORPRC2 USERID USER1
  IS 'Calculates average turn-around time for service calls for the current week'
```

COMMIT

The COMMIT statement terminates the current logical unit of work and commits the application server changes that were made by that logical unit of work.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

```
           ┌─WORK─┐
►►──COMMIT─┴──────┴─┬─────────┬──────────────────────────────────────────►◄
                    └─RELEASE─┘
```

Description

RELEASE
Specifies that when the COMMIT process is complete, your connection to the application server is severed.

For VM users, when the next SQL statement is entered, you are automatically connected with your logon user ID to the default application server. This eliminates the need to enter a CONNECT statement to return to the system default user ID after being connected to an application server as another user ID.

For VSE interactive users, when the next SQL statement is entered, you are automatically connected to the CICS default user ID on the same application server. For VSE interactive users connected to a remote DRDA application server, when the next SQL statement is entered, you are automatically connected with your CICS signon user ID to the same application server.

For VSE batch applications, an explicit CONNECT with a user ID and password is necessary after a COMMIT RELEASE to establish an SQL user ID.

In any case, if you are connected to an application server with a user ID other than the default user ID and you enter a COMMIT without specifying RELEASE, you will remain connected to the application server under that user ID.

The COMMIT statement terminates the logical unit of work in which it is processed and initiates a new logical unit of work. All changes that were made by any of the following statements during the logical unit of work are committed:

ACQUIRE DBSPACE	GRANT Package Privileges
ALTER DBSPACE	GRANT System Authorities
ALTER PROCEDURE	Alter a Stored Procedure
ALTER PSERVER	Alter a Stored Procedure Server
ALTER TABLE	GRANT Table Privileges
COMMENT ON	INSERT
CREATE INDEX	LABEL ON
CREATE PACKAGE	Extended PREPARE
CREATE PROCEDURE	Define a Stored Procedure

CREATE PSERVER	Define a Stored Procedure Server
CREATE SYNONYM	PUT
CREATE TABLE	Extended PUT
CREATE VIEW	REVOKE Package Privileges
DELETE	REVOKE System Authorities
DROP	REVOKE Table Privileges
DROP PROCEDURE	Remove a Stored Procedure
DROP PSERVER	Remove a Stored Procedure Server
DROP STATEMENT	UPDATE
EXPLAIN	UPDATE STATISTICS

All locks acquired by the logical unit of work are released. All cursors that were opened during the logical unit of work are closed. All statements that were prepared during the logical unit of work using the non-extended form of the PREPARE statement are destroyed. Any cursors associated with a prepared statement that is destroyed cannot be opened until the statement is prepared again.

Notes

If a COMMIT or ROLLBACK does not immediately precede the termination of an application process, the database manager attempts to commit the work. If there are errors during the commit process, it may not be successful. *It is strongly recommended that each application process explicitly ends its logical unit of work before terminating.*

The logical unit of work must be completed by using the COMMIT or ROLLBACK statements before the CONNECT statement can be used to switch to another user ID or application server.

TCP/IP does not perform any security checking during a physical connect. The Batch application requester will use the DRDA security handshaking flows during the logical connect to perform user ID and password verification. The physical TCP/IP connection will be deallocated and reallocated whenever the application switches to a different user ID or server name (using the CONNECT statement), and DRDA security handshaking flows will be used again during the logical connect. Either of these switches will not require the application to issue a COMMIT RELEASE or ROLLBACK RELEASE. The Batch Resource Adapter will retain and use the current user ID, password, and server name (unless different ones are specified with a new CONNECT statement) after the new TCP/IP physical connection is established. If a COMMIT RELEASE or ROLLBACK RELEASE was issued prior to a CONNECT statement, then all user ID, password and server name information is lost and must be supplied with the next CONNECT.

Examples

In a PL/I program, transfer a certain amount of commission (COMM) from one employee (EMPNO) to another in the EMPLOYEE table. Subtract the amount from one row and add it to the other. Use the COMMIT statement to ensure that no permanent changes are made to the database until both operations are completed successfully.

```
XFRCOMM:  PROC OPTIONS(MAIN);
  EXEC SQL  BEGIN DECLARE SECTION;
    DCL  AMOUNT       FIXED DECIMAL(5,2);
    DCL  FROM_EMPNO   CHAR(6);
    DCL  TO_EMPNO     CHAR(6);
  EXEC SQL  END DECLARE SECTION;
  EXEC SQL  INCLUDE SQLCA;
  EXEC SQL  WHENEVER SQLERROR GOTO SQLERR;
  EXEC SQL  CONNECT TO TOROLAB3;
  GET LIST (AMOUNT, FROM_EMPNO, TO_EMPNO);
  EXEC SQL  UPDATE EMPLOYEE
              SET COMM = COMM - :AMOUNT
              WHERE EMPNO = :FROM_EMPNO;
  EXEC SQL  UPDATE EMPLOYEE
              SET COMM = COMM + :AMOUNT
              WHERE EMPNO = :TO_EMPNO;
  EXEC SQL  COMMIT WORK;
  RETURN;
SQLERR:
  DISPLAY ('Unexpected Error -changes will be backed out');
  PUT SKIP LIST (SQLCA);
  EXEC SQL  WHENEVER SQLERROR CONTINUE;  /* continue if error on rollback */
  EXEC SQL  ROLLBACK WORK;
  RETURN;
END;  /* XFRCOMM */
```

CONNECT (for VM)

Overall Notes

The CONNECT statement connects an application process or a user, or both, to an application server.

Invocation

This statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It should be noted, however, that interactive SQL facilities, such as ISQL, provide an interface that gives the appearance of interactive execution.

Authorization

The privileges held by the authorization ID of the statement or, when specified, the *authorization_name* in the statement must include authorization to connect to the identified application server. If an *authorization_name* is specified in the statement, the appropriate password must also be specified.

Syntax

```
►►──CONNECT──┬──────────────────────────────────────────────────────────────────┬──────►
             └─┬─authorization_name─┬──IDENTIFIED BY──┬─password──────┬─┘
               └─host_variable──────┘                 └─host_variable─┘

►──┬─────────────────────────┬──────────────────────────────────────────────────►◄
   └─TO──┬─server_name───┬─┘
         └─host_variable─┘
```

Description

An application process can only be connected to one application server at a time. This is called the current server. A default application server is established when the application requester is initialized. When an application process is started, it is implicitly connected to the default application server. The application process can explicitly connect to a different application server by issuing a CONNECT statement with the TO clause. There is no default connection for CONNECT with no options. A connection lasts until one of the following occurs:

- COMMIT RELEASE or ROLLBACK RELEASE is processed
- CONNECT ... TO ... successfully switches databases
- the application terminates
- a severe error causes the connection to be severed.

authorization_name/host_variable
: Is the user ID trying to CONNECT to the application server. If used within an interactive facility, it must be a valid ordinary identifier with a maximum length of 8. If it is used in an application program, it must be a valid host variable, specified without an indicator variable, declared as a fixed-length 8-character string, and initialized before the statement is processed. (For programs written in C the host variable must be declared as a NUL-terminated string with a length of 9.) The value can be less than 8 characters; unused character positions in the host variable must be padded with blanks to the right.

IDENTIFIED BY *password/host_variable*
Is the password of the *authorization_name*. If used within an interactive facility, it must be a valid ordinary identifier with a maximum length of 8. If it is used in an application program, it must be a valid host variable, specified without an indicator variable, declared as a fixed-length 8-character string, and initialized before the statement is processed. (For programs written in C, the host variable must be declared as a NUL-terminated string with a length of 9.) The value can be less than 8 characters; unused character positions in the host variable must be padded with blanks to the right.

TO *server_name/host_variable*
Identifies the application server by the specified *server_name* or by a *host_variable* which contains the server_name. The *server_name* must be a valid ordinary identifier. Unlike authorization_name and password, if it is used in an application, it may be specified either directly or within a host variable.

If a *host_variable* is specified, it must be a character string variable with a length attribute that is not greater than 18, and an indicator variable may not be specified. (For programs written in C, if the host variable is declared as a NUL-terminated string, it must have a length attribute that is between 2 and 19.) The server_name that is contained within the *host_variable* must be left-justified and must not be delimited by quotation marks; if a fixed-length, it must be padded on the right with blanks if its length is less than that of the host variable.

The default is the currently active application server. If no application server is currently active, the default is the application server established by SQLINIT. (See the *DB2 Server for VSE & VM Database Administration* for information on SQLINIT.)

When the CONNECT statement is processed, the *server_name* must identify an application server described in the local directory (see the *DB2 Server for VM System Administration* manual) and the application process must be in the connectable state. (See “Notes” on page 187 for information about connection states.)

If the CONNECT statement is successful:

- The application process is disconnected from its previous application server, if any, and connected to the identified application server
- The name of the application server is placed in the CURRENT SERVER special register
- When using the DRDA protocol, information about the application server is placed in the SQLERRP field of the SQLCA. If the application server is an IBM product, the information has the form *pppvvrrm*, where:
  - *ppp* identifies the product as follows:
    DSN for DB2 for OS/390
    ARI for DB2 Server for VSE & VM
    QSQ for DB2 for OS/400
    SQL for DB2 for OS/2 and DB2 for AIX.
  - *vv* is a two-digit version identifier such as '02'
  - *rr* is a two-digit release identifier such as '03'
  - *m* is a one-digit modification level such as '0'.

  For example, if the application server is Version 7 Release 5 of the DB2 Server for VSE & VM, the value of SQLERRP is 'ARI06010'.

  When using the SQLDS protocol, SQLERRP is set to 'ARI   '.

For more information on the DRDA protocol and the SQLDS protocol, see "Distributed Relational Database" on page 23.

- The authorization ID and the *server_name* of the connection are placed in the SQLERRMC field of the SQLCA. The authorization ID precedes the *server_name* and these are separated by X'FF'.

If the CONNECT statement is unsuccessful because the application process is not in the connectable state or the *server_name* is not listed in the local directory, the connection state of the application process is unchanged. If the CONNECT statement is unsuccessful for any other reason, the application process remains in the connectable state.

CONNECT with No Operand

This form of the CONNECT statement returns information about the current authorization ID and application server. The information is returned in the SQLERRP and SQLERRMC fields of the SQLCA as described above. This form of CONNECT:

- Does not require the application process to be in the connectable state.
- If already connected, does not change the connection state. If unconnected, causes a connection to the default application server.

Notes

It is a good practice for the first SQL statement processed by an application process to be the CONNECT statement.

Summary of Variations of the CONNECT Statement

The various clauses may be specified in the following combinations:

1. **`CONNECT`**

   This returns information about the currently connected authorization ID and application server.
2. **`CONNECT`** `authorization_name` **`IDENTIFIED BY`** `password`

   This switches to a new authorization ID on the currently established application server.
3. **`CONNECT TO`** *`server_name`*

   This switches the currently established authorization ID to a new application server.
4. **`CONNECT`** `authorization_name` **`IDENTIFIED BY`** `password` **`TO`** *`server_name`*

   This switches to both a new authorization ID and application server.

Only variations 1 and 2 are available in single user mode.

Table 8. CONNECT Variations Supported by Communication Protocols

Variation	SQLDS Protocol	DRDA Protocol	Single User Mode	Multiple User Mode
Variation 1	Yes	Yes	Yes	Yes
Variation 2	Yes	Yes	Yes	Yes
Variation 3	Yes	Yes	No	Yes
Variation 4	Yes	Yes	No	Yes

Connection States: An application process is in one of four states at any time:

- Connectable and connected

- Unconnectable and connected
- Connectable and unconnected
- Implicitly connectable.

An application process is initially in the implicitly connectable state.

The connectable and connected state: An application process is connected to an application server and CONNECT statements can be processed. The process enters this state when it completes a rollback or successful commit from the unconnectable and connected state, or a CONNECT statement is successfully processed from the connectable and unconnected state.

The unconnectable and connected state: An application process is connected to an application server, but a CONNECT statement cannot be successfully processed to change application servers or to change authorization IDs. The process enters this state from the connectable and connected state when it processes any SQL statement other than CONNECT, COMMIT or ROLLBACK.

The connectable and unconnected state: An application process is not connected to an application server. The only SQL statement that can be processed is CONNECT. The process enters this state when an SQL statement is unsuccessful because of a failure that causes a rollback operation at the application server and the loss of the connection. The process can also enter this state if it processes a CONNECT statement unsuccessfully.

The implicitly connectable state: An application process is not connected to an application server and CONNECT statements can be processed. The process enters this state when it completes a rollback or successful commit with the release option from the unconnectable and connected state.

The following diagram shows the state transitions:

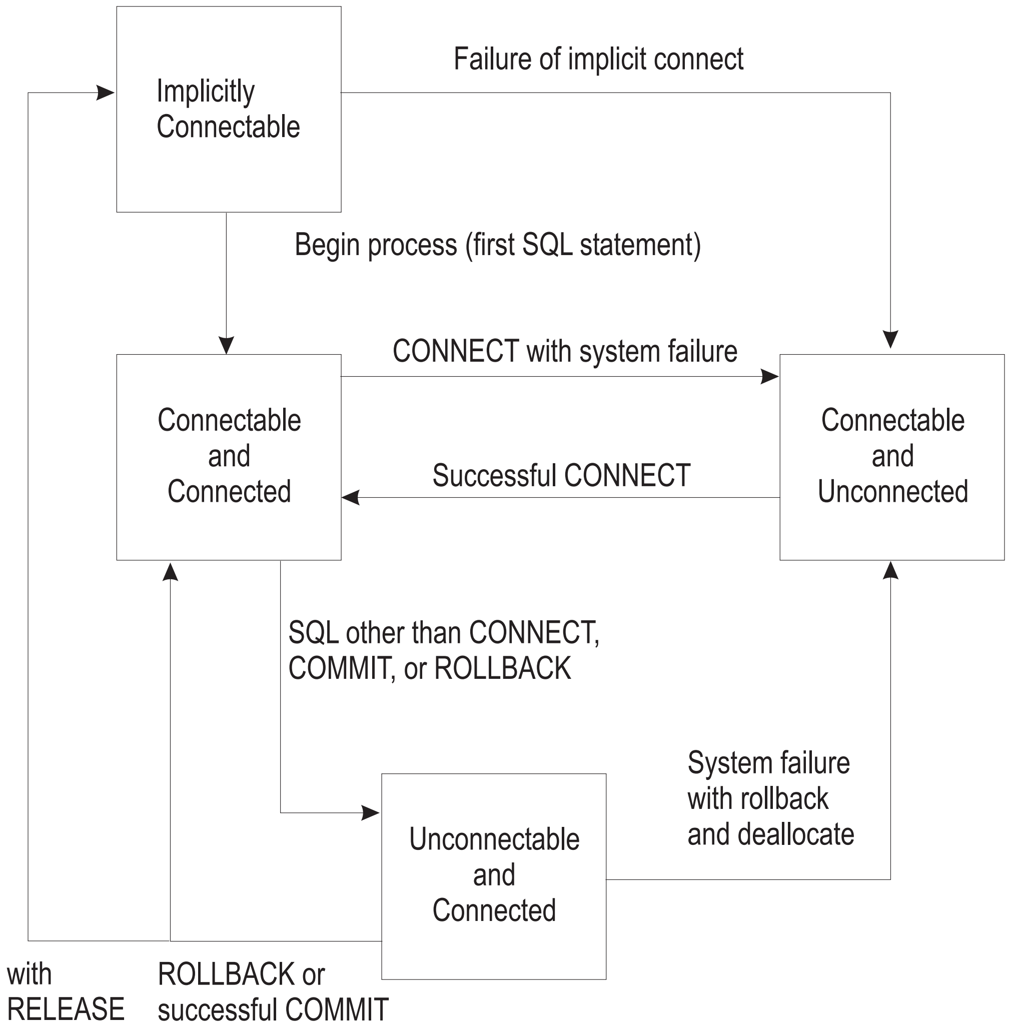

Implicitly Connectable
Failure of implicit connect
Begin process (first SQL statement)
CONNECT with system failure
Connectable and Connected
Successful CONNECT
Connectable and Unconnected
SQL other than CONNECT, COMMIT, or ROLLBACK
System failure with rollback and deallocate
Unconnectable and Connected
with RELEASE
ROLLBACK or successful COMMIT

Figure 7. VM Connection State Transitions

Additional Rules: It is not an error to process consecutive CONNECT statements because CONNECT itself does not remove the application process from the connectable state. It is an error to process any SQL statement other than CONNECT, COMMIT, or ROLLBACK, and then process CONNECT with any options. To avoid the error, process a commit or rollback operation before processing the CONNECT.

A CONNECT to the current application server is treated like any other CONNECT. Such a CONNECT can cause the redundant deallocation and allocation of a conversation.

Notes

A VM user ID may be transformed when using DRDA protocol. See the *DB2 Server for VM System Administration* manual for more information on the CMS communications directory which may cause this transformation.

The old connection will not be disconnected until the new connection is made successfully. Two connections are therefore held for a short interval. If there are many applications running concurrently that switch application servers, this may cause a wait for sessions. If experiencing delays, use COMMIT RELEASE which will disconnect explicitly.

Examples

Example 1

In a PL/I program, connect to the application server TOROLAB3.

```
EXEC SQL  CONNECT TO TOROLAB3;
```

Example 2

In a PL/I program, switch to a different application server called TOROLAB4. Assume your user ID on TOROLAB4 is different than the one you are currently using.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  USERID       CHAR(8);
  DCL  PASWRD       CHAR(8);
EXEC SQL  END DECLARE SECTION;


EXEC SQL  CONNECT :USERID IDENTIFIED BY :PASWRD
            TO TOROLAB4;
```

Example 3

In a PL/I program, connect to an application server whose name is stored in the host variable APP_SERVER (varchar(18)). Following a successful connection, copy the 3 character product identifier of the application server to the host variable PRODUCT (char(3)).

```
EXEC SQL  CONNECT TO :APP_SERVER;
IF SQLSTATE = '00000' THEN
PRODUCT = SUBSTR(SQLERRP,1,3);
```

CONNECT (for VSE)

Overall Notes

The CONNECT statement connects an application process or a user, or both, to an application server.

Invocation

This statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared. It should be noted, however, that interactive SQL facilities, such as ISQL, provide an interface that gives the appearance of interactive execution.

Authorization

The privileges held by the authorization ID of the statement or, when specified, the *authorization_name* in the statement must include authorization to connect to the identified application server. If an *authorization_name* is specified in the statement, the appropriate password must also be specified.

Syntax

```
►►──CONNECT──┬──────────────────────────────────────────────────────────────┬──────────►
             │                    (1)                                   (1) │
             └─┬─authorization_name─┬──IDENTIFIED BY──┬─password──────┬──────┘
               └─host_variable──────┘                 └─host_variable─┘

►──┬───────────────────────┬──────────────────────────────────────────────────────────►◄
   └─TO──┬─server_name───┬─┘
         └─host_variable─┘
```

Notes:

1 An implicit connect is not allowed by a Batch application requester. Therefore, the user ID and password must be supplied on the CONNECT statement used for Batch application requester processing.

Description

An application process can only be connected to one application server at a time. This is called the current server. A default application server is established when the application requester is initialized. When an application process is started and a CONNECT statement is issued, the application is connected to the default application server. The application process can explicitly connect to a different application server by issuing a CONNECT statement with the TO clause. There is no default connection for CONNECT with no options. A connection lasts until one of the following occurs:

- COMMIT RELEASE or ROLLBACK RELEASE is processed
- CONNECT ... TO ... successfully switches databases
- the application terminates
- a severe error causes the connection to be severed.

authorization_name/host_variable
Is the user ID trying to CONNECT to the application server. If used within an interactive facility, it must be a valid ordinary identifier with a maximum length of 8. If it is used in an application program, it must be a valid host

variable, specified without an indicator variable, declared as a fixed-length 8-character string, and initialized before the statement is processed. (For programs written in C the host variable must be declared as a NUL-terminated string with a length of 9.) The value can be less than 8 characters; unused character positions in the host variable must be padded with blanks to the right.

IDENTIFIED BY *password/host_variable*
Is the password of the *authorization_name*. If used within an interactive facility, it must be a valid ordinary identifier with a maximum length of 8. If it is used in an application program, it must be a valid host variable, specified without an indicator variable, declared as a fixed-length 8-character string, and initialized before the statement is processed. (For programs written in C, the host variable must be declared as a NUL-terminated string with a length of 9.) The value can be less than 8 characters; unused character positions in the host variable must be padded with blanks to the right.

Note: An implicit connect is not allowed by a Batch application requester. Therefore, the user ID and password must be supplied on the CONNECT statement used for Batch application requester processing.

TO *server_name/host_variable*
Identifies the application server by the specified *server_name* or by a *host_variable* which contains the server_name. The *server_name* must be a valid ordinary identifier. This option may be used only in an application and, unlike authorization_name and password, it may be specified either directly or within a host variable.

If a *host_variable* is specified, it must be a character string variable with a length attribute that is not greater than 18, and an indicator variable may not be specified. (For programs written in C, if the host variable is declared as a NUL-terminated string, it must have a length attribute that is between 2 and 19.) The server_name that is contained within the *host_variable* must be left-justified and must not be delimited by quotation marks; if a fixed-length, it must be padded on the right with blanks if its length is less than that of the host variable.

The default is the application server as defined in the DBNAME directory. If a batch application attempts the connect, then the *server_name* must be one that exists in the DBNAME directory. If it is a remote server, it must be identified as using TCP/IP communication. Otherwise, an SQL error will be returned to the batch application. (See the *DB2 Server for VSE System Administration* manual for information on the DBNAME directory.)

When the CONNECT statement is processed, the *server_name* must identify an application server described in the DBNAME directory (see the *DB2 Server for VSE System Administration* manual) and the application process must be in the connectable state. (See "Notes" on page 193 for information about connection states.)

If the CONNECT statement is successful:

- The application process is disconnected from its previous application server, if any, and connected to the identified application server
- The name of the application server is placed in the CURRENT SERVER special register.

If you are connected using SQLDS protocol, the SQLERRP field in the SQLCA is set to 'ARI   '. If you are connected using DRDA protocol, the format of SQLERRP will be *pppvvrrm*, where:

– *ppp* identifies the product as follows:
  DSN for DB2 for OS/390
  ARI for DB2 Server for VSE & VM
  QSQ for DB2 for OS/400
  SQL for DB2 for OS/2 and DB2 for AIX.
– *vv* is a two-digit version identifier such as '02'
– *rr* is a two-digit release identifier such as '03'
– *m* is a one-digit modification level such as '0'.

- The authorization ID and the *server_name* of the connection are placed in the SQLERRMC field of the SQLCA. The authorization ID precedes the *server_name* and these are separated by X'FF'.

If the CONNECT statement is unsuccessful because the application process is not in the connectable state or the *server_name* is not listed in the DBNAME directory, the connection state of the application process is unchanged. If the CONNECT statement is unsuccessful for any other reason, the application process remains in the connectable state.

CONNECT with No Operand

This form of the CONNECT statement returns information about the current authorization ID and application server. The information is returned in the SQLERRP and SQLERRMC fields of the SQLCA as described above. This form of CONNECT:
- Does not require the application process to be in the connectable state.
- Does not change the connection state.

Notes

In a batch program, either
- **CONNECT**
- **CONNECT** *userid* **IDENTIFIED BY** pw
- **CONNECT** *userid* **IDENTIFIED BY** pw **TO** *server_name*
- **CONNECT TO** *server_name*

must be the first SQL statement processed by the program. If a **CONNECT TO** *server_name* statement is processed first, it must be followed by one of the other three CONNECT statements above.

If a CONNECT with no options is processed first, the SQLERRMT fields will be set to a blank user ID and blank server name. In this case, there is no default application server. If the new target server is remote, then a new DRDA connection to that remote server will be allocated and DRDA security handshaking will be performed. If the new target server is local, DRDA flows are not possible and an XPCC connection will be used. A CONNECT statement with no parameters specified returns current connection information in the SQLERRP field of SQLCA.

If a DRDA connection exists when a CONNECT with no options is specified, the current connection information is returned in the SQLERRP field of the SQLCA.

One of the remaining forms from the list above is required to establish the proper identification of the user on the application server.

If a **CONNECT TO** *server_name* is processed first, the server name is placed in the CURRENT SERVER register. Also, the SQLERRMC field in the SQLCA is set with eight blanks and the *server_name* separated by X'FF'. However, a **CONNECT** `authorization_name` **IDENTIFIED BY** `password` must be processed to complete the connection and establish the user identification before any other SQL statements are processed.

If the TO clause is not specified, the application is connected to the default application server. The *server_name* must be one that exists in the DBNAME directory. If it is a remote server, it must be identitified as using TCP/IP communication. Otherwise, an SQL error will be returned to the batch application.

TCP/IP does not perform any security checking during a physical connect. The Batch application requester will use the DRDA security handshaking flows during the logical connect to perform user ID and password verification. The physical TCP/IP connection will be deallocated and reallocated whenever the application switches to a different user ID or server name (using the CONNECT statement), and DRDA security handshaking flows will be used again during the logical connect. Either of these switches will not require the application to issue a COMMIT RELEASE or ROLLBACK RELEASE. The Batch Resource Adapter will retain and use the current user ID, password, and server name (unless different ones are specified with a new CONNECT statement) after the new TCP/IP physical connection is established. If a COMMIT RELEASE or ROLLBACK RELEASE was issued prior to a CONNECT statement, then all user ID, password and server name information is lost and must be supplied with the next CONNECT.

If a Logical Unit of Work is ended by a **COMMIT**, and a **CONNECT TO** *server_name* is the next SQL statement processed, a new connection is made to the application server specified, with the user ID and password being the same as in the previous connection.

If a Logical Unit of Work is ended with a **COMMIT RELEASE**, the next SQL statement must be either:
- **CONNECT** *userid* **IDENTIFIED BY** `pw` *or*
- **CONNECT** *userid* **IDENTIFIED BY** `pw` **TO** *server_name*

to re-establish the proper user ID.

Summary of Variations of the CONNECT Statement

The various clauses may be specified in the following combinations:

1. **CONNECT**

   This returns information about the currently connected authorization ID and application server.
2. **CONNECT** `authorization_name` **IDENTIFIED BY** `password`

   This switches to a new authorization ID on the currently established application server.
3. **CONNECT TO** *server_name*

   This switches the currently established authorization ID to a new application server.
4. **CONNECT** `authorization_name` **IDENTIFIED BY** `password` **TO** *server_name*

   This switches to both a new authorization ID and application server.

Connection States

An application process is in one of three states at any time:
- Connectable and connected
- Unconnectable and connected
- Connectable and unconnected.

The connectable and connected state: An application process is connected to an application server and CONNECT statements can be processed. The process enters this state when it completes a rollback or successful commit from the unconnectable and connected state, or a CONNECT statement is successfully processed from the connectable and unconnected state.

The unconnectable and connected state: An application process is connected to an application server, but a CONNECT statement cannot be successfully processed to change application servers or to change authorization IDs. The process enters this state from the connectable and connected state when it processes any SQL statement other than CONNECT, COMMIT or ROLLBACK.

The connectable and unconnected state: An application process is not connected to an application server. The only SQL statement that can be processed is CONNECT. The process is initially in this state or enters this state when an SQL statement is unsuccessful because of a failure that causes a rollback operation at the application server and the loss of the connection. The process can also enter this state if it successfully completes a commit or rollback with the release option from the unconnectable and connected state or it processes a CONNECT statement unsuccessfully.

The following diagram shows the state transitions:

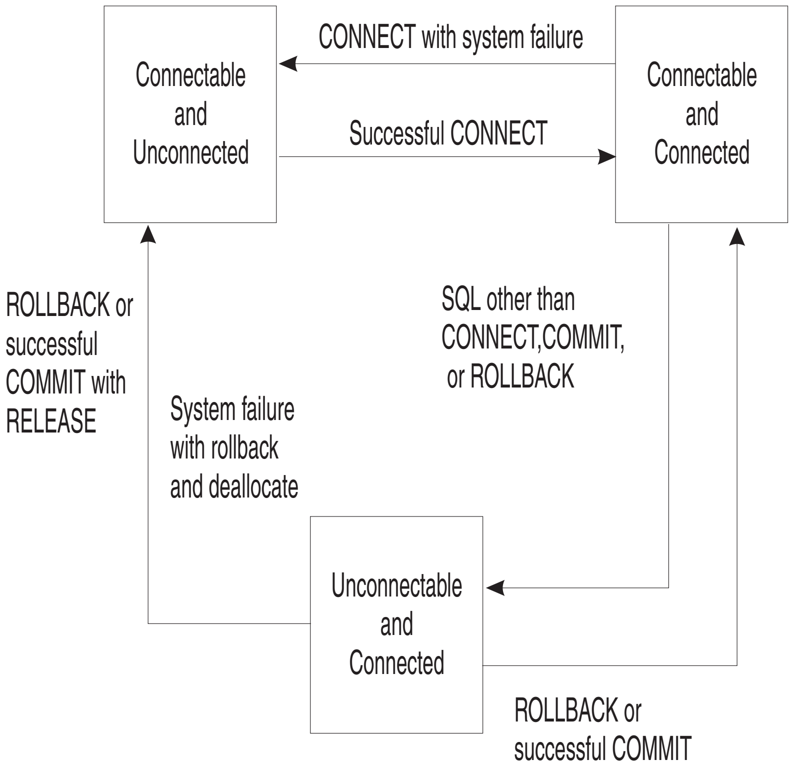

Connectable and Unconnected
CONNECT with system failure
Successful CONNECT
Connectable and Connected
ROLLBACK or successful COMMIT with RELEASE
System failure with rollback and deallocate
SQL other than CONNECT,COMMIT, or ROLLBACK
Unconnectable and Connected
ROLLBACK or successful COMMIT

Figure 8. VSE Connection State Transitions

Additional Rules

It is not an error to process consecutive CONNECT statements because CONNECT itself does not remove the application process from the connectable state. It is an error to process any SQL statement other than CONNECT, COMMIT, or ROLLBACK, and then process CONNECT with any options. To avoid the error, process a commit or rollback operation before processing the CONNECT.

Notes

If a program is connectable and connected, a **CONNECT TO** *server_name* results in the old connection being disconnected before the new connection is attempted. If the new connection fails, the program's state is connectable and unconnected.

A CONNECT to the same application server without changing the authorization ID is treated as a no-operation; the connection is not disconnected and reconnected.

Examples

Example 1

In a PL/I program, connect to the application server TOROLAB3.

```
EXEC SQL  CONNECT TO TOROLAB3;
```

Example 2

In a PL/I program, switch to a different application server called TOROLAB4. Assume your user ID on TOROLAB4 is different than the one you are currently using.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  USERID       CHAR(8);
  DCL  PASWRD       CHAR(8);
EXEC SQL  END DECLARE SECTION;


EXEC SQL  CONNECT :USERID IDENTIFIED BY :PASWRD
            TO TOROLAB4;
```

Example 3

In a PL/I program, connect to an application server whose name is stored in the host variable APP_SERVER (varchar(18)).

```
EXEC SQL  CONNECT TO :APP_SERVER;
```

Resolving Remote Server Name to Target Database

CICS Applications: If the CICS/VSE transaction issues an SQL CONNECT statement with the ″TO server name″ clause, the server name is established explicitly for the transaction and the Online Resource Adapter uses the DBNAME Directory to resolve the server name to the target database.

If the CICS/VSE transaction did not issue an SQL CONNECT statement with the“TO *server name*” clause, the Online Resource Adapter attempts to connect to the default application server, as defined in the DBNAME Directory.

If the target database is a Remote server and the communications protocol to be used is SNA, the application requester issues a GDS ALLOCATE command to acquire a session for the remote system where the server runs. The SYSID used in this ALLOCATE command is the SYSID value from the DBNAME Directory entry (and the SYSID must match a CEDA DEF CONNECTION definition). Then the application requester issues a GDS CONNECT PROCESS command to initiate an APPC basic conversation with the Remote server. The PROCNAME used by this CONNECT PROCESS command is the REMTPN value from the DBNAME Directory entry.

If the target database is a Remote server and the communications protocol to be used is TCP/IP, the application requester issues a CONNECT to the TCP/IP listener port number that is specified by the TCPPORT value from the DBNAME Directory entry. The target database is identified by the IPADDR or TCPHOST values from the DBNAME Directory entry.

If the target database is a Local or Host VM (guest sharing) server, normal communications occurs using XPCC.

The default application server is determined when the CIRB transaction is invoked and can be changed subsequently by a CIRC transaction. For more information on establishing a default application server, see *DB2 Server for VSE & VM Database Administration*.

Batch Applications: Batch applications access the Remote server in the same way as CICS Transactions, but SNA communications protocol is not supported, only TCP/IP. In addition, the Batch application must issue an SQL CONNECT statement as the first SQL statement because an implicit connect is not allowed for Batch applications.

Communications Protocols for Remote Server Access: The communications method used to access a Remote server by CICS applications is specified by the Communications Protocol setting in the SQLGLOB file, which can be either SNA or TCP/IP. The remote server to be accessed must be connected by the desired protocol. The default protocol in the SQLGLOB Default User entry is SNA, but this can be changed. The protocol option can be specified for each user ID in the SQLGLOB file. For more information about the SQLGLOB file, see *DB2 Server for VSE & VM Database Administration*.

The communications method used to access a remote server by Batch applications can only be TCP/IP; SNA is not supported for Batch applications.

If a server is identified in the DBNAME Directory as a Remote server, it must contain information that identifies which communications protocols can be used to access the Remote server. Either SNA or TCP/IP information (or both) can be specified in the DBNAME Directory. For more information about the DBNAME Directory, see *DB2 Server for VSE & VM Database Administration*.

CREATE INDEX

The CREATE INDEX statement creates an index on a table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table
- The INDEX privilege for the table
- DBA authority.

If the index name includes a qualifier that is not the same as the authorization ID of the statement, the privileges held by the authorization ID of the statement must include DBA authority.

Syntax

```
►►──CREATE──┬────────┬──INDEX──index_name──────────────────────────────────►◄
            └─UNIQUE─┘


                        ┌─,────────────────────────────┐
                        │                 ┌─ASC──┐     │
►►──ON──table_name──(───▼──column_name────┼──────┼─────┴──)──────────────────►◄
                                          └─DESC─┘


   ┌─PCTFREE = 10──────┐
►►─┼───────────────────┼───────────────────────────────────────────────────►◄
   └─PCTFREE = integer─┘
```

Description

UNIQUE
Prevents the table from containing two or more rows with the same value of the index key. The constraint is enforced when rows of the table are updated or new rows are inserted.

The constraint is also checked during the execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created.

When UNIQUE is used, null values are treated as any other values. For example, if the key is a single column that can contain null values, that column can contain no more than one null value. Unique indexes will not allow values which differ only by the number of trailing blanks.

INDEX *index_name*
Provides a name for the index. The name, including the implicit or explicit qualifier, must not identify an index that already exists at the application server.

If the index name is qualified, the qualifier is the *owner* of the index. Otherwise, the authorization ID of the statement is the owner of the index. The owner has the privilege of dropping the index.

ON *table_name*
Identifies the table on which you want the index to be created. The *table_name* must be the name of a base table (not a view) that exists at the application server. The indicated table may be empty.

(*column_name***,...)**
Identifies a column that is to be part of the index key.

Each *column_name* must be an unqualified name that identifies a column of the table. Up to 16 unique columns may be specified. Indexes cannot be created for views or for columns containing long strings.

ASC
Puts the index entries in ascending order by the column. This is the default.

DESC
Puts the index entries in descending order by the column.

PCTFREE
Controls the amount of free space reserved in an index for later insertions and updates. PCTFREE defines the percentage (integer) of the total space of the index that is to be reserved for this purpose. PCTFREE may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take more space in the database, but reduces the time required to insert or update rows in the indexed table. If you do not include a PCTFREE clause on the CREATE INDEX statement, the database manager sets PCTFREE to 10.

Notes

If the named table already contains data, CREATE INDEX creates the index entries for it. If the table does not yet contain data, CREATE INDEX creates a description of the index; the index entries are created when data is inserted into the table.

The sum of the length attributes of the indexed columns, plus approximately 25% of the length attributes of any indexed columns of varying-length character type, must not exceed 255 bytes. If you are creating the index after data has been loaded into the table, a sort is invoked during the preprocessing of the CREATE INDEX command. If duplicate keys are allowed in the index, then the sort will require 4 bytes to be added to the encoded key. These four bytes are part of the 255 total bytes.

At preprocessing time, the database manager optimizer chooses which index, if any, is to be used in processing a given query or data manipulation statement. The index provides a fast means to access the table directly by the indexed columns. However, there is a slight increase in the time required to update the indexed columns because the database manager must also update the index. It is good practice to create indexes *before preprocessing* programs that might take advantage of them. When you create a new index, existing packages are not marked incorrect because they can still use their original access path. However, an existing program may run more efficiently by taking advantage of the new index. If this is the case, you should preprocess the program again. A new package is then created for the program, possibly using the new index.

An index is maintained by the database manager until it is explicitly dropped using a DROP INDEX statement, or until its table or dbspace is dropped.

Recovery of a CREATE INDEX statement could result in the index being marked as invalid. The database manager will end if an attempt is made to mark an index as not valid if the system limit of not valid indexes has been reached. The database manager will not allow a CREATE INDEX statement to proceed if the number of currently not valid indexes plus the number of potentially not valid indexes (currently executing CREATE INDEX statements plus DBSU REORGANIZE INDEX commands) has reached the limit.

If doing many updates to an indexed column or inserting many rows into an indexed table (as the Database Services utility does), it is often best to drop the index before doing the updates and then re-create it after the updates are complete. Because the index is not being updated while the table is being updated, this can be a significant performance improvement.

For information on how to calculate the length of an encoded key refer to the *DB2 Server for VSE & VM Database Administration* manual.

Examples

Example 1

Create an index named UNIQUE_NAM on the PROJECT table. The purpose of the index is to ensure that there are not two entries in the table with the same value for project name (PROJNAME). The index entries are to be in ascending order.

```
CREATE UNIQUE INDEX UNIQUE_NAM
  ON PROJECT(PROJNAME)
```

See example 4 in ALTER TABLE for an alternate method of ensuring unique project names.

Example 2: Create an index named JOB_BY_DEPT on the EMPLOYEE table. Arrange the index entries in ascending order by job title (JOB) within each department (WORKDEPT). Leave 33 percent of the space in the index free for later insertions.

```
CREATE INDEX JOB_BY_DEPT
  ON EMPLOYEE (WORKDEPT, JOB)
  PCTFREE = 33
```

CREATE PACKAGE

The CREATE PACKAGE statement creates a package.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

None required. However, a DBA authority is required to create a package that is to be owned by someone else.

Syntax

```
                    (1)
▶▶──CREATE PACKAGE────────package_spec──┬─────────────────────────────────────────────────┬──────▶◀
                                        │                      ┌──────────────────┐        │
                                        │                      ▼   (2)            │        │
                                        └──USING OPTIONs──┬──────────────option───┴──┬─────┘
                                                          └──host_variable───────────┘
```

Notes:

1 PROGRAM is equivalent to PACKAGE, and is provided for compatibility with some older versions of the SQL/DS product.

2 An option may be specified only once.

Description

package_spec
Provides a name for the package.

If the *package_spec* is identical to the name of an existing package and the REPLACE option is specified, the existing package is implicitly dropped and replaced with a new package.

USING OPTIONs *option*
USING OPTIONs *host_variable*
The options are as follows:

CCSIDSbcs (*integer***)**
This option specifies the default CCSID to be used if a character column of subtype SBCS is defined by a CREATE or ALTER TABLE statement in this package without an explicit CCSID being specified for the column. If this option is not specified, the target application server will use its system default.

This option can only be used when connected to a DB2 Server for VM or DB2 Server for VSE application server.

CCSIDMixed (*integer***)**
This option specifies the default CCSID to be used if a character column of subtype mixed is defined by a CREATE or ALTER TABLE statement in this package without an explicit CCSID being specified for the column. If this option is not specified, the target application server will use its system default.

This option can only be used when connected to a DB2 Server for VM or DB2 Server for VSE application server.

CCSIDGraphic (*integer*)
This option specifies the default CCSID to be used if a graphic column is defined by a CREATE or ALTER TABLE statement in this package without an explicit CCSID being specified for the column. If this option is not specified, the target application server will use its system default.

This option can only be used when connected to a DB2 Server for VM or DB2 Server for VSE application server.

CHARSUB Sbcs
CHARSUB Mixed
CHARSUB Bit
This option specifies the default character subtype to be used if a character column is defined by a CREATE or ALTER TABLE statement in this package without an explicit subtype or CCSID being specified. If this option is not specified, the target application server will use its system default.

This option can only be used when connected to a DB2 Server for VM or DB2 Server for VSE application server.

DATE ISO
DATE USA
DATE EUR
DATE JIS
DATE LOCAL
This option specifies which output date format will be used by the SQL statements. If the DATE option is not specified, the format specified at installation time is used. If LOCAL is implicitly or explicitly specified, a DATE installation exit must be installed.

If using DRDA protocol, ISO is the default format.

The DATE LOCAL option is not supported for non-modifiable packages created by using extended dynamic statements with DRDA protocol. If specified, an error will occur indicating an incorrect parameter.

EXPLAIN NO
EXPLAIN YES
This option, if set to YES, specifies whether explanatory information for all explainable SQL statements in a package should be produced. NO is the default.

ISOLation RR
ISOLation CS
ISOLation USER
ISOLation RS
ISOLation UR
This option specifies the isolation level for the package. The DB2 Server for VSE & VM database manager supports RR, CS, UR, and USER. For a description of isolation levels, see “Isolation Level” on page 20. For information on USER, see the section on preprocessing and running a program in the *DB2 Server for VSE & VM Application Programming* manual. RR is the default.

In a VM environment, RS is not directly supported by the application server. In a VSE environment, RS is not supported at all. In both VM and VSE, isolation level RS is upgraded to level RR. (See the *IBM SQL Reference* manual for details on RS.)

The ISOLATION USER option is not supported for non-modifiable packages created by using extended dynamic statements with DRDA protocol. If specified, USER will be overridden with CS.

KEEP
REVOKE
This option applies if the package has previously been created and the owner of the package has granted the EXECUTE privilege on the resulting package to other users.

KEEP causes these grants of the EXECUTE privilege to remain in effect when the new package is created. KEEP is the default.

If the REVOKE option is specified, or if the owner of the package is not entitled to grant all privileges embodied in the program, the preprocessor revokes all existing grants of the EXECUTE privilege.

LABEL (*label_text*)
This option specifies a label for the package. *Label_text* can be, at most, 30 characters in length. If specified, *label_text* is stored in column PLABEL in the SYSTEM.SYSACCESS catalog table; the default is 30 spaces.

NOBLocK
BLocK
SBLocK
This option specifies if rows should be inserted and retrieved in groups.

If the BLocK option is specified, all eligible query cursors return results in groups of rows. All eligible insert cursors process inserts in groups of rows.

| If NOBLocK is specified, rows are not grouped.

| SBLocK is primarily for use with application servers that support the FOR
| FETCH ONLY clause on the DECLARE CURSOR statement. When SBLock
| is specified, all eligible cursors return results in group of rows. This is the
| default.

NOCHECK
CHECK
ERROR
This option specifies what action to take when an SQL statement is prepared into the package and checked for validity. For all options, if a statement fails its validity check, an appropriate SQLCODE and SQLSTATE is returned in the SQLCA.

- If NOCHECK is specified and any SQL statement fails its validity check, the package will not be created. This is the default.
- If CHECK is specified, the package is not created, even if all SQL statements pass their validity check.
- If ERROR is specified, the package is created even if any SQL statement fails its validity check. The subsequent execution of a not valid statement results in a -525 SQLCODE, SQLSTATE 51015. Note that for a modifiable package, ERROR and EXIST may not be specified together.

NODESCRIBE

DESCRIBE
This option allows the use of the Extended DESCRIBE for statements added to the created package.

If DESCRIBE is specified, Extended DESCRIBE statements can be processed.

If NODESCRIBE is specified, these Extended DESCRIBE statements cannot be processed. This is the default. NODESCRIBE is not supported with DRDA protocol and will be changed to DESCRIBE.

NOEXIST
EXIST
This option specifies the action to be taken when objects referenced in a program are checked for existence and their access authorizations are checked.

If NOEXIST is specified, a warning is returned to the program if object and authorization existence is not found. This will not affect the creation of the package (for instance, if NOCHECK is in effect and everything else is valid, then the package will be created). NOEXIST is the default.

If EXIST is specified, an error is returned to the program if an object does not exist or if the authorization ID of an Extended PREPARE statement does not have the appropriate privileges on an object. In such a case, the package is not created, even if ERROR is specified. For modifiable packages, ERROR and EXIST may not be specified together.

NOMODIFY
MODIFY
This option specifies whether the created package can be modified after it is stored through a COMMIT. Sections are added to the package by using the Extended PREPARE and deleted by using the DROP STATEMENT function.

Sections in packages created with the MODIFY option can also be processed or dropped before committing the logical unit of work in which they were prepared.

The MODIFY option should not be used if the entire package will be replaced using the REPLACE option. Once a package has been created with the MODIFY option specified, it can be changed but not replaced by subsequent CREATE PACKAGE statements. To replace a package created with the MODIFY option, it is necessary to enter a DROP PACKAGE statement and then enter a CREATE PACKAGE.

NOMODIFY is supported with DRDA protocol; however, there are some restrictions (see Appendix G, "DRDA Considerations," on page 425). MODIFY is not supported with DRDA protocol and will be changed to NOMODIFY. NOMODIFY is the default.

OWner (*authorization_name***)**
This option specifies the owner of the package being created. If this option is not specified, the binder's authorization ID at the application server is used.

For DB2 Server for VSE & VM application servers, the *authorization_name* must be the same as the binder's authorization ID at the application server.

QUALifier (*collection_id***)**
This option specifies the default *collection_id* to be used within the package

to resolve unqualified object names. If this option is not specified, the binder's authorization ID at the application server is used.

For DB2 Server for VSE & VM application servers, the *collection-id* must be the same as the binder's authorization ID at the application server.

RELease COMMIT
RELease DEALLOCATE

This option specifies when the application server should release the package execution resources and any associated locks.

If COMMIT is specified, the resources are released when a logical unit of work (LUW) is committed or rolled back. This is the default.

If DEALLOCATE is specified, the resources are released when the application process terminates.

For DB2 Server for VSE & VM application servers, the only acceptable option is RELEASE(COMMIT).

REPLACE
NEW

This option specifies whether the package being created is new or whether it will replace an existing package that has the same name. REPLACE is the default.

If NEW is specified, an error results if a package already exists with the same name.

If REPLACE is specified and no previous package exists with the same name, no error or warning is given. If NEW is specified along with KEEP or REVOKE, an error results.

TIME ISO
TIME USA
TIME EUR
TIME JIS
TIME LOCAL

This option specifies which output time format will be used by the SQL statements. If the TIME option is not specified, the format specified at installation time is used. If LOCAL is implicitly or explicitly specified, a TIME installation exit must be installed.

If using DRDA protocol, ISO is the default format.

The TIME LOCAL option is not supported for non-modifiable packages created by using extended dynamic statements with DRDA protocol. If specified, an error will occur indicating an incorrect parameter.

host_variable
Contains a list of options, delimited by a comma or blank. This host variable must be declared as VARCHAR and has a maximum length of 8192.

Notes

The package is stored in the database when a COMMIT is issued.

When the logical unit of work, in which the CREATE PACKAGE statement is entered, is committed (using COMMIT), a new package is created. ROLLBACK prevents the storage of the new package. A package created with the MODIFY option can be committed even if it contains no statements. Only one package may be created or modified within a logical unit of work.

Before SQL/DS Version 3 Release 1, the values for the ISOLATION, DATE, and TIME bind options were derived from the corresponding options with which the application was preprocessed. With SQL/DS Version 3 Release 1, these options became pure bind options, meaning that their values are to be based only on their specification in the CREATE PACKAGE statement. This change will only take effect after the application issuing the CREATE PACKAGE statement has been repreprocessed, reassembled, and relinked.

Note: For DB2 Server for VSE, if a combination of the NOBIND, BIND, or the PACKAGE, NOPACKAGE or the CHECK, NOCHECK and ERROR was specified, the preprocessor will generate an error message. For example, if PACKAGE, NOPACKAGE, NOBIND, BIND were all specified, the preprocessor will display the following error messages:

```
ARI0583E - Keywords PACKAGE and NOPACKAGE were both found.
         - Specify only one.
ARI0583E - Keywords NOBIND and BIND were both found.
         - Specify only one.
ARI0586I - Preprocessing ended with 2 errors and
         - 0 warnings.
```

For DB2 Server for VSE, if NOBIND, NOCHECK and NOPACKAGE are all specified, no action would be taken for this preprocessing. This is considered an error and the following error messages will be displayed:

```
ARI5411E - Keywords NOBIND, NOCHECK and NOPACKAGE are
         - specified. No preprocess will be done for this
         - operation.
ARI0586I - Preprocessing ended with 1 errors and
         - 0 warnings.
```

The restriction for non-modifiable packages created by using extended dynamic statements with DRDA protocol are as follows:

- Only one CREATE PACKAGE statement is permitted in a logical unit of work.
- After a CREATE PACKAGE with the NOMODIFY option has been issued, only Extended PREPARE SQL statements can follow in the same logical unit of work. The following statements are the only ones valid in a package created using CREATE PACKAGE:
  - Extended DECLARE CURSOR
  - Extended DROP STATEMENT
  - Extended OPEN, FETCH, PUT, and CLOSE
  - Extended EXECUTE
  - Extended DESCRIBE
- After the unit of work containing the CREATE PACKAGE has completed, an Extended PREPARE or DROP STATEMENT cannot be used to change the package that was created with the CREATE PACKAGE statement.

Examples

```
CREATE PACKAGE JERRY.MUSICIANS USING OPTIONS DESCRIBE NEW BLOCK
```

CREATE PROCEDURE

The CREATE PROCEDURE statement inserts the definition of a stored procedure and the parameters it requires into SYSTEM.SYSROUTINES and SYSTEM.SYSPARMS, and into the cache.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared. The PSERVER GROUP in which the procedure will run must exist (there must be at least one PSERVER defined in that group), before the procedure can be defined.

Authorization

The issuer of the CREATE PROCEDURE must have DBA authority.

Syntax

```
►►──CREATE PROCEDURE──procedure-name──┬──────────────────┬──(──┬────────────────┬──)─────────────────►
                                      └─AUTHID──authid───┘     └─┤ parameters ├─┘
```

```
      ┌─,──────────── (1) ──────────────────────────────────┐
      ↓   ┌─FENCED───────────────────────────────────────┐  │ (8)
►─────┴───┼──────────────────────────────────────────────┼──┴───────────────────────────────►◄
    │     ├─LANGUAGE──┬─ASSEMBLE─┬───────────────────────┤  │
    │     │           ├─C────────┤                       │  │
    │     │           ├─COBOL────┤                       │  │
    │     │           └─PLI──────┘                       │  │
    │     ├─EXTERNAL──┬───────────────────────────────┬──┤  │
    │     │           └─NAME──external-program-name───┘  │  │
    │     ├─SERVER GROUP──┬─────────────────────────┬────┤  │
    │     │               └─server-group-name───────┘    │  │
    │     │ ┌─DEFAULT SERVER GROUP YES─┐                 │  │
    │     ├─┴──────────────────────────┴─────────────────┤  │
    │     │ └─DEFAULT SERVER GROUP NO──┘                 │  │
    │     └──────────────────────────────────────────────┘  │
    │       ├─PARAMETER STYLE──────────────────────┐        │
    │       │                  (3)                 │        │
    │       │   ┌─GENERAL WITH NULLS──────────┐    │        │
    │       │             (2)                 │    │        │
    │       └───┴─GENERAL─────────────────────┴────┘        │
    │     ┌─STAY RESIDENT NO──┐                             │
    ├─────┴───────────────────┴─────────────────────────────┤
    │     └─STAY RESIDENT YES─┘                             │
    │     ┌─PROGRAM TYPE MAIN─────┐                         │
    ├─────┴───────────────────────┴─────────────────────────┤
    │                        (4)  │                         │
    │     └─PROGRAM TYPE SUB──────┘                         │
    ├─────┬────────────────────────────────────┬────────────┤
    │     └─RUN OPTIONS──run-time-options──────┘            │
    │     ┌─RESULT SET 0─────────────┐                      │
    ├─────┴──────────────────────────┴──────────────────────┤
    │     └─RESULT──┬─SET──┬──integer─┘                     │
    │               └─SETS─┘                                │
    │     ┌─COMMIT ON RETURN NO──┐                          │
    ├─────┴──────────────────────┴──────────────────────────┤
    │     └─COMMIT ON RETURN YES─┘                          │
    │                       (5)                             │
    │     ┌─NOT DETERMINISTIC─────┐                         │
    ├─────┴───────────────────────┴─────────────────────────┤
    │                    (6)      │                         │
    │     └─DETERMINISTIC─────────┘                         │
    │                   (7)                                 │
    │     ┌─CONTAINS SQL──────────┐                         │
    ├─────┼───────────────────────┼─────────────────────────┤
    │     │        (7)            │                         │
    │     ├─NO SQL────────────────┤                         │
    │     │                (7)    │                         │
    │     ├─READS SQL DATA────────┤                         │
    │     │                   (7) │                         │
    │     └─MODIFIES SQL DATA─────┘                         │
    │                (7)                                    │
    │     ┌─NO COLLID─────────────────┐                     │
    ├─────┴───────────────────────────┴─────────────────────┤
    │                              (7)│                     │
    │     └─COLLID──collection-id─────┘                     │
    ├─────┬────────────────────────────────────┬────────────┤
    │     │                 (7)                │            │
    │     ├─WLM ENVIRONMENT──────┬─name────────┤            │
    │     │                      └─(name,*)─┘  │            │
    │     │                     (7)            │            │
    │     └─NO WLM ENVIRONMENT─────────────────┘            │
    │                     (7)                               │
    │     ┌─ASUTIME NO LIMIT─────────┐                      │
    ├─────┴──────────────────────────┴──────────────────────┤
    │                             (7)│                      │
    │     └─ASUTIME LIMIT──integer───┘                      │
    │                              (7)                      │
    │     ┌─EXTERNAL SECURITY DB2──────────┐                │
    ├─────┴────────────────────────────────┴────────────────┤
    │                       (7)            │                │
    │     └─EXTERNAL SECURITY──────┬─USER──┤                │
    │                              └─DEFINER─┘              │
    │                (7)                                    │
    │     ┌─NO DBINFO──────┐                                │
    └─────┴────────────────┴────────────────────────────────┘
          │      (7)       │
          └─DBINFO─────────┘
```

Notes:

1 This parameter is included for compatibility with the DB2 family. If specified, it is ignored.

2 As an alternative to GENERAL, SIMPLE CALL may be used. This is for compatibility within the DB2 family.

3 As an alternative to GENERAL WITH NULLS, SIMPLE CALL WITH NULLS may be used. This is for compatibility within the DB2 family.

4 Currently, DB2 Server for VSE & VM supports stored procedures written as main programs only.

5 VARIANT may be specified as an alternative to NOT DETERMINISTIC. This is for compatibility within the DB2 family.

6 NOT VARIANT may be specified as an alternative to DETERMINISTIC. This is for compatibility within the DB2 family.

7 This parameter is included for compatibility with the DB2 family. If specified, it is ignored.

8 One or more clauses may be specified, however each clause may be specified at most once.

Description

Only the parameters that are meaningful to DB2 Server for VSE & VM are described.

procedure-name
Names the stored procedure. For DB2 Server for VSE & VM, the name must be an ordinary identifier of 18 characters or less. The name must not identify a stored procedure that already exists on the server. In addition, the name cannot be ′AUTHID′ or ′ACTION′.

authid
The authorization ID for the stored procedure. *authid* must be an ordinary identifier of 8 characters or less. If specified, then the stored procedure being defined will be accessible only by *authid*. Note that *authid* cannot be ′AUTHID′ or ′ACTION′.

LANGUAGE
Specifies the programming language used to create the stored procedure. All stored procedure programs must be designed to run in the IBM Language Environment.

ASSEMBLE
Specifies that the stored procedure is written in Assembler.

C Specifies that the stored procedure is written in C.

COBOL
Specifies that the stored procedure is written in COBOL.

PLI Specifies that the stored procedure is written in PLI.

Note that the LANGUAGE clause must be specified on the CREATE PROCEDURE statement.

EXTERNAL NAME *external-program-name*
Identifies the load module or phase associated with the stored procedure. The load module or phase does not need to exist when the CREATE PROCEDURE

statement is issued. However, when a CALL for the stored procedure is issued, the load module must exist and be accessible to the stored procedure server.

If *external-program-name* is not specified, the name of the load module or phase is assumed to be the same as the name of the stored procedure. In this case, the name of the stored procedure must be 8 characters or less. Note that the EXTERNAL clause must be specified on the CREATE PROCEDURE statement.

SERVER GROUP *server-group-name*
Identifies the name of the group of servers to be used to run this stored procedure. Server-group-name must be an ordinary identifier of 18 characters or less, and must be defined in SYSTEM.SYSPSERVERS.

If SERVER GROUP is specified without *server-group-name*, the stored procedure must be able to run in the default server group. At least one server must exist in the specified group. Note that the SERVER GROUP clause must be specified on the CREATE PROCEDURE statement.

DEFAULT SERVER GROUP
Specifies whether the stored procedure can run in the default server group.

YES The stored procedure can run in the default server group. This is the default.

NO The stored procedure cannot run in the default server group. If NO is specified, a *server-group-name* must be provided on the SERVER GROUP clause.

PARAMETER STYLE
Identifies the linkage convention used to pass parameters to the stored procedure. All of the linkage conventions provide arguments to the stored procedure containing the parameters specified on the SQL CALL statement. See the *DB2 Server for VSE & VM Database Administration* manual for more information. The following parameter styles options are valid for DB2 Server for VSE & VM:

GENERAL
If the GENERAL linkage convention is used:

- the SQL CALL statement must provide a parameter for each parameter expected by the stored procedure
- input parameters cannot be null
- nulls can be passed for output parameters only
- the stored procedure cannot return nulls for output parameters

Note that DB2 Server for VSE & VM does not support the parameter style DB2SQL.

GENERAL WITH NULLS
If the GENERAL WITH NULLS linkage convention is used:

- the SQL CALL statement must provide a parameter for each parameter expected by the stored procedure. When the database manager invokes the stored procedure, it sends it the parameters specified on the SQL CALL statement, as well as an array of indicator variables (with one indicator variable for each parameter). The stored procedure must contain a declaration for this array.
- input parameters can be null. This is achieved through the use of indicator variables, or by specifying the keyword null.
- the stored procedure can return nulls for output parameters, by using indicator variables.

STAY RESIDENT
Specifies whether the stored procedure load module or phase remains loaded in memory after the stored procedure ends. Possible values are:

NO The load module or phase is deleted from memory after the stored procedure ends. This is the default.

YES The load module or phase remains loaded in memory after the stored procedure ends.

PROGRAM TYPE
Specifies whether the stored procedure runs as a MAIN routine or as a SUB routine. Currently, DB2 Server for VSE & VM supports stored procedures written as MAIN routines only.

RUN OPTIONS
Specifies the Language Environment run-time options to be passed to the stored procedure. The options must be specified as a character string up to 254 bytes and must be enclosed in single quotation marks. If this option is not specified, or an empty string is passed, then DB2 Server for VSE & VM passes no run-time options to the Language Environment, and Language Environment uses its installation defaults. Note that DB2 Server for VSE & VM does not do any checking of the options provided. For a complete description of Language Environment run-time options, see *Language Environment for MVS & VM Programming Reference*.

RESULT SETS or RESULT SET
Specifies the maximum number of query result sets that can be returned by this stored procedure. The default is RESULT SETS 0, indicating that there are no result sets. The largest value that can be specified is 32767.

COMMIT ON RETURN
Indicates whether the transaction should be committed immediately upon return from the stored procedure.

NO The database manager should not issue COMMIT when the stored procedure returns. This is the default.

YES The database manager should issue COMMIT when the stored procedure returns when the following statements are true:
- The SQLCODE returned by the CALL statement is not negative
- The stored procedure is not in a *must abort* state

The COMMIT operation includes the work performed by the calling application as well as the stored procedure. Any cursors that are open when the COMMIT occurs will be closed during COMMIT processing.

Parameters

parameters:

```
     ┌─,──────────────────────────────────────────────────────────┐
     ↓ ┌─IN────┐                                                  │
├──────┼───────┼──┬────────────────┬─┤ data-type ├─┬──────────────┬─┴────────┤
       ├─OUT───┤  └─parameter-name─┘               │          (1) │
       └─INOUT─┘                                   └─AS LOCATOR───┘
```

data-type:

```
├──┬─INT──────────────────────────────────────────────────────────┬──────────┤
   ├─INTEGER──────────────────────────────────────────────────────┤
   ├─SMALLINT─────────────────────────────────────────────────────┤
   ├─REAL─────────────────────────────────────────────────────────┤
   ├─FLOAT────────────────────────────────────────────────────────┤
   ├─DOUBLE───────────────────────────────────────────────────────┤
   ├─DOUBLE PRECISION─────────────────────────────────────────────┤
   ├─┬─DECIMAL─┬──┬────────────────────────────────────┬──────────┤
   │ └─DEC─────┘  └─(─integer─┬────────────┬─)─┘          │
   │                          └─,integer───┘              │
   ├─┬─CHARACTER─┬──┬───────────┬──┬──────────────────────────┬───┤
   │ └─CHAR──────┘  └─(integer)─┘  └─FOR──┬─SBCS──┬──DATA─┘   │
   │                                      ├─MIXED─┤           │
   │                                      └─BIT───┘           │
   ├─VARCHAR(integer)──┬──────────────────────────────┬───────────┤
   │                   └─FOR──┬─SBCS──┬──DATA─┘              │
   │                          ├─MIXED─┤                      │
   │                          └─BIT───┘                      │
   ├─GRAPHIC(integer)─────────────────────────────────────────────┤
   └─VARGRAPHIC(integer)──────────────────────────────────────────┘
```

Notes:

1 This parameter is included for compatibility with the DB2 family. If specified, it is ignored.

The fields of the **parameters** syntax diagram are:

IN The parameter is an input-only parameter to the stored procedure.

OUT The parameter is an output-only parameter to the stored procedure.

INOUT
The parameter is both an input and output parameter to the stored procedure.

parameter-name
a one- to eight-character ordinary identifier defining the name of the parameter for use in messages. If you do not specify a name, the position of the parameter in the parameter list is used in the DB2 Server for VSE & VM messages.

INTEGER or INT
Large integer parameter

SMALLINT
Small integer parameter

REAL Single precision floating point

FLOAT, DOUBLE, or DOUBLE PRECISION
Double precision floating point

DECIMAL or DEC
Decimal parameter. The *(integer,integer)* optional arguments are the precision and scale respectively. The precision is the total number of digits from 1 to 31. The scale is the number of digits to the right of the decimal point, from 0 to the precision.

CHARACTER or CHAR
Fixed length character string parameter. The *(integer)* optional argument specifies the length of the string, from 1 to 254. If you do not specify *(integer)*, the length is set to 1.

VARCHAR
Varying length character string parameter. The maximum length is specified by the argument *(integer)* and varies from 1 to 32767. If the length is greater than 254 then it is a long string column.

GRAPHIC
Fixed-length graphic string parameter. The *(integer)* optional argument specifies the length of the string, from 1 to 127. If you do not specify the *(integer)* argument, the length is set to 1.

VARGRAPHIC
Varying-length graphic character string parameter. The maximum length is specified by the argument *(integer)* and varies from 1 to 16383 characters.

FOR subtype DATA
Specifies a subtype for a character string parameter. The subtype can be one of the following:

SBCS Specifies that the parameter is a single-byte character string.

MIXED
Specifies that the parameter holds mixed single-byte and double-byte data. This option is valid only when the DBCS value is set to YES.

BIT Specifies that the parameter holds bit data. Character conversion does not occur for data that is defined **FOR BIT DATA**

As an example, in the following *parameters* string:

```
PARM1 CHAR(10) IN, PARM2 INTEGER INOUT, PARM3 INT OUT
```

PARM1, PARM2, and PARM3 are identifiers for error messages. You can specify any name you want. The stored procedure associated with the PARMLIST string would expect three parameters:

- An input character parameter of length 10
- An integer parameter for both input and output
- An integer parameter for output only

Notes

1. If a parameter represents a DB2 Server for VSE & VM DATE, TIME, or TIMESTAMP value, it must be defined as CHARACTER or VARCHAR in the PARMLIST.

2. Refer to the appendices of the *DB2 Server for VSE & VM Application Programming* manual for the programming language declarations that correspond to the datatypes in the PARMLIST.

Examples

Example 1

```
CREATE PROCEDURE MYPROC (IN INT, IN PARM2 CHAR(10), OUT CHAR(20))
   EXTERNAL NAME MYMOD,
   LANGUAGE COBOL,
   PARAMETER STYLE GENERAL

CREATE PROCEDURE MYPROC2 (IN INT, IN CHAR(10), OUT CHAR(20))
   EXTERNAL NAME MYMOD2,
   LANGUAGE COBOL,
   PARAMETER STYLE GENERAL WITH NULLS,
   RUN OPTIONS 'HEAP(,,ANY),BELOW(4K,,),ALL31(ON),STACK,(,,ANY,)'
```

CREATE PSERVER

The CREATE PSERVER statement inserts the definition of a stored procedure server into SYSTEM.SYSPSERVERS and puts the new definition into the cache.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer of the CREATE PSERVER statement must have DBA authority.

Syntax

```
                                     ┌─,────────────────────────────┐
                                     │  (1)                         │
►►──CREATE PSERVER──procedure-server──▼─────┬──GROUP──group-name──────┬─┴─────────►◄
                                            │ ┌─AUTOSTART NO──┐       │
                                            ├─┴─AUTOSTART YES─┴───────┤
                                            └─DESCRIPTION──description─┘
```

Notes:

1 One or more clauses may be specified, however each clause may be specified at most once.

Description

procedure-server
: The name of the stored procedure server. The name must be an ordinary identifier of 8 characters or less. The name must not identify a stored procedure server that already exists on the server. In addition, the name cannot be one of the following:
GROUP
IMPLICIT
NOIMPLICIT
NORMAL
QUICK

GROUP
: The name of the group that this stored procedure server is in. Using stored procedure groups gives the database administrator more flexibility in defining the system. The use of stored procedure groups is optional; if the GROUP clause is not specified, the stored procedure server becomes part of the default group. If the GROUP clause is specified, the *group-name* must be an ordinary identifier of 1 to 18 characters.

AUTOSTART
: Determines whether the database manager will issue a START PSERVER command for this stored procedure server when the database is started.

NO The stored procedure server will not be started when the database is initialized. This is the default.

YES The stored procedure server will be started when the database is initialized.

DESCRIPTION
This field provides the database administrator with a place to provide information about this stored procedure server, such as virtual storage requirements, other servers in the group, and so on. *Description* can be up to 254 characters and must be enclosed in single quotation marks. The default is NULL.

Examples

Example 1

```
CREATE PSERVER SRV1 GROUP GRP1, AUTOSTART YES
CREATE PSERVER SRV2 GROUP GRP2, AUTOSTART YES
```

CREATE SYNONYM

The CREATE SYNONYM statement defines an alternative name for a table or view. This lets you refer to a table or view owned by another user without having to enter the qualified name. You may also define a synonym for a table or view that you own.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.

Syntax

```
►►──CREATE SYNONYM──synonym──FOR──┬─qualified_table_name─┬──────────────────────►◄
                                  └─qualified_view_name──┘
```

Description

synonym
: Provides an alternative name to use when referring to the table or view. The synonym must be an SQL identifier that is not identical to one of your synonyms or the unqualified name of a table or view that you own.

FOR
: Identifies the qualified name of the table or view for which the synonym is to be created. The qualifier is required, even when you are creating a synonym for one of your own tables or views. You can create a synonym for a table or view that does not as yet exist in the system catalog.

  The synonym is defined only for your authorization ID, that is, the authorization ID of the statement. If many users want to have the same synonym, each user must enter a CREATE SYNONYM statement.

qualified_table_name
qualified_view_name
: Identifies the object to which the synonym will apply. The name consists of two parts and denotes a table or view already described or which will be described in the catalog.

Notes

A synonym cannot be used with the table or view it represents in the same statement.

Examples

Define an alternative name, PARTS, for TRUDEAU.INVENTORY.

```
CREATE SYNONYM PARTS
  FOR TRUDEAU.INVENTORY
```

CREATE TABLE

The CREATE TABLE statement defines a table. You provide the name of the table and the names and attributes of its columns. Moreover, you may specify the dbspace where the table is to be created.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- DBA authority
- RESOURCE authority
- A private dbspace that was acquired for you.

If the table name is qualified by an identifier other than your authorization ID, you must have DBA authority.

See the description of the IN clause for further information on authorization.

Syntax

```
                                 ┌─,──────────────────────────────────────┐   ┌──────────────────────────────────────────┐
                                 ↓ (1)                                    │   ↓                                          │
►►──CREATE TABLE──table_name──(───┬─┤ column_definition_block ├──────────┬┴─)─┬──┬─IN──dbspace_name──────────────────────┬─┴──────────────►◄
                                  │                              (1)     │      │  │ (2)                                  │
                                  ├─┤ primary_key_block ├────────────────┤      │  └──────────DATA CAPTURE──┬─NONE────┬───┘
                                  ├─┤ referential_constraint_block ├─────┤                                    └─CHANGES─┘
                                  └─┤ unique_block ├─────────────────────┘
```

Notes:

1 There can be up to 255 columns in a table.

2 The same clause must not be specified more than once.

column_definition_block:

```
├──column_name──┤ data_type ├──┬───────────────────────────┬──────────────────────────────────────────────────┤
                               │ (1)                       │
                               └────────┤ fieldproc_block ├┘
```

Notes:

1 These clauses may be specified in any order.

data_type:

```
├──┬─INTeger─────────────────────────────────────────────────────┬──┬────────────────────────────────┬──────────────────────────────────┤
   ├─SMALLINT────────────────────────────────────────────────────┤  └─NOT NULL──┬─────────────────┬─┘
   │          ┌─(53)──────┐                                      │              ├─UNIQUE──────────┤
   ├──┬─FLOAT─┴─(integer)─┴──┬───────────────────────────────────┤              └─PRIMARY KEY─────┘
   │  ├─REAL─────────────────┤                                   │
   │  └─DOUBLE PRECISION─────┘                                   │
   │               ┌─(5,0)────────────────────────┐              │
   ├──┬─DECimal─┬──┴─(──integer──┬────────────┬─)─┴──────────────┤
   │  └─NUMERIC─┘                └─,integer───┘                  │
   │                ┌─(1)───────┐                                │
   ├──┬─CHARacter───┴─(integer)─┴─┬──┬──────────────────────┬────┤
   │  ├─VARCHAR──(integer─────────┤  ├─FOR SBCS DATA────────┤    │
   │  └─LONG VARCHAR──────────────┘  ├─FOR MIXED DATA───────┤    │
   │                                 ├─FOR BIT DATA─────────┤    │
   │                                 └─CCSID──integer───────┘    │
   │              ┌─(1)───────┐                                  │
   ├──┬─GRAPHIC───┴─(integer)─┴──┬──┬──────────────────┬─────────┤
   │  ├─VARGRAPHIC──(integer)────┤  └─CCSID──integer───┘         │
   │  └─LONG VARGRAPHIC──────────┘                               │
   ├─DATE────────────────────────────────────────────────────────┤
   ├─TIME────────────────────────────────────────────────────────┤
   └─TIMESTAMP───────────────────────────────────────────────────┘
```

fieldproc_block:

```
├──FIELDPROC──program_name──┬──────────────────────┬──────────────────────┤
                            │    ┌─,──────┐        │
                            │    ▼        │        │
                            └─(────constant──┴─)───┘
```

primary-key-block:

```
                   ┌─,──────────────────────────┐
                   ▼ (1)                        │
├──PRIMARY KEY──(─────────column_name──┬─ASC──┬─┴──)──┬─PCTFREE = 10──────┬──────┤
                                       └─DESC─┘       └─PCTFREE = integer─┘
```

Notes:

1 A PRIMARY KEY can have up to 16 columns.

referential-constraint-block:

```
                                         ┌─,───────────┐
                                         ▼             │
├──FOREIGN KEY──┬─────────────────┬──(────column_name──┴──)──REFERENCES──table_name──────►

►──┬─────────────────────────────┬─────────────────────────────────────┤
   │              ┌─RESTRICT─┐   │
   └─ON DELETE────┼─CASCADE──┼───┘
                  └─SET NULL─┘
```

(constraint_name is the optional branch under FOREIGN KEY.)

unique-block:

```
                                          ┌─,───────────────────────────┐
                                          ▼ (1)                         │
├──UNIQUE──┬─────────────────┬──(─────────column_name──┬─ASC──┬─┴──)──┬─PCTFREE = 10──────┬──────┤
           └─constraint_name─┘                         └─DESC─┘       └─PCTFREE = integer─┘
```

Notes:

1 There can be up to 16 columns on a unique constraint.

Description

table_name

Provides a name for the table. The name, including the implicit or explicit qualifier, must not identify a table, view, or synonym that already exists at the application server.

If the table name is qualified, the qualifier is the owner of the table. Otherwise, the authorization ID of the statement is the owner of the table. The owner has all privileges on the table. The privileges can be granted by the owner and cannot be revoked from the owner.

If user SCOTT preprocesses a program that creates a table named SUMMARY, and user JONES runs the program, the owner of the SUMMARY table is

SCOTT. Note that JONES must be a DBA to run the program. Any program preprocessed by SCOTT can refer to the SUMMARY table simply by the name SUMMARY. When another authorization ID preprocesses a program that refers to the SUMMARY table, the program must use SCOTT as a prefix to the table_name, SCOTT.SUMMARY.

(*column_name***,...)**
Names a column of the table. Do not qualify *column_name* and do not use the same name for more than one column of the table.

data_type
Specifies one of the types in the following list.

INTeger
For a large integer. The value may range from -2147483648 to 2147483647.

SMALLINT
For a small integer. The value may range from -32 768 to 32 767.

FLOAT(*integer***)**
FLOAT
For a floating-point number. If the integer is between 1 and 21 inclusive, the format is that of single precision floating-point. If the integer is between 22 and 53 inclusive, the format is that of double precision floating-point. If the integer is omitted from the specification, double precision floating-point is assumed.

In place of FLOAT(*integer*) you may specify either:

REAL	For single precision floating-point
DOUBLE PRECISION	For double precision floating-point

DECIMAL(*precision-integer***,***scale-integer***)**
For a packed decimal number. The first integer is the precision of the number; that is, the total number of digits; it can range from 1 to 31. The second integer is the scale of the number; that is, the number of digits to the right of the decimal point; the scale of the number can range from 0 to the precision of the number.

DECIMAL(*p*) can be used for DECIMAL(*p*,0), and DECIMAL can be used for DECIMAL(5,0).

NUMERIC
NUMERIC is a synonym for DECIMAL.

CHARacter(*integer***)**

CHARacter
For a fixed-length character string of length *integer*, which can range from 1 to 254 bytes. If the length specification is omitted, a length of 1 character is assumed.

VARCHAR(*integer***)**
For a varying-length character string of maximum length *integer*, which can range from 1 to 32 767 bytes. An *integer* greater than 254 defines a long string column.

LONG VARCHAR
For a varying-length character string with a maximum length of 32 767 bytes.

A LONG VARCHAR column is always a long string column (even if its actual length is 254 or less).

GRAPHIC(*integer*)
For a fixed-length graphic string of length *integer*, which can range from 1 to 127 double-byte characters. If the length specification is omitted, a length of 1 character is assumed.

VARGRAPHIC(*integer*)
For a varying-length graphic string of maximum length *integer*, which must range from 1 to 16 383 double-byte characters. An *integer* greater than 127 double-byte characters defines a long string column.

LONG VARGRAPHIC
For a varying-length string of double-byte characters, of maximum length 16 383 bytes. A LONG VARGRAPHIC column is always a long string column (even if its actual length is 127 double-byte characters or less).

DATE
For a date.

TIME
For a time.

TIMESTAMP
For a timestamp.

column clauses (can be specified in any order)

NOT NULL
Prevents the column from containing null values.

NOT NULL PRIMARY KEY
Establishes a primary key column. (See “PRIMARY KEY” in the *primary-key-block* section for a definition of a primary key.)

NOT NULL UNIQUE
Establishes a unique index on the column. (See “UNIQUE” in the “unique-block” section for a definition of a unique index.)

FOR SBCS DATA
Indicates the column will contain single-byte characters.

FOR MIXED DATA
Indicates the column might contain values that have a mixture of single-byte or double-byte characters.

FOR BIT DATA
Indicates that the values of a character column are not associated with a coded character set and therefore are never converted. For example, the bit pattern of the data should not be modified when moving table data between ASCII and EBCDIC environments. The database manager sets the SUBTYPE column in the SYSCOLUMNS catalog table to 'B' when this option is specified.

CCSID *integer*
Uniquely identifies an encoding scheme and one or more pairs of character sets and code pages, for either character or graphic data.

Depending on the specification of subtypes and CCSIDs, the database manager assigns different values:
- If either SBCS or mixed data is specified, then the database default CCSID for the subtype is assigned.
- If a CCSID is specified, then the subtype that matches the CCSID is assigned.

- If neither SBCS nor mixed data is specified and a CCSID is also not specified, then first the default subtype is assigned; then the database default CCSID for that subtype is assigned.
- If graphic data is specified without a CCSID, then the database default CCSID for graphic data is assigned.

The choice of CCSID, including allowing it to default, may significantly affect performance. For performance implications related to CCSID, consult the *DB2 Server for VSE & VM Performance Tuning Handbook* manual.

fieldproc-block

FIELDPROC *program_name*
Names a field procedure for the column. Use a field procedure only with a short string column. The column has no field procedure if you omit FIELDPROC.

constant
Is a parameter of the field procedure. A parameter list is optional. The number of parameters and the data type of each are determined by the field procedure. The maximum length of the parameter list is 254 bytes, including commas, but excluding insignificant blanks and the delimiting parentheses after blank compression occurs.

primary-key-block

PRIMARY KEY
Is a set of column values in the table that enforces a unique constraint. Only one primary key is allowed in a parent table. Primary key values must be unique and must be defined as NOT NULL.

Defining a primary key on a table sets up the table to be referenced by another table's foreign key to establish a referential constraint.

column_name
Identifies the column or columns that comprise the primary key. Each *column_name* must be an unqualified name that identifies a column of the table, and that column must be defined as NOT NULL. No column in a primary key can contain a long string. The same column cannot be specified more than once.

ASC
Creates the primary key such that the values from this column are arranged in ascending order. This is the default.

DESC
Creates the primary key such that the values from this column are arranged in descending order.

PCTFREE
Is the percentage of space in each index page reserved for later insertions and updates of the primary key. The integer can range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take up more space, but reduces the time required to insert or update primary key rows of the indexed table.

referential-constraint-block

FOREIGN KEY
Defines a foreign key which consists of one or more columns in a dependent table that together must take on a value that exists in the

primary key of the related parent table. The columns in the dependent table may contain nulls. If any of the columns contain a null value, the foreign key is considered null.

constraint_name
Provides a name for the referential constraint. You cannot use a *constraint_name* more than once in the same table. Although the database manager generates a constraint_name if you do not specify one, you should specify your own *constraint_name* to make it easier for you to drop, activate, and deactivate the foreign key.

column_name
Identifies the column or columns that comprise the foreign key. Each *column_name* must be an unqualified name that identifies a column of the table. The data type and length of foreign key columns must match the data type and length of the primary key columns. Only the null attribute of a foreign key column may be different. The same column cannot be specified more than once.

REFERENCES *table_name*
Specifies the name of the parent table involved in the referential constraint. The *table_name* cannot identify the table that is being created. The identified table must already exist and cannot be the system catalog table.

ON DELETE
Defines the delete rule to be followed when a row is deleted from the parent table in a relationship.

RESTRICT
Prevents deletion of a parent row until all the dependent rows have been deleted. RESTRICT is the default value.

CASCADE
Causes all dependent rows to be deleted also.

SET NULL
Sets to null all columns of the foreign keys in each dependent row that can contain nulls. At least one column of the foreign key in the dependent table must be able to contain nulls.

The following restriction for ON DELETE is checked when a table is created.

- If a table has more than one referential constraint referencing the same parent, all the delete rules on those constraints must be the same and must not be SET NULL.
- If a table is delete-connected to the same parent through multiple paths, all of the delete rules on a path, except for the last one, must be CASCADE. The last constraint on all paths must be the same, and must not be SET NULL.

unique-block

UNIQUE
Adds a unique index for the column or columns specified. If there are duplicates in the values of the columns, then a unique constraint is not added.

constraint_name
Provides a name for the unique constraint. You cannot use the same

constraint_name more than once in the same table. Although the database manager generates a constraint_name if you do not specify one, you should specify your own constraint_name to make it easier for you to drop, activate, and deactivate the unique constraint.

column_name
Identifies the column or columns that comprise the unique key. Each *column_name* must be an unqualified name that identifies a column of the table, and that column must be defined as NOT NULL. No column in a unique constraint can be nullable. You cannot specify the same column more than once. These columns should not be the same as that of a primary key in the same table.

ASC
Creates the unique key such that the values from this column are arranged in ascending order. This is the default.

DESC
Creates the unique key such that the values from this column are arranged in descending order.

PCTFREE
Is the percentage of space in each index page reserved for later insertions and updates of unique keys. The integer may range from 0 to 99, but for practical purposes should not exceed 50. Increasing PCTFREE causes the index to take up more space, but reduces the time required to insert or update unique keys.

IN *dbspace_name*
The name of the dbspace into which the table is to be placed. The dbspace must exist at the application server. The default qualifier portion of *dbspace_name* is the authorization ID of the statement. If *dbspace_name* is omitted, the table will be created in one of the owner's private dbspaces (if the owner does not have any private dbspace, an error condition will result).

A newly created table is placed in one of the existing dbspaces of the database according to the following rules:

1. Specifying a *dbspace_name* in the CREATE TABLE statement puts the table into the named dbspace. The owner of the dbspace must be either the user who preprocessed the current program, or PUBLIC. If you have DBA authority, you can create a table in a private dbspace of any user by qualifying the *dbspace_name* with its owner's user ID, as follows:
   ```
   CREATE TABLE ... IN SCOTT.DSP3
   ```
2. Not specifying a *dbspace_name* in the CREATE TABLE statement puts the table into any private dbspace owned by the authorization ID who preprocessed the program. Consider the following case:
   a. The person who preprocessed the program has DBA authority.
   b. No dbspace is specified.
   c. The table name is qualified with an *authorization_name*.

   The database manager places the table into any private dbspace owned by the specified authorization ID. If there is no such dbspace, an error condition results.
3. If the *dbspace_name* is not qualified, the database manager will not place the table into a nonrecoverable dbspace by default. Specify the *dbspace_name* to create a table in a nonrecoverable dbspace.

4. If both the *table_name* and the *dbspace_name* are qualified, but are not qualified with the same *authorization_name*, and the authorization ID who preprocessed the program has DBA authority, the database manager uses both qualifiers. That is, if JIM has DBA authority, he may create table KELLI.SUPPLIERS in JOE.SPACE1.

   Table 9 summarizes where a table is placed depending on what is specified. X represents the user ID of the person who preprocessed the program. X is denoted as optional below because if no user ID is specified, the creator always defaults to the user ID of the person who preprocessed the program (X). Y represents some other user ID.

Table 9. Default table placement when user X preprocesses the program.

DBA Authority Needed?	Table Creator	Table Name	DBSPACE Owner	DBSPACE Name	Database Manager Action
No	X	A			User X creates X.A in a private dbspace owned by X.
Yes	Y	A			User X creates Y.A in any private dbspace owned by Y.
No	X	A	X	B	User X creates X.A in X.B[1]
Yes		A	Y	B	User X creates X.A in Y.B
Yes	Y	A		B	User X creates Y.A in Y.B[1]
Yes	Y	A	Z	B	User X creates Y.A in Z.B

[1] If there is no PRIVATE DBSPACE B, but there is PUBLIC DBSPACE B, the PUBLIC DBSPACE will be used.

Concatenate the desired *authorization_names* to both the *table_name* and the *dbspace_name* to avoid confusion. This concatenation always identifies both the owner of the table and where the table will be placed.

DATA CAPTURE
Specifies if log records for this table should contain the full before image (DATA CAPTURE CHANGES) or the partial before image (DATA CAPTURE NONE) for UPDATE operations. If this option is not specified, it defaults to DATA CAPTURE NONE. If DataPropagator Capture is being used to capture changes to this table, DATA CAPTURE CHANGES must be specified. If DataPropagator Capture is not being used to capture updates to this table, DATA CAPTURE NONE should be specified to reduce the amount of data logged for updates to this table.

NONE
Include the partial before image in the log records for UPDATE operations. If DataPropagator Capture is not being used to capture updates to this table, DATA CAPTURE NONE should be specified to reduce the amount of data logged for updates to this table.

CHANGES
Include the full before image in the log records for UPDATE operations. If

DataPropagator Capture is being used to capture changes to this table, DATA CAPTURE CHANGES must be specified.

Notes

Once a table has been created, the data types of its columns may not be changed, and columns may not be deleted from a table. However, new columns may be added to the table (with the ALTER TABLE statement).

Byte counts

The sum of the byte counts of the columns must not be greater than 4077. The list that follows gives the byte counts of columns by data type for columns that do not allow null values. For a column that allows null values the byte count is one more than shown in the list.

Data Type	Byte Count
INTEGER	4
SMALLINT	2
FLOAT(*n***)**	If *n* is from 1 to 21, the byte count is 4. If *n* is from 22 to 53, the byte count is 8.
DECIMAL(*p***,** *s***)**	(*p*/2 + 1), rounded down to an integer.
CHAR(*n***)**	*n*
VARCHAR(*n***)**	*n*+2, or 6 if n>254
LONG VARCHAR	6
DATE	4
TIME	3
TIMESTAMP	10
GRAPHIC(*n***)**	2*n*
VARGRAPHIC(*n***)**	2*n*+2, or 6 if *n*>127
LONG VARGRAPHIC	6

Dbspace use for long string columns: Actual data for a long string column is stored in its own internal table in the dbspace. Thus, each table that contains long string columns uses one of the available 255 tables in the dbspace.

Tables as part of a referential structure: Tables that are part of a referential structure must be defined in DBSPACEs that are in the same type of storage pool. That is, both parent and dependent tables in the same referential structure must be in DBSPACEs that are in either a recoverable storage pool or a nonrecoverable storage pool. If the tables are not in the same type of storage pool, any attempt to define or change a referential constraint will produce an error.

Examples

Example 1

Given that you have DBA authority, create a table named 'ROSSITER.INVENTORY' with the following columns:

part number	integer between 1 and 9,999, must be present
description	character of length 1 to 24
quantity on hand	integer between 0 and 100,000

```
CREATE TABLE ROSSITER.INVENTORY
     (PARTNO         SMALLINT     NOT NULL,
     DESCRIPTION    VARCHAR(24),
     QONHAND        INT)
```

Example 2

Given that you have DBA authority, create the SITE1_SUPPLIERS table in the PUBLIC dbspace SPACE3 with the following columns and make KRISTEL the owner of the table:

supplier number	integer between 1 and 99, must be present
name	character of length 15
address	character of length 1 to 35

```
CREATE TABLE KRISTEL.SITE1_SUPPLIERS
     (SUPPNO         SMALLINT     NOT NULL,
   NAME          CHAR(15),
   ADDRESS       VARCHAR(15) )
  IN "PUBLIC".SPACE3
```

Example 3

Create the EQUIPMENT table in one of your private dbspaces with the following columns:

equipment number	integer between 0100000 and 8999999
equipment description	varying length string of up to 50 characters
location	varying length string of up to 50 characters
equipment owner	the number of the department that owns this equipment, null if not owned by any department

Ensure there is a unique entry in the table for each piece of equipment and order the entries in ascending order by equipment number (EQUIP_NO).

Also define a referential constraint with the table so that the equipment owner (EQUIP_OWNER) must be a department (DEPTNO) that is present in the DEPARTMENT table. If a department is removed from the DEPARTMENT table, the equipment owner values for all equipment owned by that department should become unassigned (that is, set to null). Give the constraint a name of DEPT_EQUIP.

```
CREATE TABLE EQUIPMENT
(EQUIP_NO INT      NOT NULL,
             EQUIP_DESC     VARCHAR(50),
             LOCATION       VARCHAR(50),
             EQUIP_OWNER    CHAR(3),
             PRIMARY KEY(EQUIP_NO),
             FOREIGN KEY DEPT_EQUIP (EQUIP_OWNER)
               REFERENCES DEPARTMENT
               ON DELETE SET NULL )
```

Example 4

On a DB2 Server for VM or DB2 Server for VSE application server with mixed data supported and with a default character subtype (that is, CHARSUB) of mixed, create a table named 'MAPS' in one of your private dbspaces. This table is designed to be maintained from a DB2 for OS/2 application requester. The table is to have the following columns (all values must be present):

Column	Description	Data Stored
MAP_NUMBER	map number	7 SBCS characters (to be converted to EBCDIC)
LAST_UPD	last update	date

Column	Description	Data Stored
DESC	description	up to 40 ASCII mixed (to be converted to EBCDIC)
MAP	the map	up to 4000 bytes (not to be converted to EBCDIC)

```
CREATE TABLE MAPS
            (MAP_NUMBER CHAR(7)  FOR SBCS DATA NOT NULL,
            LAST_UPD       DATE  NOT NULL,
            DESC           VARCHAR(40)  NOT NULL,
            MAP            VARCHAR(4000)  FOR BIT DATA NOT NULL)
```

Example 5

Similar to example 4, except that the default character subtype in the system is SBCS.

```
CREATE TABLE MAPS
       (MAP_NUMBER CHAR(7) NOT NULL,
       LAST_UPD       DATE                        NOT NULL,
       DESC           VARCHAR(40)  FOR MIXED DATA NOT NULL,
       MAP            VARCHAR(4000)       FOR BIT DATA NOT NULL)
```

Example 6

Similar to example 5, except that not only is the default character subtype SBCS but the default CCSIDs for both SBCS and mixed are not those required in the table (the table requires an SBCS CCSID of 290 and a mixed CCSID of 5026).

```
CREATE TABLE MAPS
       (MAP_NUMBER     CHAR(7)   CCSID 290    NOT NULL,
       LAST_UPD       DATE   NOT NULL,
       DESC           VARCHAR(40)   CCSID 5026   NOT NULL,
       MAP            VARCHAR(4000)        FOR BIT DATA NOT NULL)
```

Example 7

Create a table and include the partial before image on UPDATE log records because DataPropagator Capture is not capturing updates for this table:

```
CREATE TABLE SALARY1 .....
         OR
CREATE TABLE SALARY1 .....
   DATA CAPTURE NONE
```

Example 8

Create a table and include the full before image on UPDATE log records because DataPropagator Capture requires this information for update log records:

```
CREATE TABLE SALARY2 .....
   DATA CAPTURE CHANGES
```

CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables or views.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- DBA authority, or
- For each table or view identified in the subselect:
  - The SELECT privilege on the view, or
  - Ownership of the table or view.

If the specified view name includes a qualifier that is not the same as the authorization ID of the statement, the privileges held by the authorization ID of the statement must include DBA authority. If the view name is qualified by an identifier that is not your authorization ID, you must have DBA authority.

Syntax

```
►►──CREATE VIEW──view_name──┬─────────────────────────┬──────────────────────────►
                            │    ┌─,───────────┐      │
                            │    ▼             │      │
                            └─(──┴─column_name─┴──)───┘

►──AS──subselect──┬───────────────────┬─────────────────────────────────────────►◄
                  └─WITH CHECK OPTION─┘
```

Description

view_name
: Provides a name for the view. The name, including the implicit or explicit qualifier, must not identify a table, view, or synonym that already exists at the application server.

  The implicit or explicit qualifier of the view_name is the owner of the view. The owner always acquires the SELECT privilege on the view, and the authority to drop the view. The SELECT may be granted to others only if the owner has the authority to grant the SELECT privilege on every table or view identified in the first FROM clause of the subselect.

  If the owner has the INSERT, UPDATE, or DELETE privileges on the table or view identified in the first FROM clause of subselect, then the owner also acquires these privileges on the view being created. Only the owner of the table or view identified in the first FROM clause of the subselect can grant the privilege.

(*column_name***,...)**
: Names the columns in the view. If you specify a list of column names, it must consist of as many names as there are columns in the result table of the subselect. Each *column_name* must be unique and unqualified. If you do not specify a list of column names, the columns of the view inherit the names of the columns of the result table of the subselect.

You must specify a list of column names if the result table of the subselect has duplicate column names or an unnamed column (a column derived from a constant, function, or expression).

AS *subselect*
Defines the view. At any time, the view consists of the rows that would result if the subselect were processed. Note that the subselect is not processed when the view is created, which means that semantic errors (for example, specifying "WHERE COL = '10'" when COL is a decimal column) are not detected until the view is used. To determine whether a statement contains semantic errors, you can enter a 'SELECT *' against the view after creating it.

subselect must not reference host variables. For an explanation of *subselect*, see Chapter 5, “Queries,” on page 121.

WITH CHECK OPTION
Specifies the constraint that every row of the view must conform to the search condition of the view. The constraint is enforced by the database manager whenever rows of the view are inserted or updated. If the search condition is not true for any inserted or updated row, an error is returned, and no rows are inserted or updated.

The search condition of a view is the search condition specified in the first WHERE clause of the subselect used to define the view.

WITH CHECK OPTION must not be specified if the view is read-only or if its search condition includes a subquery. WITH CHECK OPTION is ignored if the view is updateable but does not have a search condition. If WITH CHECK OPTION is specified for an updateable view that does not allow inserts, the constraint only applies to updates.

If WITH CHECK OPTION is omitted, the search condition of the view is not used in the checking of any insert or update operations. The view can then be used to insert a row that does not conform to the search condition of the view and to update a row so that it no longer conforms to the search condition of the view. A row that does not conform to the search condition of a view cannot be retrieved using that view. It is also possible for this situation to exist when WITH CHECK OPTION is specified; this can happen when the view is directly or indirectly dependent on a view that was defined without the constraint.

The WITH CHECK OPTION constraint on view V is inherited by any updateable view that is directly or indirectly dependent on V. Thus, if an updateable view is defined on V, the constraint on V also applies to that view, regardless of whether WITH CHECK OPTION is specified in the definition of that view.

Consider the following updateable views:
- V1 defined on T0
- V2 defined on V1 WITH CHECK OPTION
- V3 defined on V2
- V4 defined on V3 WITH CHECK OPTION
- V5 defined on V4

When a row of V5 or V4 is inserted or updated, it is checked against the conjunction of the search conditions of V4 and V2. When a row of V3 or V2 is inserted or updated, it is checked against the search condition of V2. When a row of V1 is inserted or updated, it is not checked against any search condition.

FOR UPDATE OF, ORDER BY, and UNION cannot be used in the definition of a view.

Notes

Read-only views

A view is *read-only* if its definition involves any of the following:

- The first FROM clause identifies more than one table or view
- The first FROM clause identifies a read-only view
- The first SELECT clause specifies the keyword DISTINCT
- The outer subselect contains a GROUP BY clause
- The outer subselect contains a HAVING clause
- The first SELECT clause contains a column function
- It contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table
- The first FROM clause identifies a catalog table with no updatable columns.

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement. Note that the fact that a table contains expressions does not make it a *read only view*. As long as the expressions reference a single base table, such a view can be used to delete rows from the base table or to update columns that are defined without expressions. Rows can also be inserted into such views if the columns defined as expressions are nullable.

If you use a 'SELECT *' clause in the view definition and then you add a column to an underlying table (with the ALTER TABLE statement), the new column will *not* appear in the view.

There is no specific number for the limit on the number of columns in a view, because it depends on many factors which affect this limit. A view of up to 140 columns should work in most situations.

Examples

Example 1

Create a view named MA_PROJ upon the PROJECT table that contains only those rows with a project number (PROJNO) starting with the letters ‘MA’.

```
CREATE VIEW MA_PROJ
AS SELECT * FROM PROJECT
  WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 2

Create a view as in example 1, but select only the columns for project number (PROJNO), project name (PROJNAME) and employee in charge of the project (RESPEMP).

```
CREATE VIEW MA_PROJ
AS SELECT PROJNO, PROJNAME, RESPEMP
  FROM PROJECT
       WHERE PROJNO LIKE 'MA____'
```

Example 3

Create a view as in example 2, but, in the view, call the column for the employee in charge of the project IN_CHARGE.

```
CREATE VIEW MA_PROJ
(PROJNO, PROJNAME, IN_CHARGE)
  AS SELECT PROJNO, PROJNAME, RESPEMP FROM PROJECT
       WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Note: Even though you are changing only one of the column names, the names of all three columns in the view must be listed in the parentheses that follow MA_PROJ.

Example 4

Create a view named PRJ_LEADER that contains the first four columns (PROJNO, PROJNAME, DEPTNO, RESPEMP) from the PROJECT table together with the last name (LASTNAME) of the person who is responsible for the project (RESPEMP). Obtain the name from the EMPLOYEE table by matching EMPNO in EMPLOYEE to RESEMP in PROJECT.

```
CREATE VIEW PRJ_LEADER
AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO
```

Example 5

Create a view as in example 4, but in addition to the columns PROJNO, PROJNAME, DEPTNO, RESEMP and LASTNAME, show the total pay (SALARY + BONUS +COMM) of the employee who is responsible. Also select only those projects with mean staffing (PRSTAFF) greater than one.

```
CREATE VIEW PRJ_LEADER (PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, TOTAL_PAY )
  AS SELECT PROJNO, PROJNAME, DEPTNO, RESPEMP, LASTNAME, SALARY+BONUS+COMM
       FROM PROJECT, EMPLOYEE
       WHERE RESPEMP = EMPNO AND PRSTAFF > 1
```

Example 6

This example shows something that can happen when a view defined WITH CHECK OPTION depends on a view defined without this option. In this case, a view named VV depends on a view named WW, and WW depends on the EMPLOYEE sample table. The view definitions are as follows:

```
CREATE VIEW WW
  AS SELECT * FROM EMPLOYEE
       WHERE SALARY < 35000.00

CREATE VIEW VV
  AS SELECT * FROM WW
       WHERE SALARY > 30000.00
       WITH CHECK OPTION
```

Assume both views have a single owner, who uses VV in the following UPDATE statement:

```
UPDATE VV SET SALARY = SALARY + 5000.00
```

The update applies to every row in which SALARY is greater than 30000 but less than 35000. After the update, all rows that were visible to WW have salaries greater than 35000. Such salaries conform to the search condition of VV but not to that of WW. Even so, because WW is not defined WITH CHECK OPTION, all these rows are updated. As a result, any row in EMPLOYEE that was visible to VV is now invisible to WW and is therefore also invisible to VV.

DECLARE CURSOR

The DECLARE CURSOR statement defines the cursor through which a user may OPEN, FETCH, PUT, or CLOSE the results of a statement prepared using PREPARE. There are two types of cursor:

- A *query cursor* is a cursor associated with a select-statement and used by an application to access rows in a result table.
- An *insert cursor* is a cursor associated with an insert-statement and used by an application to insert rows into an active set.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

No authorization is required to use this statement except in the case of Fortran. Programs in these languages will fail if the authorization ID is not the same as that used to preprocess the program.

To use the OPEN statement for the cursor, the privileges held by the authorization ID of the statement are outlined below.

The cursor must always be linked to a *select-statement* or an *insert-statement*. This linked statement may be identified in one of three ways. The authorization required to manipulate the cursor varies accordingly.

1. If the statement is a fullselect of the form identified by *select-statement*, then the authorization ID is the one that is used to preprocess the program. This authorization ID must have SELECT privileges on every table and view identified in the SELECT.
2. If the statement is an INSERT (using VALUES), then the authorization ID is the one that preprocesses the program. This authorization ID must have INSERT authority on that table.
3. If the statement is a prepared SELECT or INSERT (using VALUES) statement named by a *statement_name* clause, then the authorization ID is the run-time authorization ID. Depending on whether the statement to be prepared is a SELECT or INSERT (using VALUES), this authorization ID must have appropriate SELECT or INSERT privileges.

Someone with DBA authority may do any of the above.

Syntax

```
►►──DECLARE──cursor-name──CURSOR──┬─────────────────┬──FOR───────────────────────►
                                  ├─WITH RETURN─────┤
                                  │            (1)  │
                                  └─WITH HOLD───────┘

►──┬─select-statement─┬──────────────────────────────────────────────────────────►◄
   └─statement-name───┘
```

Notes:

1 Note that DB2 Server for VSE & VM does not support CURSOR WITH HOLD.

Description

cursor_name
: Provides a name for the cursor. The name must not be the same as the name of another cursor declared in your source program. In REXX, *cursor_name* must not be the same as a *statement_name* prepared in the program.

A cursor in the open state designates an *active set* (for query cursors this is also known as the cursor's *result table*) and a position relative to the rows of that active set. The active set is specified by the SELECT or INSERT statement of the cursor.

A program may contain many DECLARE CURSOR statements that define different cursors and associate them with different queries or inserts. During processing of a program, several of these cursors may be in the open state at one time. The DECLARE CURSOR statement that defines a cursor must occur earlier in the program than any cursor manipulation statement operating on that cursor. The DECLARE CURSOR statement does not result in any actual processing when the program is run (that is, it does not automatically open the cursor).

The DECLARE CURSOR statement must precede all statements that explicitly reference the cursor by name.

Following is a description of each form of DECLARE CURSOR.

DECLARE CURSOR for SELECT

select-statement
: Specifies the SELECT statement of the cursor.

  The *select-statement* must not include parameter markers, but can include references to host variables. In host languages, other than assembler and REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. Host variable declarations can follow the DECLARE CURSOR statement in assembler and host variables are not declared at all in REXX.

The result table is *read-only* if any of the following are true:
- The first FROM clause identifies more than one table or view.
- The first FROM clause identifies a read-only view.
- The first SELECT clause specifies the keyword DISTINCT.
- The outer subselect contains a GROUP BY clause.
- The outer subselect contains a HAVING clause.
- The first SELECT clause contains a column function.
- The select-statement contains a subquery such that the base object of the outer subselect and of the subquery is the same table.
- The select-statement contains a UNION or UNION ALL operator.
- The select-statement includes an ORDER BY clause.
- Isolation UR is used.

If the select-statement of a cursor contains CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, all references to these special registers will yield the same value on each FETCH. This value is determined when the cursor is opened.

Examples:

Example 1: In a PL/I program, use the cursor C1 to fetch the values for a given project (PROJNO) from the first four columns of the EMP_ACT table a row at a time and put them into the following host variables: EMP (char(6)), PRJ (char(6)), ACT (smallint), and TIM (dec(5,2)). Obtain the value of the project to search for from the host variable SEARCH_PRJ (char(6)).

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  EMP             CHAR(6);
  DCL  PRJ             CHAR(6);
  DCL  SEARCH_PRJ      CHAR(6);
  DCL  ACT             BINARY FIXED(15);
  DCL  TIM             DEC    FIXED(5,2);
EXEC SQL  END DECLARE SECTION;
  .
  .
  .
EXEC SQL  DECLARE C1 CURSOR FOR
            SELECT EMPNO, PROJNO, ACTNO, EMPTIME
              FROM EMP_ACT
              WHERE PROJNO = :SEARCH_PRJ;

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;

IF SQLSTATE = '02000' THEN
  CALL DATA_NOT_FOUND;
ELSE
  DO WHILE (SUBSTR(SQLSTATE,1,2) = '00' | SUBSTR(SQLSTATE,1,2) = '01');
    EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;
  END;

EXEC SQL  CLOSE C1;
  .
  .
  .
```

Example 2: In a PL/I program, declare a cursor named INCREASE to return from the EMPLOYEE table all the employee numbers (EMPNO), surnames (LASTNAME) and price (SALARY increased by 10 percent) of people who have the job of clerk (JOB). Order the result table in descending order by the increased salary.

```
EXEC SQL  DECLARE INCREASE CURSOR FOR
            SELECT EMPNO, LASTNAME, SALARY * 1.1
              FROM EMPLOYEE
              WHERE JOB = 'CLERK'
              ORDER BY 3 DESC;
```

Example 3: In a PL/I program, declare a cursor named UP_CUR to update all the columns of the DEPARTMENT table.

```
EXEC SQL  DECLARE UP_CUR CURSOR FOR
            SELECT *
              FROM DEPARTMENT
              FOR UPDATE OF DEPTNO, DEPTNAME, MGRNO, ADMRDEPT;
```

Example 4: In a PL/I program, declare a cursor named DEL_CUR to examine, and potentially delete, rows in the DEPARTMENT table.

```
EXEC SQL  DECLARE DEL_CUR CURSOR FOR
            SELECT *
              FROM DEPARTMENT;
```

DECLARE CURSOR for INSERT

insert-statement

This is an INSERT using VALUES statement as defined with the INSERT statement. The *insert-statement* must not include parameter markers, but can include references to host variables. In host languages, other than assembler and REXX, the declarations of the host variables must precede the DECLARE CURSOR statement in the source program. In assembler host variable declarations can follow the DECLARE CURSOR statement. In REXX, host variables are not declared at all.

Once a cursor has been defined and opened, you may insert new rows into the table using the PUT statement.

Example 5: This example shows portions of a pseudo COBOL program. In this program, use the cursor C2 to insert a row into the DEPARTMENT table based on the values in the host variables DPT_NO (char(3), DPT_NM (varchar(29)), MGR_NO (char(6)), and DPT_AD (char(3)).

```
* in working storage:
    EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
      77 DPT-NO           PIC X(3).
      77 MGR-NO           PIC X(6).
      77 DPT-AD           PIC X(3).
      01 DPT-NM.
         49 DPT-NM-LEN    PIC S9(4) COMP  VALUE +29.
         49 DPT-NM-VAL    PIC X(29)       VALUE SPACES.
    EXEC SQL  END DECLARE SECTION  END-EXEC.

* at start of processing:
    EXEC SQL  DECLARE C2 CURSOR FOR
                INSERT INTO DEPARTMENT
                  VALUES (:DPT-NO, :DPT-NM, :MGR-NO, :DPT-AD)  END-EXEC.
    EXEC SQL  OPEN C2  END-EXEC.

* loop as many times as necessary:
*   solicit values from screen and assign to DPT-NO, DPT-NM, MGR-NO, DPT-AD
    EXEC SQL  PUT C2  END-EXEC.

* at end of processing
    EXEC SQL  CLOSE C2  END-EXEC.
```

DECLARE CURSOR for Dynamic Queries

statement_name
Identifies a SELECT or INSERT statement defined in a PREPARE statement. (When communicating with an application server that is not DB2 Server for VM or DB2 Server for VSE, this restriction might not be enforced.) The DECLARE CURSOR statement and its associated PREPARE statement must be in the same logical unit of work. They may be specified in either order, except in Fortran programs when the *string-constant* form of the PREPARE statement is used. For information on this restriction, see “PREPARE” on page 313.

Example 6

This example is similar to Example 1 under DECLARE CURSOR for SELECT. The difference is that the right hand side of the WHERE clause is to be specified dynamically; thus the entire select-statement is placed into a host variable and dynamically prepared.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  EMP              CHAR(6);
  DCL  PRJ              CHAR(6);
  DCL  SEARCH_PRJ       CHAR(6);
  DCL  ACT              BINARY     FIXED(15);
  DCL  TIM              DEC        FIXED(5,2);
  DCL  SELECT_STMT      CHAR(200)  VARYING;
EXEC SQL  END DECLARE SECTION;


SELECT_STMT = 'SELECT EMPNO, PROJNO, ACTNO, EMPTIME ' ||
                'FROM EMP_ACT ' ||
                'WHERE PROJNO = ?';
   .
   .
   .
 EXEC SQL  PREPARE SELECT_PRJ FROM :SELECT_STMT;

 EXEC SQL  DECLARE C1 CURSOR FORSELECT_PRJ;

 EXEC SQL  OPEN C1 USING :SEARCH_PRJ;

 EXEC SQL  FETCH C1 INTO :EMP, :PRJT, :ACT, :TIM;

 IF SQLSTATE = '02000' THEN
   CALL DATA_NOT_FOUND;
 ELSE
   DO WHILE (SUBSTR(SQLSTATE,1,2) = '00' | SUBSTR(SQLSTATE,1,2) = '01');
     EXEC SQL  FETCH C1 INTO :EMP, :PRJ, :ACT, :TIM;
   END;

 EXEC SQL  CLOSE C1;
   .
   .
   .
```

DECLARE CURSOR WITH RETURN

WITH RETURN
Specifies that the cursor, if declared in a stored procedure, can return a result set to a caller.

Example 7: The following statements could be included in a stored procedure. If the cursors are opened and not closed, the result sets are returned to the requester.

```
EXEC SQL DECLARE CURS1 CURSOR WITH RETURN FOR
  SELECT A.X,Y,Z FROM TABLEX A, TABLEY B WHERE A.X = B.X

EXEC SQL DECLARE CURS2 CURSOR WITH RETURN FOR STMT1
```

Overall Notes

The scope of *cursor_name* is the source program in which it is defined; that is, the program submitted to the preprocessor. Thus, you can only reference a cursor by statements that are preprocessed with the cursor declaration. For example, a program called from another separately preprocessed program cannot use a cursor that was opened by the calling program.

The NOFOR Option

The NOFOR preprocessor option concerns the use of the UPDATE clause when a cursor is declared for a static (embedded) query. With NOFOR in effect, this clause is optional. When the clause is used, updates are restricted to the columns designated within it. NOFOR is only useful when the UPDATE statements are static. See the *DB2 Server for VSE & VM Application Programming* manual for more details on the NOFOR preprocessor option.

Extended DECLARE CURSOR

The Extended DECLARE CURSOR statement defines the cursor through which a user may OPEN, FETCH, PUT, or CLOSE the results of a statement prepared using Extended PREPARE. There are two types of cursor:

- A *query cursor* is a cursor associated with a select-statement and used by an application to access rows in a result table.
- An *insert cursor* is a cursor associated with an insert-statement and used by an application to insert rows into an active set.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──DECLARE──cursor_variable──CURSOR FOR──section_variable──IN──package_spec──────►◄
```

Description

cursor_variable
Provides a name for the cursor. The name placed into *cursor_variable* must be unique within the logical unit of work in which it is used.

CURSOR FOR *section_variable*
Identifies a select-statement or insert-statement defined in an Extended PREPARE statement. A cursor need not be declared in the same logical unit of work or program in which the statement was prepared.

IN *package_spec*
Identifies the package in which the referenced SQL statement resides. The *package_spec* must identify a package that exists at the application server.

Notes

Cursors are associated with a prepared select-statement or insert-statement by the value returned in the *section_variable* and the *package_spec* specified in the Extended DECLARE CURSOR statement. Extended DECLARE CURSOR may be used for any select-statement or insert-statement in a package created using the CREATE PACKAGE statement.

A cursor name used in a WHERE CURRENT OF clause of a DELETE statement or an UPDATE statement cannot be specified from a host variable. Therefore, at execution time, the content of cursor_name in the Extended DECLARE CURSOR statement must be the same as the cursor_name hard-coded in the WHERE CURRENT OF clause.

After the Extended DECLARE CURSOR statement is entered, a cursor is established; the cursor can then be opened and used to retrieve or insert rows through the Extended OPEN, FETCH, and PUT statements.

Examples

```
DECLARE :CURSOR1 CURSOR FOR :STMID IN :USERID.:PACKNAME
```

DELETE

The DELETE statement deletes rows from a table or view. Deleting a row from a view deletes the row from the table on which the view is based.

There are two forms of this statement:

- The *Searched* DELETE form deletes one or more rows (optionally determined by a search condition).
- The *Positioned* DELETE form deletes exactly one row (as determined by the current position of a cursor).

Invocation

A Searched DELETE statement can be embedded in an application program or issued interactively. A Positioned DELETE must be embedded in an application program. Both Searched DELETE and Positioned DELETE are executable statements that can be dynamically prepared.

A Positioned DELETE in Fortran, and programs prepared using extended dynamic SQL cannot be used with the DRDA protocol.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- Ownership of the table
- The DELETE privilege for the table or view
- DBA authority.

The DELETE privilege on a view is only inherent in DBA authority. Ownership of a view does not necessarily include the DELETE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

If the *search-condition* includes a subquery, the privileges designated by the authorization ID of the statement must also include the SELECT privilege on every table or view identified in the subquery. The privilege may have been explicitly granted or may be inherent in another privilege. The SELECT privilege on a table or view is inherent in DBA authority and ownership of a table or view.

Syntax

Searched delete (I,P)

```
►►──DELETE FROM──┬─table_name─┬──┬──────────────────┬──────────────────────────►◄
                 └─view_name──┘  └─correlation_name─┘

►►──┬─────────────────────────┬──┬─────────────────┬────────────────────────────►◄
    └─WHERE──search_condition─┘  └─WITH──┬─RR─┬─┘
                                         └─CS─┘
```

Positioned delete (P)

```
                                                    (1)
►►──DELETE FROM──┬─table_name─┬──WHERE CURRENT OF────────cursor_name────────────►◄
                 └─view_name──┘
```

Notes:

1 A Positioned DELETE in Fortran, and programs prepared using Extended dynamic SQL cannot be used with DRDA protocol.

Description

FROM *table_name* or *view_name*
Identifies the table or view from which rows are to be deleted. The name must identify a table or view that exists at the application server, but must not identify a catalog table, a view of a catalog table, or a read-only view. (For an explanation of read-only views, see "CREATE VIEW" on page 231.)

Note: Someone with DBA authority may delete rows from a few of the catalog tables. See "Updateable Columns" on page 371.

correlation_name
Can be used within the *search_condition* to designate the table or view. (For an explanation of *correlation_name*, see Chapter 3.)

WHERE
Specifies the rows to be deleted. You can omit the clause, give a search condition, or name a cursor. If you omit the clause, all rows of the table or view are deleted.

search_condition
Is any search condition as described in Chapter 3. Each *column_name* in the search condition, other than in a subquery, must name a column of the table or view.

The *search_condition* is applied to each row of the table or view and the deleted rows are those for which the result of the *search_condition* is true.

If the search condition contains a subquery, the subquery can be thought of as being processed each time the *search condition* is applied to a row, and the results used in applying the *search condition*. In actuality, a subquery with no correlated references is processed once, whereas a subquery with a correlated reference may have to be processed once for each row.

The following restriction is enforced when a DELETE statement is prepared or preprocessed with a WHERE clause containing a subquery. Let T2 denote the object table of a DELETE statement, and let T1 denote a table that is referenced in the FROM clause of a subquery of that statement, T1 must not be a table that can be affected by the DELETE on T2. The following example demonstrates the relationships.

```
DELETE FROM T2 WHERE FIELD2 IN (SELECT FIELD1 FROM T1);
```

The following rules apply to the above situation:

- T1 and T2 must not be the same table.
- T1 must not be a dependent of T2 in a relationship with a delete rule of CASCADE or SET NULL.
- T1 must not be a dependent of another table T3 in a relationship with a delete rule of CASCADE or SET NULL if deletes of T2 cascade to T3.

WITH
Specifies the isolation level used when locating the rows to be deleted by the statement.

RR
Repeatable read

CS
Cursor stability

The default isolation level of the statement is the isolation level of the package. WITH can only be specified on a SEARCHED delete; it is incompatible with the WHERE CURRENT OF clause.

CURRENT OF *cursor_name*
Identifies the cursor to be used in the delete operation. The *cursor_name* must identify a declared cursor as explained in “DECLARE CURSOR” on page 235. The *cursor_name* can be a delimited identifier. If *cursor_name* is a reserved word, it must be a delimited identifier.

The table or view specified must also be specified in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see “DECLARE CURSOR” on page 235.)

When the DELETE statement is processed, the cursor must be positioned on a row; that row is the one deleted. The cursor goes into a *between* state in which it remains open but has no current row until you reposition it with a FETCH statement. You cannot use the cursor for further deletions or updates while it is in the between state.

To maintain data integrity between tables when data is deleted from a parent table, the database manager checks that delete rules are followed. The delete rule in a referential constraint clause defines what action should be taken by the system when a parent row is deleted. The delete rules are:

- The RESTRICT rule prevents the deletion of a parent row unless all the dependent rows have been deleted first. This is the default rule.
- The CASCADE rule tells the database manager to delete the descendent rows as well as the parent row. Multi-level cascade is supported, subject to the restrictions described in “Definition Restrictions” on page 16.

- The SET NULL rule tells the database manager to set all nullable columns of the foreign key to null before deleting the parent row. At least one column of the foreign key must be nullable.

Notes

If an error occurs during the execution of any delete operation, no rows are deleted. If an error occurs during the execution of a Positioned DELETE, the position of the cursor is unchanged. However, it is possible for an error to make the position of the cursor incorrect, in which case the cursor is closed. It is also possible for a delete operation to cause a rollback, in which case the cursor is closed.

If an error occurs during the execution of a Searched DELETE, it is necessary to inspect SQLWARN6 to determine the extent of the failure. The following are current settings of SQLWARN6 along with possible responses:

1. SQLWARN6 is set to 'S'. A severe error has occurred, leaving the system in an unusable state.
   - No further requests are possible. The application must end, or, in a DB2 Server for VSE & VM environment, may switch to another database
2. SQLWARN6 is set to 'W'. An error occurred causing the LUW to be rolled back automatically. The system is still in a usable state. The application can either:
   - begin a new LUW and proceed **or**
   - end.
3. SQLWARN6 is blank. An error has occurred, but the LUW is still active. For recoverable storage pools any changes made by the request have been rolled back, hence the failing request has not left any partial results in the database. For more information about recoverable storage pools, see the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual.

   The application can do one of the following:
   - Continue forward processing of the LUW
   - Commit the changes made before the failing request
   - Roll back the LUW.

Unless appropriate locks already exist, one or more exclusive locks are acquired by executing a successful DELETE statement. Until the locks are released, they can prevent other application processes from performing operations on the table. For further information about locking, see the description of the COMMIT WORK, ROLLBACK WORK, LOCK TABLE, and LOCK DBSPACE statements. The isolation level associated with the application process defines the degree to which rows deleted by one process are visible to other concurrent processes.

If an application process deletes a row on which any of its cursors are positioned, those cursors are positioned before the next row of their result table. Let C be a cursor that is positioned before row R (as a result of an OPEN, a DELETE through C, a DELETE through some other cursor, or a searched DELETE). In the presence of INSERT, UPDATE, and DELETE operations that affect the base table from which R is derived, the next FETCH operation referencing C does not necessarily position C on R. For example, the operation can position C on R’, where R’ is a new row that is now the next row of the result table.

When a DELETE statement is completed, the number of rows deleted is returned in SQLERRD(3) in the SQLCA. The value in SQLERRD(3) does not include the number of rows that were deleted as a result of a CASCADE delete rule.

SQLERRD(5) in the SQLCA shows the number of rows affected by referential constraints. It includes rows that were deleted as a result of a CASCADE delete rule and rows in which foreign keys were set to NULL as the result of a SET NULL delete rule.

If you preprocess your program with the BLOCK option, and you wish to process a Positioned DELETE dynamically, the cursor must be a SELECT...FOR UPDATE statement, even if you do not plan to process any updates with the cursor. The FOR UPDATE clause is needed to tell the database manager that blocking should be overridden when the SELECT statement is prepared. If you do not use the FOR UPDATE clause in this instance, an error will occur on your DELETE statement at execution time.

Examples

Example 1

Delete department (DEPTNO) ‘D11’ from the DEPARTMENT table.

```
DELETE FROM DEPARTMENT
  WHERE DEPTNO = 'D11'
```

Example 2

Delete all the departments from the DEPARTMENT table (that is, empty the table).

```
DELETE FROM DEPARTMENT
```

Example 3

Use a PL/I program statement to delete all the subprojects (MAJPROJ is NULL) from the PROJECT table for a department (DEPTNO) equal to that in the host variable HOSTDEPT (char(6)).

```
EXEC SQL DELETE FROM PROJECT
           WHERE DEPTNO = :HOSTDEPT AND MAJPROJ IS NULL;
```

Example 4

Code a portion of a PL/I program that will be used to display retired employees (JOB) and then, if requested to do so, remove certain employees from the EMPLOYEE table.

```
EXEC SQL  DECLARE C1 CURSOR FOR
            SELECT *
              FROM EMPLOYEE
              WHERE JOB = 'RETIRED';

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO ...     ;

PUT ...      ;
GET LIST (REMOVE);
IF REMOVE = 'YES' THEN
  EXEC SQL  DELETE FROM EMPLOYEE
              WHERE CURRENT OF C1;

EXEC SQL  CLOSE C1;
```

DESCRIBE

The DESCRIBE statement obtains information about a prepared statement. It is primarily used for describing a SELECT statement. For an explanation of prepared statements, see “PREPARE” on page 313.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required. See “PREPARE” on page 313 for the authorization required to create a prepared statement.

Syntax

```
►►──DESCRIBE──statement_name──INTO──descriptor_name──┬──────────────────┬──────►◄
                                                     │        ┌─NAMES──┐ │
                                                     └─USING──┼─ANY────┼─┘
                                                              ├─BOTH───┤
                                                              └─LABELS─┘
```

Description

statement_name
Identifies the statement about which information is to be obtained. When the DESCRIBE statement is processed, the name must identify a statement dynamically prepared in the same logical unit of work.

INTO *descriptor_name*
Identifies an SQL descriptor area (SQLDA). Before the DESCRIBE statement is processed, the following variable in the SQLDA must be set:

SQLN	Indicates the number of variables represented by SQLVAR. (SQLN acts as a dimension of the SQLVAR array.) SQLN must be set to a value greater than or equal to zero before the DESCRIBE statement is processed. When the USING clause is set to NAMES, LABELS, or ANY, this should specify the maximum number of expected select list items. When the USING clause is set to BOTH, twice the expected number of select list items should be specified.

When the DESCRIBE statement is processed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID	This field serves only as an SQLDA eye-catcher. It is set to 'SQLDA' by the database manager when a DESCRIBE is first processed.
SQLDABC	16 + SQLN*44 (the length of the SQLDA).
SQLD	For a SELECT statement, the number of columns described by occurrences of SQLVAR (or, if USING BOTH was specified on DESCRIBE, twice the number of columns). For a non-SELECT statement, 0.
SQLVAR	This is an array with an arbitrary number of occurrences of the

five variables listed below. If the value of SQLD is 0, or greater than the value of SQLN, no values are assigned to occurrences of SQLVAR.

If the value of SQLD is *n*, where *n* is greater than 0 but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR so that the first occurrence of SQLVAR contains a description of the first column of the result table, the second occurrence of SQLVAR contains a description of the second column of the result table, and so on.

In cases where the USING clause is set to BOTH, the database manager returns twice as many SQLVAR entries as there are columns in the select list. Given that there are *n* columns, the first *n* SQLVAR entries are for column names and the second *n* entries are for column labels.

SQLTYPE A code showing the data type of the column and whether it can contain null values. For information about the SQLTYPE codes returned following the execution of a DESCRIBE statement, see Table 22 on page 362.

SQLLEN A length value depending on the data type of the result columns. For the possible values of SQLLEN, see Table 22 on page 362.

SQLDATA Contains the CCSID of a string column, as shown in Table 23 on page 363.

SQLIND Indicates the subtype of a character column, if using the SQLDS protocol. Does not provide any information if using the DRDA protocol. For values, see Table 21 on page 360.

SQLNAME Contains the name or label associated with the column used in the select list of the DESCRIBE statement. Exceptions to this are select list items that are unnamed, such as built-in functions (SUM(SALARIES)), constants (’ABC’), and expressions (A+B+C). In these cases, position 1 of SQLNAME is blank (X'40'), and positions 3 through 30 contain a description of the unnamed field. Because a blank is not allowed in the first byte of SQL identifiers, the application program can tell whether a column name is returned.

If no column name is returned, the following rules govern the content and format of the SQLNAME field.

If the select list item involves:
- *A basic function*: SQLNAME contains the name of the function followed by the column name in parentheses (for example, SUM(SALARIES)). Position 2 of SQLNAME is blank.
- *A DISTINCT object of a function*: SQLNAME contains the name of the function, followed

by the keyword DISTINCT and the name of the column in parentheses (for example, SUM(DISTINCT SALARIES)). If this entire description is too long to fit in positions 3 through 30 of SQLNAME, it is truncated, and position 2 is set to X'FF'.

- *An expression*: SQLNAME is set to the character string `EXPRESSION n`, where `n` is a number that identifies the `n`th expression in the select list. For example, for the sixth expression in the select list, the database manager sets positions 3 through n of SQLNAME to `EXPRESSION 6`. Position 2 is blank. This rule is true even for expressions that contain built-in functions, and, because expressions include constants, for constants such as 'ABC'.
- *A function whose object is an expression*: SQLNAME contains the name of the function followed by the character string `EXPRESSION n` in parentheses (for example, SUM(EXPRESSION 7)). Position 2 is blank.

USING
Indicates what value to assign to each SQLNAME variable in the SQLDA. If the requested value does not exist, SQLNAME is set to a length of 0.

NAMES
Assigns the name of the column. This is the default.

LABELS
Assigns the label of the column. (Column labels are defined by the LABEL ON statement.)

ANY
Assigns the column label, and if the column has no label, the column name.

BOTH
Assigns both the label and name of the column. In this case, two occurrences of SQLVAR per column are needed to accommodate the additional information. The first *n* occurrences of SQLVAR for each of the columns in the result table contain the column names. The second *n* occurrences contain the column labels.

Notes

Before the DESCRIBE statement is processed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA and enough storage must be allocated to contain SQLN occurrences. To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must not be less than the number of columns.

Allocating the SQLDA

Among the possible ways to allocate the SQLDA are the three described below.

First Technique: Allocate an SQLDA with enough occurrences of SQLVAR to accommodate any select list that the application will have to process. At the extreme, the number of SQLVARs could equal the maximum number of columns allowed in a result table. Having done the allocation, the application can use this SQLDA repeatedly.

This technique uses a large amount of storage that is never deallocated, even when most of this storage is not used for a particular select list.

Second Technique: Repeat the following two steps for every processed select list:

1. Process a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR; that is, an SQLDA for which SQLN is zero. The value returned for SQLD is equal to the required number of occurrences of SQLVAR.
2. Use the returned value of SQLD to allocate an SQLDA with enough occurrences of SQLVAR. Then process the DESCRIBE statement again, using this new SQLDA.

This technique allows better storage management than the first technique, but it doubles the number of DESCRIBE statements.

Third Technique: Allocate an SQLDA that is large enough to handle most, and perhaps all, select lists but is also reasonably small. If an execution of DESCRIBE fails because the SQLDA is too small, allocate a larger SQLDA and process DESCRIBE again. For the new SQLDA, use the value of SQLD returned from the first execution of DESCRIBE for the number of occurrences of SQLVAR.

This technique is a compromise between the first two techniques. Its effectiveness depends on a good choice of size for the original SQLDA.

Examples

In a PL/I program, process a DESCRIBE statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then process a DESCRIBE statement using that SQLDA.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  STMT1_STR   CHAR(200)  VARYING;
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

... /* code to prompt user for a query, then to generate */
    /* a select-statement in the STMT1_STR           */
EXEC SQL  PREPARE STMT1_NAME FROM :STMT1_STR;

... /* code to set SQLN to zero and to allocate the SQLDA */
EXEC SQL  DESCRIBE STMT1_NAME INTO :SQLDA;

... /* code to check that SQLD is greater than zero, to set */
    /* SQLN to SQLD, then to re-allocate the SQLDA          */
EXEC SQL  DESCRIBE STMT1_NAME INTO :SQLDA;

... /* code to prepare for the use of the SQLDA            */
EXEC SQL  OPEN DYN_CURSOR;

... /* loop to fetch rows from result table                */
EXEC SQL  FETCH DYN_CURSOR USING DESCRIPTOR :SQLDA;
.
.
.
```

Extended DESCRIBE

The Extended DESCRIBE statement obtains information about a select-statement prepared by an Extended PREPARE statement.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──DESCRIBE──section_variable──IN──package_spec──────────────────────────────►

►─INTO──descriptor_name──┬──────────────────────┬─────────────────────────────►◄
                         │            ┌─NAMES──┐ │
                         └─USING──────┼─ANY────┼─┘
                                      ├─BOTH───┤
                                      └─LABELS─┘
```

Description

section_variable
: Identifies a statement defined by an Extended PREPARE statement (see "Extended PREPARE" on page 317). The Extended DESCRIBE statement does not have to be in the same logical unit of work or program as the PREPARE statement that was originally used to process the statement.

IN *package_spec*
: Identifies the package in which the referenced SQL statement resides. The *package_spec* must identify a package that exists at the application server.

  The DESCRIBE option must have been specified on the CREATE PACKAGE statement that was used to create the package.

INTO *descriptor_name*
: Identifies an output SQLDA structure that is to receive information about the columns that are to be retrieved by the described SQL statement. This is identical to the descriptor used for the dynamic DESCRIBE statement.

USING
: This works the same as in the dynamic DESCRIBE statement, and follows the same rules. (See "DESCRIBE" on page 247 for more information). The labels returned in the SQLDA are those which were in the SYSCOLUMNS catalog table when the SQL statement was prepared.

Examples

```
DESCRIBE :STMID IN :USERID.:PACKNAME INTO MYSQLDA
```

DESCRIBE CURSOR

The DESCRIBE CURSOR statement obtains information about the result set that is associated with the cursor. The information, such as column information, is put into a descriptor. Use DESCRIBE CURSOR for result set cursors from stored procedures. The cursor must be defined with the ALLOCATE CURSOR statement.

Invocation

This statement can be embedded in an application program only. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

```
►►──DESCRIBE CURSOR──┬─cursor-name───┬──INTO──descriptor-name──────────────────►◄
                     └─host-variable─┘
```

Description

cursor-name **or** *host-variable*
: Identifies a name for the cursor. The name specified for *cursor-name* must be unique within the logical unit of work in which it is used. It is an ordinary identifier.

  If a *host-variable* is specified, the following rules apply:

  - It must be a character string variable with a length attribute that is not greater than 18 bytes (A C NULL-terminated character string may be up to 19 bytes).
  - It must be preceded by a colon and must not be followed by an indicator variable.
  - The cursor name must be left justified within the host variable and must not contain embedded blanks.
  - If the length of the cursor name is less that the length of the host variable, it must be padded on the right with blanks.

INTO *descriptor-name*
: Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the columns in the result set associated with the named cursor. The considerations for allocating and initializing the SQLDA are similar to those of a DESCRIBE statement used for describing a SELECT statement. After executing the DESCRIBE CURSOR statement, the contents of the SQLDA are the same as the DESCRIBE of a SELECT statement, with the following exceptions:

  - The first five bytes of the SQLDAID field are set to ’SQLRS’.
  - Bytes 6 to 8 of the SQLDAID field are reserved. If the cursor is declared WITH HOLD in a stored procedure, the high-order bit of the eighth byte is set to one.

    Note: DB2 Server for VSE & VM does not support CURSOR WITH HOLD. As a result, neither does its requester. If a cursor is opened WITH

HOLD by a stored procedure, it will be implicitly closed by the DB2 Server for VSE & VM requester when the unit of work is committed.

Notes

1. For the DESCRIBE CURSOR statement to be successful, the application must be connected to the site at which the stored procedure was executed.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1

Place information about the result set associated with cursor C1 into the descriptor named by :sqlda1:

```
EXEC SQL DESCRIBE CURSOR C1 INTO :sqlda1
```

Example 2

Place information about the result set associated with the cursor named by :hv1 into the descriptor named by :sqlda2:

```
EXEC SQL DESCRIBE CURSOR :hv1 INTO :sqlda2
```

DESCRIBE PROCEDURE

The DESCRIBE PROCEDURE statement obtains information about the result sets returned by a stored procedure. The information, such as the number of result sets, is put into a descriptor.

Invocation

This statement can be embedded in an application program only. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

```
►►──DESCRIBE PROCEDURE──┬─host-variable──┬──INTO──descriptor-name───────────────►◄
                        └─procedure-name─┘
```

Description

host-variable **or** *procedure-name*
: Identifies the stored procedure to describe. The procedure name may be specified either directly or within a host-variable.

  If a *host-variable* is specified, it must be a character-string variable and it must not include an indicator variable. Note that the value is not converted to uppercase. Procedure name must be left-justified.

  If *procedure-name* is specified, it must be an ordinary identifier, which implies that it cannot contain blanks or special characters, and the value is converted to uppercase. Therefore, if it is necessary to use a lowercase name that contains blanks or special characters, then the name must be specified in a host variable. The form in which a procedure name exists varies according to the server where the procedure is stored.

  DB2 Server for VSE & VM:
  The name of the procedure to execute. The name can be up to 18 characters long and must match a value in the NAME column of the SYSTEM.SYSROUTINES catalog table.

  DB2 Common Server/UDB:

  procedure-name
  The name (with no extension) of the procedure to execute. This is used both as the name of the stored procedure library and the function name within that library.

  procedure-library!function-name
  The exclamation point character acts as a delimiter between the library name and the function name of the stored procedure.

  absolute-path!function-name
  The absolute-path specifies the complete path to the stored procedure library.

  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.

DB2 for MVS V4 or DB2 for OS/390 V5 Server:
An implicit or explicit three-part name. The parts are as follows:

high order
The location name of the server where the procedure is stored.

middle
SYSPROC

low order
Some value in the PROCEDURE column of the SYSIBM.SYSPROCEDURES catalog table.

DB2 for OS/400 (V3.1 or later) Server:
The external program name is assumed to be the same as the procedure-name. For portability, the procedure-name should be specified as a single token no larger than eight bytes. The ASSOCIATE LOCATORS statement can only be executed against a stored procedure that has already been invoked by the program using the SQL CALL statement.

INTO *descriptor-name*
Identifies an SQL descriptor area (SQLDA). The information returned in the SQLDA describes the result sets returned by the stored procedure. Before the DESCRIBE PROCEDURE statement is processed, the following variable in the SQLDA must be set:

SQLN Indicates the number of variables represented by SQLVAR. (SQLN acts as a dimension of the SQLVAR array.) SQLN must be set to a value greater than or equal to zero before the DESCRIBE PROCEDURE statement is processed. This value should reflect the expected number of result sets the stored procedure is to return.

When the DESCRIBE PROCEDURE statement is processed, the database manager assigns values to the variables of the SQLDA as follows:

SQLDAID
This field serves only as an SQLDA eye-catcher. It is set to 'SQLPR'.

SQLD This field is set to the total number of result sets. A value of zero in the field indicates there are no result sets.

SQLVAR
This is an array with an arbitrary number of occurrences of the variables listed below, and others that are not mentioned. There is one SQLVAR entry for each result set. If the value of SQLD is zero, or greater than the value of SQLN, no values are assigned to the occurrences of SQLVAR. If the value of SQLD is *n*, where *n* is greater than zero but less than or equal to the value of SQLN, values are assigned to the first *n* occurrences of SQLVAR. Therefore, the first occurrence of SQLVAR contains a description of the first result set, the second occurrence of SQLVAR contains a description of the second result set, and so on.

SQLDATA
This field of each SQLVAR entry is set to the result set locator value associated with the result set.

SQLIND
This field of each SQLVAR entry is set to the estimated number of rows in the result set.

SQLNAME
This field is set to the name of the cursor used by the stored procedure to return the result set.

Notes

1. A value of –1 in the SQLIND field indicates that an estimated number of rows in the result set is not provided.
2. DESCRIBE PROCEDURE does not return information about the parameters expected by the stored procedure.

Examples

The statements in the following examples are assumed to be in PL/I programs.

Example 1

Place information about the result sets returned by stored procedure P1 into the descriptor named by :sqlda1:

```
EXEC SQL DESCRIBE PROCEDURE P1 INTO :sqlda1
```

Example 2

Place information about the result sets returned by stored procedure named by :hv1 into the descriptor named by :sqlda2:

```
EXEC SQL DESCRIBE PROCEDURE :hv1 INTO :sqlda2
```

DROP

The DROP statement deletes an object. Any objects that are directly or indirectly dependent on that object are also deleted. Whenever an object is deleted, its description is deleted from the catalog and any packages that reference the object are invalidated.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table, view, index, synonym, dbspace or package.
- DBA authority.

Syntax

```
►►──DROP──┬─DBSPACE──dbspace_name──────┬───────────────────────────────────►◄
          ├─INDEX──index_name──────────┤
          │         (1)                │
          ├─PACKAGE────────package_spec┤
          ├─SYNONYM──synonym───────────┤
          ├─TABLE──table_name──────────┤
          └─VIEW──view_name────────────┘
```

Notes:

1 PROGRAM is equivalent to PACKAGE and is provided for compatibility with older versions of SQL/DS.

Description

DBSPACE *dbspace_name*
Identifies the dbspace to be dropped. It must be a dbspace that exists at the application server. Dropping a dbspace destroys the contents of a dbspace. When the logical unit of work is committed, the dbspace is available to be acquired. All existing packages with dependencies on tables within the dropped dbspace are automatically marked unusable. Both private and public dbspaces can be dropped, but only someone with DBA authority can drop a public dbspace. No user, even with DBA authority, can drop the dbspace containing the database manager catalogs.

INDEX *index_name*
Identifies the index to be dropped. It must be an index that exists at the application server. The table on which the index is defined is not affected. All existing packages that use the dropped index are marked unusable.

An index created by a primary key cannot be dropped.

PACKAGE *package_spec*
Identifies the package to be dropped. It must be a package that exists at the application server. Once a package is dropped, the program that uses that package cannot be run. An owner can only drop packages which that owner has preprocessed. Only someone with DBA authority can drop another user's package.

DROP PACKAGE cannot support a qualified host structure subfield name in the *package_spec*. A host structure subfield name may be used here as a normal *host_variable* but must be unqualified. If being unqualified results in an ambiguous reference, the subfield identifier name cannot be used with DROP PACKAGE.

If the package was created using a host identifier which was not an ordinary identifier (such as a package name beginning with a number), it must be dropped using a host identifier; otherwise an SQL error will result. For example, if a package named 071PACK was created using a host identifier and a

```
DROP PACKAGE 071PACK
```

statement is issued, an SQLCODE of -105 (SQLSTATE of 37501) will result.

SYNONYM *synonym*
Identifies a synonym to be dropped. In a static DROP SYNONYM statement, the name must identify a synonym that is owned by the owner of the package. In a dynamic DROP SYNONYM statement, the name must identify a synonym that is owned by the authorization ID that is executing the statement.

Dropping a synonym has no effect on the table or view that it references. Dropping a synonym does not affect the packages of existing programs that use the synonym, because in the packages the synonym has already been resolved to a real table name. However, a program containing a dropped synonym cannot be preprocessed successfully, either automatically or by user request.

TABLE *table_name*
Identifies a table to be dropped. It must be a base table that exists at the application server and cannot be a catalog table. The table is deleted from the database and the contents of the table are lost. All indexes, keys, constraints, and views defined on the table, and all privileges granted on the table, are also dropped. Synonyms are not dropped. No user, even with DBA authority, can drop a table which forms part of the database manager system catalog.

All existing packages affected by dropping the table are marked unusable. The unusable packages remain in the database until they are explicitly dropped by a DROP PACKAGE statement. When an SQL statement attempts to invoke an unusable package, the database manager tries to dynamically rebind the package. However, if the SQL statement refers to a dropped DBSPACE or table, that SQL statement returns an error code at execution time.

VIEW *view_name*
Identifies the view to be dropped. It must be a view that exists at the application server. The definition of the view is deleted from the catalog. The definition of any view that is directly or indirectly dependent on that view is also deleted. Whenever the definition of a view is deleted from the catalog, all privileges on that view are also deleted.

All existing packages that use the dropped view are marked unusable.

Notes

If a DROP statement is issued for an object while some program that depends on the object is running and has a logical unit of work in progress, the DROP statement does not take effect until the end of the running logical unit of work. Meanwhile, the program that has issued the DROP waits.

When dropping a table, the database manager temporarily requires additional space so it can restore the table in case the logical unit of work is not committed. The database manager behaves as though a table approximately doubles in size immediately before it is dropped. The empty pages are taken from the DBSPACE from which the table was dropped. If the number of empty pages is less than approximately double the table size, the database manager will stop processing and will not issue a ROLLBACK. Note that if all rows of a table have previously been deleted, such additional space is not required.

Examples

Example 1

Drop your table named MY_IN_TRAY.

```
DROP TABLE MY_IN_TRAY
```

Example 2

Drop your view named MA_PROJ.

```
DROP VIEW MA_PROJ
```

Example 3

Drop the package named PACKA.

```
DROP PACKAGE PACKA
```

Example 4

Drop the dbspace named MYSPACE that is owned by MIKE. (Note that the authorization id submitting this statement must have DBA authority.)

```
DROP DBSPACE MIKE.MYSPACE
```

DROP PROCEDURE

The DROP PROCEDURE statement removes the definition of a stored procedure from the database manager, and takes the information for that procedure out of the cache.

The STOP PROC command must be issued with the REJECT option before the DROP PROCEDURE statement will be accepted.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer of the DROP PROCEDURE statement must have DBA authority.

Syntax

```
►►──DROP PROCEDURE──procedure-name──┬────────────────┬──┬──────────┬──────────────►◄
                                    └─AUTHID──authid─┘  └─RESTRICT─┘
```

Description

procedure-name
: must identify a stored procedure that has been defined (that is, a CREATE PROCEDURE has been processed for it).

  Note that DROP PROCEDURE removes the definition of the procedure only; the package associated with the procedure, as well as the load module or phase, is untouched.

authid
: The authorization ID for the stored procedure. If specified, then only the version of *procedure-name* that is accessible only by *authid* will be dropped.

RESTRICT
: This is included for compatibility with the DB2 family. If specified, it is ignored.

Examples

Example 1

```
DROP PROCEDURE MYPROC
```

DROP PSERVER

The DROP PSERVER statement removes the definition of a stored procedure server from the database manager, and takes the information for that server out of the cache.

The STOP PSERVER command must be issued with the NOIMPLICIT option before the DROP PSERVER statement will be accepted.

A stored procedure server cannot be dropped if the following are all true:
- The stored procedure server is the only one in its group
- Stored procedures exist that must run in this stored procedure server's group.

Note: If the drop fails for this reason, issue the ALTER PROCEDURE statement and use the SERVER GROUP clause to indicate that the procedure is to be moved to a different group, then issue the DROP PSERVER statement again.

Invocation

This statement can be issued from an application program or interactively. It is an executable statement that can be dynamically prepared.

Authorization

The issuer of the DROP PSERVER statement must have DBA authority.

Syntax

```
►►──DROP PSERVER──procedure-server─────────────────────────────────►◄
```

Description

procedure-server
The name of the stored procedure server. This name must be an ordinary identifier of 1 to 8 characters.

Examples

Example 1

```
DROP PSERVER SRV1
```

DROP STATEMENT

The DROP STATEMENT statement selectively deletes a statement from a package. DROP STATEMENT applies only to packages created with a CREATE PACKAGE statement with the MODIFY option.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──DROP STATEMENT──section_variable──IN──package_spec──────────────────────►◄
```

Description

section_variable
  Identifies the statement defined by an Extended PREPARE statement.

IN *package_spec*
  Identifies the package in which the referenced SQL statement resides. The *package_spec* must identify a package that exists at the application server.

Notes

When a statement references an incorrect package, dynamic re-preprocessing will occur to restore the package to a usable state. If the package has any unresolved dependencies, the re-processing will fail and a message will be issued.

Examples

```
DROP STATEMENT :STMID IN :USERID.:PACKNAME
```

END DECLARE SECTION

The END DECLARE SECTION statement marks the end of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It is not supported in REXX.

Authorization

None required.

Syntax

```
►►──END DECLARE SECTION──────────────────────────────────────────────────────►◄
```

Description

See “BEGIN DECLARE SECTION” on page 169 for a description of the END DECLARE SECTION statement.

Examples

See “BEGIN DECLARE SECTION” on page 169 for examples using the END DECLARE SECTION statement.

EXECUTE

The EXECUTE statement processes a prepared SQL statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See “PREPARE” on page 313 for the authorization required to create a prepared statement.

Syntax

```
►►──EXECUTE──statement_name──┬─────────────────────────────────────┬──────────►◄
                             ├─USING───host_variable_list──────────┤
                             └─USING DESCRIPTOR──descriptor_name───┘
```

Description

statement_name
Is an ordinary identifier that identifies the prepared statement to be processed. *Statement_name* must identify a statement that was previously prepared within the logical unit of work and the prepared statement must not be a SELECT statement.

USING
Introduces a list of host variables, host structures, or both, whose values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see “PREPARE” on page 313.) If the prepared statement includes parameter markers, the USING clause must be used. USING is ignored if there are no parameter markers.

host_variable_list
Identifies one or more host variable, host structure, or both that must be declared in the program in accordance with the rules for declaring host variables and host structures.

The total number of host variables and host structure subfields must be the same as the number of parameter markers in the prepared statement. The *n*th variable or subfield corresponds to the *n*th parameter marker in the prepared statement.

DESCRIPTOR *descriptor_name*
Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the statement was prepared.

Before the EXECUTE statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA
- SQLDABC to indicate the number of bytes of storage allocated for the SQLDA
- SQLD to indicate the number of variables used in the SQLDA when processing the statement

- SQLVAR occurrences to indicate the attributes of the variables and the addresses of the data areas allowed to contain the result.

The SQLDA must have enough storage to contain all SQLVAR occurrences. Therefore, the value in SQLDABC must be greater than or equal to 16 + SQLN*(44).

SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN. It must be the same as the number of parameter markers in the prepared statement. The *n*th variable described by the SQLDA corresponds to the nth parameter marker in the prepared statement. (For a description of an SQLDA, see “SQL Descriptor Area (SQLDA)” on page 359.)

Parameter Marker Replacement

Before the prepared statement is processed, each parameter marker in the statement is effectively replaced by its corresponding host variable or host structure subfield. The replacement is an assignment operation in which the source is the value of the host variable or host structure subfield and the target is a variable within the database manager. The assignment rules are those described for assignment to a column in “Assignments and Comparisons” on page 53. The attributes of the target variable depend on the role that the parameter marker plays in its SQL statement. The rules for the various roles are shown below. In those rules, “P” represents the parameter marker in question.

Arithmetic Operand: When P is an operand for an infix operator, the other operand cannot also be a parameter marker. The data type, scale, and precision of the target for P are the same as those of the other operand. When P is the operand of unary minus, the data type of the target is double precision floating point.

The Pattern in a LIKE Predicate: With P in this role, the target is a varying length string.
- If the first operand in the predicate is a short character string column, the target is a VARCHAR(n), where n is 10 more than the length attribute of the column, with this exception: if that length attribute is greater than 244, n is 254.
- If the first operand in the predicate is a long character string column, the target is VARCHAR(255).
- If the first operand is a short graphic string column, the target is VARGRAPHIC(n), where n is 5 more than the length attribute of the column, with the following exception: if that length attribute is greater than 122, n is 127.
- If the first operand in the predicate is a long graphic string column, the target is VARGRAPHIC(128).

Comparand: In this case, P can be a comparand in a basic predicate (for example, “?>10”), in an IN predicate, or in a BETWEEN predicate. At least one of the comparands in such a predicate must *not* be a parameter marker.

For a basic predicate, the other comparand cannot be a parameter marker.

When the parameter marker is specified as a comparison operand in the BETWEEN predicate,
- If there is an operand that is specified solely as a column name (or a column function with the argument being a column with a field procedure defined on it), then the attributes of the leftmost operand are used.
- Otherwise, the attributes of the leftmost operand that is not a parameter marker are used.

When the parameter marker is specified as a comparison operand in the IN predicate,

- The attributes of the leftmost operand that is not a parameter marker are used.

The attributes of the target for P are the same as those of the other comparand in the predicate, unless the data type of that comparand is DATE, TIME, or TIMESTAMP, in which case the target is effectively CHAR(254).

Assignment Operand: For this case, P must be the value for a column in an INSERT or UPDATE. The attributes of the target are the same as those of the column, with the following exceptions:

- If the column has the data type DATE, the target is CHAR(n), where n is the value of the LOCAL DATE LENGTH install option. If that option is not specified, n is 10.
- If the column has the data type TIME, the target is CHAR(n), where n is the value of the LOCAL TIME LENGTH install option. If that option is not specified, n is 8.
- If the column has the data type TIMESTAMP, the target is CHAR(26).

If the column has the data type DATE, TIME, or TIMESTAMP, trailing blanks are removed from the resulting string before assignment to the target. This is the one exception to the rule that the target is treated like a column.

General Rules: Let V denote a host variable that corresponds to a parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column:

- V must be compatible with the target.
- If V is a string, its length must not be greater than the length attribute of the target. (Trailing blanks are included in the length of the string.)

  The following is an exception to the rule:

  – If V is a fixed length host variable and the target is short varying-length column, all the trailing blanks of V, if any, are always truncated before assignment. Hence, if V’s length attribute is greater than the target’s length attribute and all the excess positions in V contain blanks, the assignment is completed without an error being returned.
- If V is a number, the absolute value of its integral part must not be greater than the maximum absolute value of the integral part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

When the prepared statement is processed, the value used in place of P is the value of the target variable V. For example, if V is CHAR(6) and the target is CHAR(8), the value used in place of P is the value of V padded on the right with two blanks.

Examples

This example of portions of a COBOL program shows how an INSERT statement with parameter markers is prepared and processed.

```
EXEC SQL  BEGIN DECLARE SECTION  END-EXEC.
  77 EMP             PIC X(6).
  01 PROJECT.
     05 PRJ              PIC X(6).
     05 ACT              PIC S9(4) COMP-4.
     05 TIM              PIC S9(3)V9(2).
     01 HOLDER.
        49  HOLDER-LENGTH    PIC S9(4) COMP-4.
        49  HOLDER-VALUE     PIC X(80).
   EXEC SQL  END DECLARE SECTION  END-EXEC.
     .
     .
     .
   MOVE 70 TO HOLDER-LENGTH.
   MOVE "INSERT INTO EMP_ACT (EMPNO, PROJNO, ACTNO, EMPTIME)
-              VALUES (?, ?, ?, ?)" TO HOLDER.
   EXEC SQL  PREPARE MYINSERT FROM :HOLDER  END-EXEC.

   IF SQLCODE = 0
     PERFORM DO-INSERT THRU END-DO-INSERT
   ELSE
     PERFORM ERROR-CONDITION.

   DO-INSERT.
     MOVE "000010" TO EMP.
     MOVE "AD3100" TO PRJ.
     MOVE 160      TO ACT.
     MOVE .50      TO TIM.
     EXEC SQL  EXECUTE MYINSERT USING :EMP, :PROJECT END-EXEC.
   END-DO-INSERT.
     .
     .
     .
```

Extended EXECUTE

The Extended EXECUTE statement processes an SQL statement that was prepared previously using an Extended PREPARE statement.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──EXECUTE──section_variable──IN──package_spec──────────────────────────►

►──┬──────────────────────────────────────┬──────────────────────────────►
   └─USING DESCRIPTOR──descriptor_name1───┘

►──┬─────────────────────────────────────────────┬───────────────────────►◄
   └─USING OUTPUT DESCRIPTOR──descriptor_name2───┘
```

Description

section_variable
: Identifies a statement defined by an Extended PREPARE statement. The Extended EXECUTE statement does not have to be in the same logical unit of work or program as the Extended PREPARE statement that was originally used to process the statement.

IN *package_spec*
: Identifies the package in which the referenced SQL statement resides. The *package_spec* must identify a package that exists at the application server.

USING DESCRIPTOR *descriptor_name1*
: Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the statement was prepared.

USING OUTPUT DESCRIPTOR *descriptor_name2*
: Identifies an output SQLDA structure that provides information about variables into which individual fields are to be returned by the query.

  This clause is only valid when using the EXECUTE statement against a section created by the PREPARE SINGLE ROW statement and in such cases the clause is required.

Before the Extended EXECUTE statement is processed, the user must set the fields in the SQLDA described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

Notes

When the statement is processed, the host variables specified in the SQLDA are substituted, in order, into the statement in place of the parameter markers (?) that were given in the Extended PREPARE statement. Each variable must be of a data type that is compatible with its usage in the "prepared" SQL statement. Extended EXECUTE will fail if the prepared statement was a select-statement (in this case, an Extended DECLARE CURSOR coupled with an Extended OPEN, FETCH, and CLOSE should be used).

Examples

```
EXECUTE :STMID IN :USERID.:PACKNAME
  USING DESCRIPTOR INSQLDA
  USING OUTPUT DESCRIPTOR OUTSQLDA
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement
- Processes the SQL statement
- Destroys the executable form.

EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. It may be used to prepare and process SQL statements that contain neither host variables nor parameter markers.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization rules are those defined for the SQL statement specified by EXECUTE IMMEDIATE. For example, see “INSERT Rules” on page 300 for the authorization rules that apply when an INSERT statement is processed using EXECUTE IMMEDIATE. The authorization ID is the run-time authorization ID.

Syntax

```
►►──EXECUTE IMMEDIATE──┬─string_constant─┬──────────────────────────────►◄
                       └─host_variable───┘
```

Description

string_constant
String constants are supported in all languages except Assembler and C.

It is advisable to avoid using either delimited identifiers or DBCS strings in statements specified in string constants.

host_variable
Identifies a host variable that must be described in the program in accordance with the rules for declaring host variables. An indicator variable must not be specified.

In Assembler, C, COBOL, REXX, the host variable must be a varying-length string variable. In C, it cannot be a NUL-terminated string. In Fortran, the host_variable must be a fixed-length string variable. In PL/I, the host variable can either be a fixed-length or varying-length string variable. The host variable must have a maximum length of 8192.

See “PREPARE” on page 313 for more information on the use of DBCS constants in prepared statements in PL/I Version 2 programs.

The *string_constant* or *host_variable* must contain one of the following SQL statements:

ACQUIRE DBSPACE
ALTER DBSPACE

ALTER PROCEDURE
ALTER PSERVER
ALTER TABLE
COMMENT ON
CREATE INDEX
CREATE PROCEDURE
CREATE PSERVER
CREATE SYNONYM
CREATE TABLE
CREATE VIEW
DELETE
DROP
DROP PROCEDURE
DROP PSERVER
EXPLAIN
GRANT Package Privileges
GRANT System Authorities
GRANT Table/View Privileges
INSERT
LABEL ON
LOCK DBSPACE
LOCK TABLE
REVOKE Package Privileges
REVOKE System Authorities
REVOKE Table/View Privileges
UPDATE
UPDATE STATISTICS

Furthermore, the statement string must not:
- Begin with EXEC SQL and end with a statement terminator
- Include references to host variables or parameter markers
- Include comments.

Notes

When an EXECUTE IMMEDIATE statement is processed, the specified statement string is parsed and checked for errors. If the SQL statement is incorrect it is not processed and the error condition that prevents its execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

If the same SQL statement is to be processed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Examples

Use PL/I program statements to move an SQL statement to the host variable QSTRING (char(80)) and prepare and process whatever SQL statement is in the host variable QSTRING.

```
IF ACCOUNTS = 'BIG' THEN
 QSTRING = 'INSERT INTO WORK_TABLE SELECT * FROM EMP_ACT WHERE    ACTNO <100';
ELSE
 QSTRING = 'INSERT INTO WORK_TABLE SELECT * FROM EMP_ACT WHERE    ACTNO >=100';
 .
 .
 .
EXEC SQL  EXECUTE IMMEDIATE :QSTRING;
```

EXPLAIN

The EXPLAIN statement places information about the structure and execution performance for a DELETE, INSERT, UPDATE, or SELECT statementinto one or more user-supplied tables.

The information applies to the statement for which the EXPLAIN was issued, and for any statements that have been generated internally by the database manager. Internal statements are generated to ensure referential integrity.

The result tables used by the EXPLAIN statement are updated during preprocessing of the containing program.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include both:
- Ownership of an explanation table for each of the specified options
- The proper privileges to process the SQL statement defined by the *explainable_sql_statement*.

Syntax

```
>>──EXPLAIN───┬─ALL──────────────────┬──┬─────────────────────────┬──────────────>
              │  ┌─,───────────────┐ │  └─SET QUERYNO ═──integer─┘
              │  ▼                 │ │
              └────┬─COST──────┬───┴─┘
                   ├─PLAN──────┤
                   ├─REFERENCE─┤
                   └─STRUCTURE─┘

>──FOR──explainable_sql_statement─────────────────────────────────────────────><
```

Description

COST
: Inserts into the COST_TABLE the complete cost of the command being analyzed and for any statements internally generated by the database manager to enforce referential integrity.

PLAN
: Inserts information into the PLAN_TABLE about the order in which tables are accessed during execution of the statement being analyzed and for any internally generated statements used to enforce referential integrity. Also describes the indexes used to access the tables, the methods that the database manageruses to do joins, and the sorts done as part of processing.

REFERENCE
: Inserts one row into the REFERENCE_TABLE for each column referenced in the statement and for any statements internally generated by the database managerto enforce referential integrity.

STRUCTURE
Inserts one row into the STRUCTURE_TABLE for each query block in the statement.

ALL
Inserts information into all four of the above tables.

SET QUERYNO=*integer*
An integer constant that can fit into an INTEGER field. The SET QUERYNO clause lets you place an integer value into the QUERYNO fields of the rows in the explanation tables. Assigning a different number on each EXPLAIN will make it easier to identify information collected. The *integer* value must not be preceded by a sign and may range from 1 to 2147483647.

The SET QUERYNO clause is optional. If you omit it, a null value is placed in the fields of the rows inserted by the EXPLAIN statement.

FOR *explainable_sql_statement*
The SQL statement to be analyzed. You can analyze UPDATE, DELETE, and INSERT statements as well as SELECT statements. (SELECT statements are considered the primary candidates for EXPLAIN analysis.) *explainable_sql_statement* is not a quoted-string and must not be put in a host variable. Host variables may not be placed in the statement; rather parameter markers must be used and the entire EXPLAINstatement must be dynamically prepared and processed.

The length of the SQL statement is limited to 8192 characters.

The database manager supplies customizable macros to build a set of EXPLAIN tables for each authorization ID that needs it. For IBM VM systems, the macro file is ARISEXP MACRO; for VSE systems, the macro is an A-type member, ARISEXP. Both macros contain comments describing the required customizing procedure.

| EXPLAIN may be invoked either explicitly as an SQL statement or implicitly with
| the EXPLAIN(YES) option for CREATE PACKAGE statement, application program
| preprocessing and the EXPLAIN(YES) option of the DBSU REBIND PACKAGE
| command. The following tables describe the columns required in each table
| associated with EXPLAIN. For more information about interpreting the data in
| these tables, see the *DB2 Server for VSE & VM Performance Tuning Handbook*,
| GC09-2987.

Table 10. Columns in COST_TABLE

Column Name	Data Type	Description
QUERYNO	INTEGER	Query number is intended to distinguish among queries. QUERYNO is set to the value specified in the SET QUERYNO clause. If the clause is omitted, QUERYNO is set to NULL. For an entry generated by the EXPLAIN(YES) option during program preprocessing, QUERYNO corresponds to the section number in the package for the statement being explained.

Table 10. Columns in COST_TABLE (continued)

Column Name	Data Type	Description
RINO	SMALLINT NOT NULL	RINO is set to zero for the user's original statement and will be automatically incremented by one for each internally-generated statement that is processed for referential integrity or cascade delete. RINO is intended to distinguish among queries and internally-generated queries. If RINO reaches 32,767, the next internally-generated statement will have a corresponding RINO value of 1, and so on.
QBLOCKNO	SMALLINT NOT NULL	Query block number, where 1 is the outer-level query block. Different query blocks (as occur in subqueries) receive different numbers.
PKGNAME	CHAR(8) NOT NULL	This identifies the name of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
PKGOWNER	CHAR(8) NOT NULL	This identifies the owner of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
COST	FLOAT NOT NULL	When QBLOCKNO is 1, this is a floating point number that represents the total estimated cost of executing the statement for which the EXPLAIN is issued and for any statement internally generated by the database managerto enforce referential integrity. For other values of QBLOCKNO, this is the cost of the subquery that has this query block as its root (as opposed to the cost of the query block alone). To find the cost of the query block alone, use information from the STRUCTURE_TABLE. The technique for doing this is described in the *DB2 Server for VSE & VM Database Administration* manual.
TIMESTAMP	TIMESTAMP NOT NULL	The time at which the EXPLAIN statement was processed.

Table 11. Columns in PLAN_TABLE

Column Name	Data Type	Description
QUERYNO	INTEGER	Query number is intended to distinguish among queries. (See COST_TABLE for a description of QUERYNO.)
RINO	SMALLINT NOT NULL	RINO is intended to distinguish among queries and internally-generated queries. (See COST_TABLE for a description of RINO.)
QBLOCKNO	SMALLINT NOT NULL	Query block number, where 1 is the outer level query block (which may have subqueries). Different query blocks receive different numbers. The plans for executing different query blocks do not refer to each other. However, STRUCTURE_TABLE provides the parent block for each query block, and indicates when the query block is done. This information is always implicitly part of the execution plan.
PKGNAME	CHAR(8) NOT NULL	This identifies the name of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.

Table 11. Columns in PLAN_TABLE (continued)

Column Name	Data Type	Description
PKGOWNER	CHAR(8) NOT NULL	This identifies the owner of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
PLANNO	SMALLINT NOT NULL	A number identifying the current step of the plan. PLANNO indicates the order in which the database managerdoes the actions of the plan for processing the query block. The PLAN_TABLE row with PLANNO 1 indicates the first action, PLANNO 2 indicates the second action, and so on. For each query block, each row entered as the result of an execution of EXPLAIN PLAN has a different PLANNO value.
METHOD	SMALLINT NOT NULL	METHOD is the action done at this step; it is either 0, 1, 2 or 3. Method is 0 only for the first table accessed (which has PLANNO 1). Because this is the first table, there is not yet a composite. Also, because there is no composite, SORTCOMP (described below) is blank for this row. Methods 1 and 2 correspond to plan steps that are joins, and identify the method by which the join is performed. Method 1 is the *nested loop* join. That is, for each row of the composite, the database managerfinds and joins matching rows of the new table. Method 2 is the *merge scan* join. In a merge scan join, the database managerscans the composite and the new table in order according to the join column. It then joins rows with matching join columns. This resembles processes used in merging files, except that one row in the composite may match many rows of the new table, and many rows in the composite may match one row of the new table. Method 3 indicates that the database managermust perform additional sorts at the end of processing the query block. The following sorts are possible: ORDER BY GROUP BY SELECT DISTINCT UNION. When METHOD is 3, CREATOR is all blanks, TNAME is the empty string, TABNO is zero, and SORTNEW is N.
CREATOR	CHAR(8) NOT NULL	Creator of the new table accessed in this plan step.
TNAME	VARCHAR(18) NOT NULL	Name of the new table accessed in the plan step.
TABNO	SMALLINT NOT NULL	Because a table may be joined to itself, there may be several references to the same table in a query block. TABNO distinguishes the different references. TABNO, CREATOR, and TNAME correspond to the columns with the same names in REFERENCE_TABLE. When there is no new table then these columns have the values specified when METHOD is 3 (see above).

Table 11. Columns in PLAN_TABLE (continued)

Column Name	Data Type	Description
ACCESSTYPE	CHAR(2) NOT NULL	Indicates how the database manager will access the data. These are the character values that can appear in ACCESSTYPE: **I1** Accesses the new table by a fetch operation on a fully-qualified unique index. This includes fetching the first or last value of the index. **I** Accesses the new table using an index and specific key values (identified in ACCESSCREATOR and ACCESSNAME). **N** Accesses the new table using an index on the column in an IN predicate with a list of literals. **W** Accesses the new table using an index, but without specific key values. This non-selective index scan locates the rows in a table when it is more efficient to scan the index than to scan all pages in the DBSPACE. **R** Accesses the new table by a scan of the DBSPACE in which it resides. **L** Accesses the new table through the internal list. The internal list is like a temporary table. It is created to contain the result of a materialized view. For any type of index access, ACCESSCREATOR and ACCESSNAME identify the index. ACCESSTYPE is blank for the top block of INSERT statements, as well as for UPDATE and DELETE statements that use WHERE CURRENT OF CURSOR clauses. ACCESSCREATOR is blanks and ACCESSNAME is the null value in that case. (Access for INSERT's is performed using the first index created; UPDATE and DELETE statements using the CURRENT OF CURSOR clause access using their cursors.)
MATCHCOLS	SMALLINT NOT NULL	For ACCESSTYPE 'I1', 'I', or 'N', the number of index keys that have key-matching predicates used in an index scan; otherwise, 0.
ACCESSCREATOR	CHAR(8) NOT NULL	For ACCESSTYPE 'I1', 'I', 'N', or 'W', ACCESSCREATOR contains the owner of the access path (index) that the database manageruses to access the table. Otherwise, ACCESSCREATOR contains blanks.
ACCESSNAME	VARCHAR(18) NOT NULL	For ACCESSTYPE 'I1', 'I', 'N', or 'W', ACCESSNAME contains the name of the access path (index) that the database manageruses to access the table. Otherwise, ACCESSNAME contains blanks.
INDEXONLY	CHAR(1) NOT NULL	Indicates whether an index is sufficient to satisfy the request, and to what degree. **Y** All predicates may be applied to the index pages and all data may be retrieved from the index pages. **W** All sargable predicates may be applied to the index pages, but data pages must be accessed to retrieve data satisfying the predicates or residual predicates. **N** Data pages must be accessed to resolve predicates and retrieve data. Note that, in a few circumstances, some predicate filtering may still be achieved using an index.

Table 11. Columns in PLAN_TABLE (continued)

Column Name	Data Type	Description
SORTNEW	CHAR(1) NOT NULL	To access a table in a particular order, the database manager may sort some fields of some rows of the new table (for example, for merge scan joins). These are the character values that can appear in SORTNEW: **N** If the database manager does not sort the new table. **U** If the database manager does sort, and removes duplicates. **Y** If the database manager sorts, and does not remove duplicates. SORTNEW is blank when no sort of the new table is possible, that is, when METHOD is 3 and there is no new table.
SORTCOMP	CHAR(1) NOT NULL	To access a composite in a particular order, the database manager may sort some fields of some rows of the composite. These are the character values that can appear in SORTCOMP: **N** If the database manager does not sort the composite. **U** If the database manager does sort, and removes duplicates. **Y** If the database manager does sort, and does not remove duplicates. SORTCOMP is blank when no sort of the composite is possible; that is, when METHOD is 0 and there is no composite yet.
SORTN_UNIQ	CHAR(1) NOT NULL	Whether a sort is performed on the new table to remove duplicate rows. Y = Yes; N = No.
SORTN_JOIN	CHAR(1) NOT NULL	Whether a sort is performed on the new table if METHOD is 2. Y = Yes; N = No.
SORTN_ORDERBY	CHAR(1) NOT NULL	Whether an ORDER BY clause results in a sort on the new table. Y = Yes; N = No.
SORTN_GROUPBY	CHAR(1) NOT NULL	Whether a GROUP BY clause results in a sort on the new table. Y = Yes; N = No.
SORTC_UNIQ	CHAR(1) NOT NULL	Whether a sort is performed on the composite table to remove duplicate rows. Y = Yes; N = No.
SORTC_JOIN	CHAR(1) NOT NULL	Whether a sort is performed on the composite table if METHOD is 2. Y = Yes; N = No.
SORTC_ORDERBY	CHAR(1) NOT NULL	Whether an ORDER BY clause results in a sort on the composite table. Y = Yes; N = No.
SORTC_GROUPBY	CHAR(1) NOT NULL	Whether a GROUP BY clause results in a sort on the composite table. Y = Yes; N = No.
TIMESTAMP	TIMESTAMP NOT NULL	The time at which the EXPLAIN statement was processed.
REMARKS	VARCHAR(254) NOT NULL	A field into which you can insert any character string of 254 or fewer characters.

Table 12. Columns in REFERENCE_TABLE

Column Name	Data Type	Description
QUERYNO	INTEGER	Query number. QUERYNO is intended for your use to distinguish among queries. (See COST_TABLE for a description of QUERYNO.)
RINO	SMALLINT NOT NULL	RINO is intended to distinguish among queries and internally generated queries. (See COST_TABLE for a description of RINO.)

Table 12. Columns in REFERENCE_TABLE (continued)

Column Name	Data Type	Description
QBLOCKNO	SMALLINT NOT NULL	Query block number, where 1 is the top level query block, that may have subqueries. Different query blocks receive different numbers.
PKGNAME	CHAR(8) NOT NULL	This identifies the name of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
PKGOWNER	CHAR(8) NOT NULL	This identifies the owner of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
REFTYPE	CHAR(6) NOT NULL	An indication of the purpose of the current row in this table. Rows are inserted for three reasons: 1. For each SQL statement, REFTYPE has a value indicating the type of statement: **SELECT** A select statement **INSERT** An insert statement **UPDATE** An update statement **DELETE** A delete statement **SELUPD** A select statement with a 'FOR UPDATE' clause **DELCUR** A delete where current of cursor statement **UPDCUR** An update where current of cursor statement 2. For each table referenced, REFTYPE has the value 'TABLE'. 3. For each column referenced, REFTYPE has the value 'COLUMN'.
CREATOR	CHAR(8) NOT NULL	Creator of a table referenced in the query block.
TNAME	VARCHAR(18) NOT NULL	Name of the table referenced in the query block.
TABNO	SMALLINT NOT NULL	Because there may be several references to the same table in a query block (because a table may be joined to itself), TABNO differentiates among the different references. TABNO may correspond to the order of tables in the FROM clause of the query.
CNAME	VARCHAR(18) NOT NULL	Name of the column.
COLNO	SMALLINT NOT NULL	Column number of a column in the table identified by CREATOR, TNAME, and TABNO. EXPLAIN REFERENCE causes at most one new row to be entered in REFERENCE_TABLE for a particular column (COLNO) of a table (TABNO) in a query block (QBLOCKNO).

Table 12. Columns in REFERENCE_TABLE (continued)

Column Name	Data Type	Description
FILTER	FLOAT NOT NULL	The filter factor associated with the query block's most selective predicate on this column. The selectivity of a predicate is the fraction of the rows of the column's table that is estimated to satisfy the predicate. Not all columns referenced in a statement have filter factors, however. For each reference to a column, the EXPLAIN statement determines a filter factor if the reference to the column meets these qualifications: 1. The column must be in a predicate that is connected by the AND logical operator to the rest of the WHERE clause. If the predicate is not connected by AND, it must have the only predicate in the WHERE clause. 2. The predicate in which the column appears must have the form "column op expression." For each such column reference, the EXPLAIN statement determines a "filter factor." The smallest of these filter factors is returned in FILTER. This value is between 0.0 and 1.0, and will be 1.0 if there are no predicates with filter factors for the column. Filter factor may be used to estimate the cost of modifying rows and indexes. Also, a small filter factor is one indicator that an index on the column might be useful for processing the statement.
DBSSPRED	CHAR(1) NOT NULL	Is there a sargable predicate (predicate applied at the first stage) associated with this column? **Y** There is a sargable predicate associated with this column. However, this sargable predicate may not necessarily be the most selective one. **N** There may be no sargable predicate associated with this column. For each reference to a column, the EXPLAIN statement determines sargability if the reference to the column meets these qualifications: 1. The column must be in a predicate that is connected by the AND logical operator to the rest of the WHERE clause. If the predicate is not connected by AND, it must have the only predicate in the WHERE clause. 2. The predicate in which the column appears must have the form "column op expression." For each such column reference, the EXPLAIN statement determines the sargability of the predicate associated with the column. If a sargable predicate exists, the value is set to 'Y'; otherwise, it is set to 'N'.
JOINPRED	CHAR(1) NOT NULL	Is there a sargable equi-join predicate (using equal value in tables to join) associated with this column? Y = Yes; N = No. If yes, then DBSSPRED must be Y as well.
ORDERCOL	SMALLINT NOT NULL	If this column is referenced in an ORDER BY clause, give its relative position in the ORDER BY clause and sort direction. If the column is not referenced in the ORDER BY clause, ORDERCOL is zero. Sort direction is indicated by a positive number for ascending order and a negative number for descending order.
GROUPCOL	SMALLINT NOT NULL	If this column is referenced in a GROUP BY clause, give its relative position in the GROUP BY clause. If the column is not referenced in the GROUP BY clause, GROUPCOL is zero.
UPDATECOL	CHAR(1) NOT NULL	If this column is in the SET clause of an UPDATE statement, indicate how it is updated. **L** Updated by a literal. **X** Updated by a column or expression. **blank** Column is not referenced in the SET clause

Table 12. Columns in REFERENCE_TABLE (continued)

Column Name	Data Type	Description
TIMESTAMP	TIMESTAMP NOT NULL	The time at which the EXPLAIN statement was processed.

Table 13. Columns in STRUCTURE_TABLE

Column Name	Data Type	Description
QUERYNO	INTEGER	Query number. QUERYNO is intended to distinguish among queries. (See COST_TABLE for a description of QUERYNO.)
RINO	SMALLINT NOT NULL	RINO is intended to distinguish among queries and internally generated queries. (See COST_TABLE for a description of RINO.)
QBLOCKNO	SMALLINT NOT NULL	Query block number, where 1 is the top level query block that may have subqueries. Different query blocks will receive different numbers.
PKGNAME	CHAR(8) NOT NULL	This identifies the name of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
PKGOWNER	CHAR(8) NOT NULL	This identifies the owner of the package in which this SQL statement originated. This field is blank for explicit EXPLAIN processing invoked by the EXPLAIN statement.
ROWCOUNT	INTEGER NOT NULL	Estimated number of rows returned for the query or subquery corresponding to this query block. For queries, this is the estimated size of the response. For update and delete statements, this is the estimated number of affected rows. ROWCOUNT can be used in estimating update costs. For insert statements, the ROWCOUNT for the top level query block (QBLOCKNO 1) is always 0, but the ROWCOUNT's for other query blocks, if any, are normal estimates. ROWCOUNT is also 0 for UPDATE and DELETE statements that use WHERE CURRENT OF CURSOR clauses.
TIMES	FLOAT NOT NULL	Estimated number of times that "dependent" query blocks of this block will be processed for each execution of this query block. This field is no longer in use, but is retained to provide for compatibility with older versions of the SQL/DS product.
PARENT	SMALLINT NOT NULL	The query block for which this block is performed. This may be the query block in whose WHERE clause the current query block appears. However, some query blocks can be processed earlier, at the opening of a "parent" query block, because there are no correlations to intermediate query blocks tables. In this case, PARENT identifies that ancestor, rather than the parent given by the statement's structure.

Table 13. Columns in STRUCTURE_TABLE (continued)

Column Name	Data Type	Description
ATOPEN	CHAR (1) NOT NULL	These are the characters that can appear in ATOPEN: **Y** If the query is done once at each open (new invocation) of the PARENT. **N** If the number of times that the current query block is invoked (per invocation of its parent) equals the TIMES field value of the parent.
TIMESTAMP	TIMESTAMP NOT NULL	The time at which the EXPLAIN statement was processed.

Examples

Place information about a SELECT statement that selects all the rows from the EMP_ACT table into your tables named REFERENCES_TABLE and COST_TABLE. Tag the entries that contain this information with the reference number 1500.

```
EXPLAIN REFERENCE, COST
SET QUERYNO = 1500
FOR SELECT * FROM EMP_ACT
```

FETCH

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables, host structures, or both.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See "DECLARE CURSOR" on page 235 for an explanation of the authorization required to use a cursor.

Syntax

```
►►──FETCH──cursor_name──┬─INTO────host_variable_list──────┬──────────────────►◄
                        └─USING DESCRIPTOR──descriptor_name─┘
```

Description

cursor_name
Identifies the select cursor to be used in the fetch operation. The *cursor_name* must identify a declared cursor as explained in "DECLARE CURSOR" on page 235. When the FETCH statement is processed, the cursor must be in the open state.

If the cursor is currently positioned on or after the last row of the result table:
- SQLCODE is set to +100, and SQLSTATE is set to '02000'.
- The cursor is positioned after the last row.
- Host variables and host structure subfields are not assigned values.

If the cursor is currently positioned before a row, the cursor is positioned on that row, and the values of that row are assigned to host variables and host structure subfields as specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, after execution of the FETCH statement the cursor is positioned on the next row. Values of that row are assigned to host variables and host structure subfields as specified by INTO or USING.

INTO
Introduces a list of host variables, host structures, or both.

host_variable_list
Identifies one or more host variables, host structures, or both, that must be declared in the program in accordance with the rules for declaring host variables and host structures.

The first value in the result row is assigned to the first host variable or host structure subfield in the list, the second value to the second variable or subfield, and so on.

USING DESCRIPTOR *descriptor_name*
Identifies an output SQLDA that must contain a valid description of zero or more host variables.

Before the FETCH statement is processed, the user must set some fields in the SQLDA as described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

The data type of a variable must be compatible with its corresponding value. If the value is numeric, the variable must have the capacity to represent the whole part of the value. For a datetime value, the variable must be a character string variable of a minimum length as defined in "String Representations of Datetime Values" on page 49. If the value is null, an indicator variable must be specified.

Each value with a corresponding variable is assigned to the variable in accordance with the assignment rules described in Chapter 3. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLCA is set to 'W'. If an assignment error occurs, the value is not assigned to the variable, and no more values are assigned to variables. Any values that have already been assigned to variables remain assigned.

Error Conditions

See the *DB2 Server for VSE & VM Application Programming* manual for a description of the possible errors when FETCH is processed.

Notes

Cursor Positioning

An open cursor has three possible positions:
- Before a row
- On a row
- After the last row.

If a cursor is on a row, that row is called the *current row* of the cursor. A cursor referenced in an UPDATE or DELETE statement must be positioned on a row. A cursor can only be on a row as a result of a FETCH statement.

It is possible for an error to occur that makes the state of the cursor unpredictable.

Examples

There are two tables, FORUM and ARCHIVE, each with the following columns:

Name:	FORUM	RECEIVED	SOURCE	TOPIC	ENTRY_TEXT
Type:	char(8) not null	timestamp not null	char(8) not null	char(64) not null	varchar(4000) not null
Desc:	Forum name	Date and time entry received	Userid of person appending entry	Topic within the forum	The text appended in this entry

The FORUM table contains a number of named forums. Each forum contains one or more topics and each topic contains one or more entries. When a topic is no longer current its entries are either deleted or moved to the ARCHIVE table.

The following PL/I program performs maintenance on the forum table. A user can invoke the program with one of three commands. Each command is accompanied by a string of text that can be found within the TOPIC column of the entries for a given topic (this need not be the entire TOPIC value). The three commands are:

- 1 (changes the contents of the TOPIC value for all that topic's entries)
- 2 (moves all entries for that topic to the ARCHIVE table)
- 3 (deletes all entries for that topic *without* archiving them).

```
CLEANUP:  PROC OPTIONS(MAIN);
    DCL NOT_END BIT(1);
    DCL  ACTION       BINARY FIXED(15);   /* 1=chg-topic   2=archive   3=delete */
  EXEC SQL  BEGIN DECLARE SECTION;
    DCL  SRCH_FORUM   CHAR(8);
    DCL  SRCH_TOPIC   CHAR(66) VARYING;
    DCL  NEW_TOPIC    CHAR(64) VARYING;
    DCL  FORUM        CHAR(8);
    DCL  1 ENTRY,
           5 TSTMP    CHAR(26),
           5 PERSON   CHAR(8),
           5 TOPIC    CHAR(64) VARYING;
    DCL  TXT          CHAR(4000) VARYING;
  EXEC SQL  END DECLARE SECTION;
  EXEC SQL  INCLUDE SQLCA;
  EXEC SQL  WHENEVER NOT FOUND CONTINUE;
  EXEC SQL  WHENEVER SQLWARNING CONTINUE;
  EXEC SQL  WHENEVER SQLERROR GOTO ERRCHK;

  EXEC SQL  CONNECT TO TOROLAB3;
  GET LIST (ACTION, SRCH_FORUM, SRCH_TOPIC, NEW_TOPIC);
  SRCH_TOPIC = '%' || SRCH_TOPIC || '%';
  EXEC SQL  DECLARE CUR CURSOR FOR
                     SELECT * FROM FORUM
                       WHERE FORUM = :SRCH_FORUM AND TOPIC LIKE :SRCH_TOPIC
                       FOR UPDATE OF TOPIC;
  EXEC SQL  OPEN CUR;
  EXEC SQL  FETCH CUR INTO :FORUM, :ENTRY, :TXT;
  IF SQLSTATE = '02000'
    THEN DO;
      DISPLAY ('No notes found for requested forum and topic');
      GO TO FINISHED;
    END;

  NOT_END = '1'B;
  DO WHILE (NOT_END);
    EXEC SQL  FETCH CUR INTO :FORUM, :ENTRY, :TXT;
    IF SQLSTATE = '02000' THEN
      NOT_END = '0'B;
    ELSE DO;
      SELECT;
        WHEN (ACTION = 1)                /* change topic value */
          EXEC SQL  UPDATE FORUM
                      SET TOPIC = :NEW_TOPIC
                      WHERE CURRENT OF CUR;
        WHEN (ACTION = 2)                /* archive entry to another table */
          DO;
            EXEC SQL  INSERT INTO ARCHIVE
                        VALUES (:FORUM, :TSTMP, :PERSON, :TOPIC, :TXT);
            EXEC SQL  DELETE FROM FORUM WHERE CURRENT OF CUR;
          END;
        WHEN (ACTION = 3)                /* delete topic */
          EXEC SQL  DELETE FROM FORUM WHERE CURRENT OF CUR;
      END; /* select */
    END;  /* else do */
  END;  /* do while */

FINISHED:
  EXEC SQL  CLOSE CUR;
  EXEC SQL  COMMIT WORK;
  RETURN;
ERRCHK:
  DISPLAY ('Unexpected Error -changes will be backed out');
  PUT SKIP LIST (SQLCA);
  EXEC SQL  WHENEVER SQLERROR CONTINUE;  /* continue if error on rollback */
  EXEC SQL  ROLLBACK WORK;
  RETURN;
END;  /* CLEANUP */
```

Extended FETCH

The Extended FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables. The cursor must have been opened using the Extended OPEN statement.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──FETCH──cursor_variable──USING DESCRIPTOR──descriptor_name──────────────────►◄
```

Description

cursor_variable
: Identifies the cursor that is to be used. The cursor must have been defined by a preceding Extended DECLARE CURSOR statement in the same logical unit of work.

USING DESCRIPTOR *descriptor_name*
: Identifies an output SQLDA that must contain a valid description of host variables.

: Before the Extended FETCH statement is processed, the user must set some fields in the SQLDA as described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

The indicated cursor must be declared and opened.

Notes

In most respects, the Extended FETCH statement is identical to the FETCH statement (see "FETCH" on page 283). However, in the Extended FETCH statement, the *cursor_name* is a host variable, thereby making it possible for a user to provide the cursor name when the program is run and to FETCH in a logical unit of work or program other than the one in which the statement was prepared. Extended DECLARE CURSOR, OPEN, and FETCH must occur in the same logical unit of work.

Examples

```
FETCH :CURSOR1 USING DESCRIPTOR MYSQLDA
```

GRANT (Package Privileges)

This form of the GRANT statement grants the privilege to process statements in a package.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

To process this statement, the privileges held by the authorization ID of the statement must include the EXECUTE privilege on the package and GRANT authority on that privilege. Someone with DBA authority may grant the EXECUTE privilege on a package owned by another user.

Syntax

```
                                               ┌─,──────────────────┐
                          (1)                  ▼                    │
►►──GRANT EXECUTE ON─────────package_name──TO────┬─authorization_name─┬┴────────────►
                                                 └─PUBLIC─────────────┘

►──┬──────────────────┬──────────────────────────────────────────────────────────►◄
   └─WITH GRANT OPTION─┘
```

Notes:

1 RUN can be specified as a synonym for EXECUTE to support applications developed for previous releases of SQL/DS.

Description

EXECUTE ON *package_name*
: Identifies the package upon which the EXECUTE privilege is being granted. The *package_name* must identify a package that exists at the application server.

TO
: Specifies to whom the privileges are granted.

authorization_name,...
: Lists one or more authorization IDs. You cannot use the ID of the GRANT statement itself; you cannot grant privileges to yourself.

PUBLIC
: Grants the EXECUTE privilege on the package to all users.

WITH GRANT OPTION
: Allows the named *authorization_name*s to grant the EXECUTE privilege on the package to other users.

If WITH GRANT OPTION is omitted, the named *authorization_name*s cannot grant the EXECUTE privilege to others unless they have received that authority from some other source.

The GRANT authority cannot be passed to PUBLIC. If you use PUBLIC and WITH GRANT OPTION together, the statement is processed; but a warning is given and the EXECUTE privilege is granted to PUBLIC without GRANT authority.

Notes

Only the authorization ID that preprocesses a package (or an authorization ID with DBA authority) can drop that package from the database. A 'drop' privilege cannot be granted to another authorization ID.

Examples

Example 1

Grant the ability to process the TIMESHEET package (which is used by the TIMESHEET program) to everyone.

```
GRANT EXECUTE ON TIMESHEET TO PUBLIC
```

Example 2

Grant the ability to process the TABB package (which is used by the TABB program) to KING, BROWN, and BLACK. Allow them to grant this privilege to others.

```
GRANT EXECUTE ON TABB
  TO KING, BROWN, BLACK
  WITH GRANT OPTION
```

GRANT (System Authorities)

This form of the GRANT statement changes passwords and authorities.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

DBA authority is needed to grant authorities and to change others' passwords. DBA authority is not needed for someone to change their own password if they have been granted connect authority explicitly by a DBA. (A user able to access the database only because connect authority has been granted to ALLUSERS cannot use this command to change their own password.)

Syntax

```
>>──GRANT──────┬──┬─CONNECT──┬──TO──┤ AUTH ├──┬────────┬──┬──────────────────────><
               │  ├─DBA──────┤                └┤ ID ├──┘  │
               │  └─RESOURCE─┘                            │
               │                                          │
               │              ┌─,──────────────────┐      │
               │              ▼                    │      │
               ├─CONNECT TO────┬─authorization_name─┴─────┤
               │               │            (1)     │     │
               │               └─ALLUSERS───────────┘     │
               └─SCHEDULE TO──subsystemid──IDENTIFIED BY──password──┘
```

AUTH:

```
   ┌─,──────────────────┐
   ▼                    │
├───authorization_name──┴─────────────────────────────────────────────┤
```

ID:

```
                  ┌─,────────┐
                  ▼          │
├──IDENTIFIED BY────password──┴────────────────────────────────────────┤
```

Notes:

1 ALLUSERS can only be specified once and is not applicable to a VSE application server.

Description

CONNECT
Grants CONNECT authority to the specified *authorization_name*s. A user can use this parameter with the IDENTIFIED BY clause to change his or her own password.

DBA
Grants DBA authority to the specified *authorization_name*s. This also means that the specified *authorization_name*s will be automatically granted CONNECT and RESOURCE authority. Someone with DBA authority has all privileges on all

objects in the database, including the authority to drop any object. However, a DBA may not grant any privileges on an object the DBA does not own unless the owner has given the DBA that right. A DBA also cannot revoke any privilege on an object unless the DBA granted that privilege in the first place. For a complete description of DBA authority, see the *DB2 Server for VSE & VM Database Administration* manual.

RESOURCE
Grants RESOURCE authority to the specified user(s). This also means that the specified user(s) will be automatically granted CONNECT authority. Someone with RESOURCE authority has the ability to create tables in public dbspaces.

TO
Introduces a list of one or more *authorization_names*

authorization_name
An authorization id.

ALLUSERS
Specifies that the CONNECT authority is granted implicitly to every system-defined user. Granting CONNECT to ALLUSERS is a special case that establishes implicit connect capability for all users in the system when operating under the DB2 Server for VM environment.

VSE Users

ALLUSERS is not a valid option since implicit CONNECT authority is not applicable to VSE application servers.

IDENTIFIED BY *password***...**
Adds or changes the password for each *authorization_name* specified. If you specify IDENTIFIED BY, you must include a password for every *authorization_name* specified. The password specifies the new or changed password for each of the specified *authorization_name*s. Passwords are limited to eight characters. The passwords and *authorization_name*s must correspond as shown in example 2 below. If the password is the same as the one that currently exists for the *authorization_name*, or if no passwords are specified, the change has no real effect.

SCHEDULE
Grants the authority to connect users without specifying a password. Used with the VSE Guest Sharing facility. For more information, see the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

TO *subsystemid*
The subsystem ID of the CICS subsystem running under the VSE guest.

IDENTIFIED BY *password*
The new or changed password by which the subsystem will identify itself.

Examples

Example 1

Grant DBA authority to THOMPSON and THORN.

```
GRANT DBA TO THOMPSON, THORN
```

Example 2

Grant CONNECT authority to BRIAN (with the password CONCON), ED (with the password NDPNDP), and JOHN (with the password LIBLIB).

```
GRANT CONNECT TO BRIAN, ED, JOHN
  IDENTIFIED BY CONCON, NDPNDP, LIBLIB
```

GRANT (Table Privileges)

This form of the GRANT statement grants privileges on table and views.

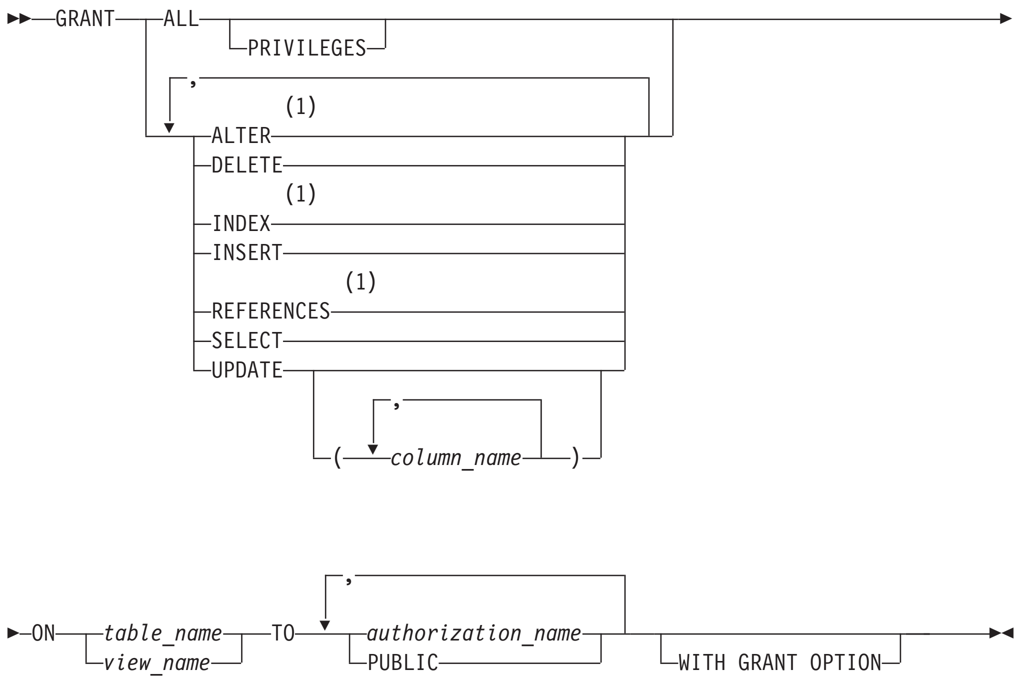
Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include the privilege being granted and GRANT authority on that privilege. Someone with DBA authority may grant table privileges on a table or view owned by another user.

Syntax



Notes:

1 The ALTER, INDEX and REFERENCES options do not apply to views.

Description

ALL or ALL PRIVILEGES
Grants table privileges on the table or view identified in the ON clause. The privileges granted are those possessed by the authorization ID of the GRANT statement. ALL PRIVILEGES is the default.

ALTER
Grants the privilege to use the ALTER TABLE statement. This privilege cannot be granted on a view.

DELETE
Grants the privilege to use the DELETE statement.

INDEX
Grants the privilege to use the CREATE INDEX statement. This privilege cannot be granted on a view.

INSERT
Grants the privilege to use the INSERT statement.

REFERENCES
Grants the privilege to create, drop, activate, or deactivate a referential constraint in which the table is the parent table. This privilege does not apply to views.

This privilege is required to reference the parent table when a referential constraint is defined or added by the CREATE TABLE or ALTER TABLE statement respectively.

This privilege is also required on the parent table when the user wants to use the ALTER TABLE statement to drop, activate, or deactivate a foreign key on a dependent table that references the parent table.

SELECT
Grants the privilege to use the SELECT statement or the CREATE VIEW statement.

UPDATE
Allows the grantee(s) to update the table or view.

(*column_name***,...)**
Restricts the update privilege to the columns listed. If a list of column names is not specified or if UPDATE is granted using the specification of ALL PRIVILEGES, the grantee(s) may update all updateable columns of the table, even those created later by the ALTER TABLE statement.

ON *table_name* or *view_name*
Identifies the table or view upon which you are granting the privileges. The *table_name* or *view_name* must identify a table or view that exists at the application server.

TO
Indicates to whom the privileges are granted.

*authorization_name***,...**
Lists one or more authorization IDs. The ID of the GRANT statement itself cannot be used. (Privileges cannot be granted to oneself.)

PUBLIC
Grants the privileges to all users.

WITH GRANT OPTION
Allows the named *authorization_name*s to grant the privileges to other *authorization_name*s. If you omit WITH GRANT OPTION, the named *authorization_name*s cannot grant the privileges to others unless they have that authority from some other source.

You cannot pass the GRANT authority to PUBLIC. If you use PUBLIC and WITH GRANT OPTION together, the statement is processed; but a warning is given and the privileges are granted to PUBLIC without GRANT authority.

Examples

Example 1

Given that you have DBA authority, and that you have all grant authorities on the table WESTERN_COURSES (owned by KATHLEEN), grant all privileges on the table to PUBLIC.

```
  GRANT ALL ON KATHLEEN.WESTERN_COURSES
    TO PUBLIC
```

Example 2

Grant the appropriate privileges on your CALENDAR table so that ROANNA and EMMA can read it and insert new entries into it, but do not allow them to change or remove any entries. Do not allow ROANNA or EMMA to grant those privileges to others.

```
  GRANT SELECT, INSERT ON CALENDAR
    TO ROANNA, EMMA
```

Example 3

Grant the UPDATE privilege on the RATING and CRITIQUE columns from the public table TORONTO_RESTAURANT (owned by ONTARIO) to MARGARET and COMPDEPT. Allow them to grant those privileges to others.

```
GRANT UPDATE (RATING, CRITIQUE) ON ONTARIO.TORONTO_RESTAURANT
  TO MARGARET, COMPDEPT
  WITH GRANT OPTION
```

INCLUDE

The INCLUDE statement inserts declarations, statements, or both, into a source program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It is not supported in REXX.

Authorization

None required.

Syntax

```
►►──INCLUDE───┬─SQLCA──────────┬──────────────────────────────────────────►◄
              ├─SQLDA──────────┤
              └─text_file_name─┘
```

Description

SQLCA
: Indicates the description of an SQL communication area (SQLCA) is to be included. INCLUDE SQLCA must not be specified more than once in the same program. INCLUDE SQLCA must not be specified if the program includes a stand-alone SQLCODE (see “SQL Return Codes” on page 142). For a description of the SQLCA, see “SQL Communication Area (SQLCA)” on page 353.

SQLDA
: Indicates the description of an SQL descriptor area (SQLDA) is to be included. SQLDA should not be specified in a COBOL, or Fortran program, as it will be interpreted as a text_file_name. For a description of the SQLDA, see “SQL Descriptor Area (SQLDA)” on page 359.

text_file_name
: Identifies an external source file to be used as input when your program is precompiled.

  The statements contained in the external source specified by *text_file_name* may be host language statements or SQL statements (except for another INCLUDE statement). INCLUDE *text_file_name* statements may not be nested, but the external source may contain INCLUDE SQLDA or INCLUDE SQLCA statements. The INCLUDE *text_file_name* may appear in an SQL DECLARE section or the entire SQL DECLARE section(s) may be placed within an external source file.

Notes

The INCLUDE statement may be used to obtain secondary input from a CMS file in VM or a source member in VSE. If a source program input to a preprocessor uses the INCLUDE facility, any files to be used as secondary input must be accessed by the user. The INCLUDE statement causes input to be read from the specified file name until the end of the file, at which time the SYSIN input in VM or the SYSIPT input in VSE resumes.

In VM

The file to be included must have one of the following file types:

Language	File Type
Assembler	ASMCOPY
C	CCOPY
COBOL	COBCOPY
Fortran	FORTCOPY
PL/I	PLICOPY

In VSE

The source member must be cataloged as one of the following source types:

Language	Source Type
Assembler	A
C	B
COBOL	C
Fortran	G
PL/I	P

For COBOL programs, INCLUDE SQLCA must not be specified in other than the Working Storage Section.

See the *DB2 Server for VSE & VM Application Programming* manual for more information on using external source files.

Examples

Include an SQL Communications Area into a PL/I program.

```
EXEC SQL  INCLUDE SQLCA;
```

INSERT

The INSERT statement inserts rows into a table or view. Inserting a row into a view also inserts the row into the table on which the view is based.

There are two forms of this statement:

- The *INSERT using VALUES* form inserts a single row into the table or view using the values provided or referenced.
- The *INSERT by subselect* form inserts one or more rows into the table or view using values from other tables or views.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

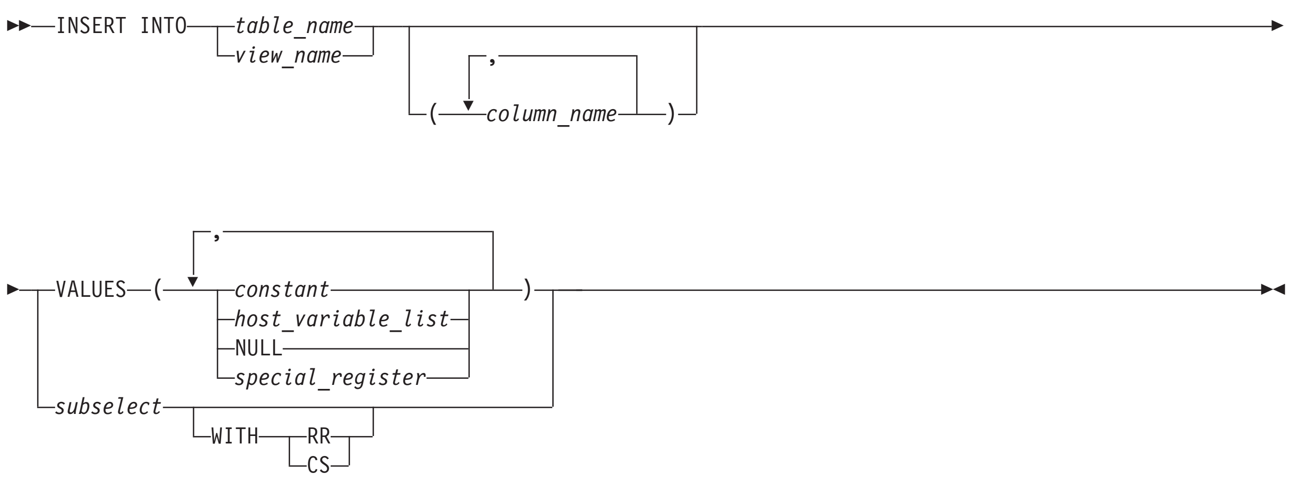
- Ownership of the table
- The INSERT privilege for the table or view
- DBA authority.

The INSERT privilege on a view is only inherent in DBA authority. Ownership of a view does not necessarily include the INSERT privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

If a *subselect* is specified, the privileges held by the authorization ID of the statement must also include at least one of the following:

- Ownership of the tables or views identified in the subselect
- The SELECT privilege on every table or view identified in the subselect
- DBA authority.

Syntax



Description

INTO *table_name*

INTO *view_name*
: Identifies the object of the insert operation. The name must identify a table or view that exists at the application server, but it must not identify a catalog table, a view of a catalog table, or a read-only view (see "Read-only views" on page 233). However, someone with DBA authority may insert rows into a few of the catalog tables. See "Updateable Columns" on page 371.

  A value cannot be inserted into a view column that is derived from:

  - A constant, expression, or scalar function
  - The same base table column as some other column of the view.

  If the object of the insert operation is a view with such columns, a list of column names must be specified, and the list must not identify these columns.

(*column_name***,...)**
: Specifies the columns for which insert values are provided. Each name must be an unqualified name that identifies a column of the table or view. The same column must not be identified more than once. A view column that cannot accept insert values must not be identified.

  Omission of the column list is an implicit specification of a list in which every column of the table or view is identified in left-to-right order. This list is established when the statement is prepared and therefore does not include columns that were added to a table after the statement was prepared.

  SQL statements can be implicitly or explicitly rebound (prepared again). The effect of a rebind on INSERT statements that do not include a column list is to re-establish the list. Therefore, the number of columns into which data will be inserted may change.

VALUES
: Introduces one row of values to be inserted. The values of the row are the values of the constants, host variables, host structure subfields, and keywords specified in the clause.

  Each host variable and host structure named must be described in the program in accordance with the rules for declaring host variables and host structures.

  The number of values in the VALUES clause must equal the number of names in the column list. The first value is inserted in the first column in the list, the second value in the second column, and so on.

  For an explanation of *constant* and *host-variable-list*, see Chapter 3. For a description of *special-register*, see "Special Registers" on page 62. NULL specifies the null value. A constant or special register cannot be used to specify the insert value for a long string column.

subselect
: Inserts the rows of the result table of a subselect. There may be one, more than one, or none. If there are none, SQLCODE is set to +100 and SQLSTATE is set to '02000'.

  (For an explanation of subselect, see Chapter 5, "Queries," on page 121.)

  The base object of the INSERT, and the base object of the subselect, or any subquery of the subselect, must not be the same table.

  The number of columns in the result table must equal the number of names in the column list. The value of the first column of the result is inserted in the first column in the list, the second value in the second column, and so on.

A non-null value cannot be inserted into a long string column using a subselect.

WITH Specifies the isolation level at which the subselect is executed.

RR
Repeatable read

CS
Cursor stability

The default isolation level of the statement is the isolation level of the package.

INSERT Rules

Insert values must satisfy the following rules. If they do not, or if any other errors occur during the execution of the INSERT statement, no rows are inserted.

- *Default values*: The value inserted in any column that is not in the column list is null. Columns that do not allow null values must be included in the column list. Similarly, if you insert into a view, the null value is inserted into any column of the base table that is not included in the view. Hence, all columns of the base table that are not in the view must allow null values.
- *Assignment*: Insert values are assigned to columns in accordance with the assignment rules described in Chapter 3.
- *Validity*: If the table named, or the base table of the view named, has one or more unique indexes, each row inserted into the table must conform to the constraints imposed by those indexes.

  If you name a view whose definition includes WITH CHECK OPTION, each row inserted into the view must conform to the definition of the view. If the view you name is dependent on other views whose definitions include WITH CHECK OPTION, the inserted rows must also conform to the definitions of those views.

  If you name a view whose definition does not include WITH CHECK OPTION, rows can be inserted that do not conform to the definition of the view. Those rows cannot appear in the view but are inserted into the base table of the view.

  For an explanation of the rules governing these situations, see "CREATE VIEW" on page 231.
- *Referential Integrity*: For each constraint defined on the table, each non-null insert value of each foreign key must be equal to a primary key value of the parent table.
- *Length*: If the insert value is a number, the column must be a numeric column with the capacity to represent the integral part of the number. For INSERT using VALUES, if the insert value is a string, the column must be a string column with a length attribute at least as great as the length of the string. For INSERT by subselect, the column may be a string column with a shorter length attribute, in which case truncation will occur with no error. Note that character string values may also be assigned to datetime columns as defined in "Datetime Assignments" on page 57.

If you are inserting rows into a parent table that is part of a referential constraint, the database manager implicitly checks that the primary key remains unique and does not contain null values.

Notes

Rows are inserted in an order determined by the database manager; that is, no facility is provided to specify the position in the table of a newly inserted row.

If an error occurs during the execution of an INSERT, you must inspect SQLWARN6 to determine the extent of the error. The following are current settings for SQLWARN6 when there is an error indication and the possible responses:

1. SQLWARN6 is set to 'S'. A severe error has occurred, leaving the system in an unusable state.
   - No further requests are possible. The application must end, or, in a DB2 Server for VSE & VM environment, may switch to another database.
2. SQLWARN6 is set to 'W'. An error occurred causing the LUW to be rolled back automatically. The system is still in a usable state. The application can either:
   - begin a new LUW and proceed **or**
   - stop.
3. SQLWARN6 is blank. An error has occurred, but the LUW is still active. For recoverable pools, any changes made by the request have been rolled back, hence the failing request has not left any partial results in the database. For information on nonrecoverable storage pools, see the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual. The application can do one of the following:
   - Continue forward processing of the LUW
   - Commit the changes made before the failing request
   - Roll back the LUW.

The order of rows being inserted is determined by the database manager; no facility is provided to specify the position in the table of a newly inserted row. The SQLERRD(3) portion of the SQLCA indicates the number of rows that were inserted.

Unless appropriate locks already exist, one or more exclusive locks are acquired at the execution of a successful INSERT statement. Until the locks are released, an inserted row can only be accessed by the application process that performed the insert. For further information about locking, see the description of the COMMIT, ROLLBACK, LOCK TABLE, and LOCK DBSPACE statements.

Put blocking is available with the DRDA protocol if the application has been preprocessed with the IBLOCK option. The database manager does not notify the application program of an insert error until the INSERTs that fills a block is processed. To determine when (or if) rows are actually inserted into the database, your program should examine SQLERRD(3) in the SQLCA when doing INSERTs.

For example, assuming 10 data rows to be inserted fit into one block, and that the data for the fourth insert is in error. The database manager tries to process the block of ten inserts, but encounters the error in the fourth row. It stops processing the block - that is, three rows are inserted successfully. SQLERRD(3) contains the number of rows that were successfully inserted. In this case, it contains a value of 3. If all rows were inserted successfully, it would contain 10. The application program can use SQLERRD(3) to determine where the error occurred.

Examples

Example 1

Insert a new department with the following specifications into the DEPARTMENT table:

- Department number (DEPTNO) is ‘E31’
- Department name (DEPTNAME) is ‘ARCHITECTURE’
- Managed by (MGRNO) a person with number ‘00390’
- Reports to (ADMRDEPT) department ‘E01’.

```
INSERT INTO DEPARTMENT
  VALUES ('E31', 'ARCHITECTURE', '00390', 'E01')
```

Example 2

Insert a new department into the DEPARTMENT table as in example 1, but do not assign a manager to the new department.

```
INSERT INTO DEPARTMENT (DEPTNO, DEPTNAME, ADMRDEPT)
  VALUES ('E31', 'ARCHITECTURE', 'E01')
```

Example 3

Create a temporary table MA_EMP_ACT with the same columns as the EMP_ACT table. Load MA_EMP_ACT with the rows from the EMP_ACT table with a project number (PROJNO) starting with the letters 'MA'.

```
CREATE TABLE MA_EMP_ACT
      (EMPNO     CHAR(6)  NOT NULL,
       PROJNO    CHAR(6)  NOT NULL,
       ACTNO     SMALLINT  NOT NULL,
       EMPTIME   DEC(5,2),
       EMSTDATE  DATE,
       EMENDATE  DATE )
INSERT INTO MA_EMP_ACT
  SELECT * FROM EMP_ACT
    WHERE SUBSTR(PROJNO, 1, 2) = 'MA'
```

Example 4

Use a PL/I program statement to add a skeleton project to the PROJECT table. Obtain the project number (PROJNO), project name (PROJNAME), department number (DEPTNO), and responsible employee (RESPEMP) from host variables and a host structure. Use the current date as the project start date (PRSTDATE). Assign a NULL value to the remaining columns in the table.

```
.
.
DCL 1 PROJECT,
      5 PRJNO   CHAR(5),
      5 PRJNM   CHAR(24) VARYING;
DCL 1 EMPLOYEE,
      5 DPTNO   CHAR(3),
      5 REMP    CHAR(6),
      5 LNAME   CHAR(25);
.
.
.
EXEC SQL  INSERT INTO PROJECT  (  PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE)
            VALUES (:PROJECT, :EMPLOYEE.DPTNO, :REMP, CURRENT DATE);
```

LABEL ON

The LABEL ON statement adds or replaces labels in the catalog descriptions of tables, views, or columns.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table or view
- DBA authority.

Syntax

```
►►──LABEL ON──┬─┤ options_a ├─IS──string_constant───────────────────────────►◄
              └─┬─table_name─┬──(──┤ options_b ├──)─┘
                └─view_name──┘
```

options_a

```
├──┬─TABLE───┬─table_name─┬───────────┬──────────────────────────────────────┤
   │         └─view_name──┘           │
   └─COLUMN───┬─table_name.column_name─┬┘
              └─view_name.column_name──┘
```

options_b

```
   ┌─,─────────────────────────────────┐
   ▼                                   │
├────column_name──IS──string_constant──┴────────────────────────────────────┤
```

Description

TABLE
Indicates that the label is for a table or a view.

table_name
view_name
Identifies a table or view to which the label applies. The name must identify a table or view at the application server.

The label is placed into the TLABEL column of the SYSTEM.SYSCATALOG catalog table for the row that describes the table or view.

COLUMN
Indicates that the label is for a column.

table_name.column_name
view_name.column_name
Identifies the column, qualified by the name of the table or view in which it appears. The *column_name* must identify a column of the specified table or view that exists at the application server.

The label is placed in the CLABEL column of the SYSTEM.SYSCOLUMNS catalog table, for the row that describes the column.

Multiple Labels:
To define a label for more than one column within the same table or view within the same statement, the table or view name is followed by a list of one or more column_name and string-constant pairs in parentheses.

The *column_name* must identify a column of the specified table or view that exists at the application server.

IS Introduces the label you want to provide.

string_constant
Can be any SQL character string constant of up to 30 characters. The constant may contain mixed double-byte and single-byte characters.

Notes

Unlike synonyms, labels cannot be used as identifiers. Instead, they can be used in displays created by applications that process SQL statements dynamically.

A DESCRIBE statement specified with USING BOTH or USING LABELS can be used to return column labels in an SQLDA. The program can then move the label from the SQLNAME field of the SQLDA into a work area. A column is considered to have no label if either its LABEL column in SYSTEM.SYSCOLUMNS is NULL, or if it has a zero length value. If there is no column label when the program issues a DESCRIBE, the SQLNAME field of the SQLDA is set to length 0, and the field is cleared to 30 blanks. For this reason, the program should move the label into a work area using the length returned in SQLDA only after it makes sure that the length is not zero.

Examples

Example 1

Insert a label for the EMP_ACT table into the catalog.

```
LABEL ON TABLE EMP_ACT
  IS 'EMPLOYEE ACTIVITY BY PROJECT'
```

Example 2

Insert a label for the EMP_VIEW1 view into the catalog.

```
LABEL ON TABLE EMP_VIEW1
  IS 'EMPLOYEE WITHOUT SALARY'
```

Example 3

Insert a label for the EDLEVEL column of the EMPLOYEE table into the catalog.

```
LABEL ON COLUMN EMPLOYEE.EDLEVEL
  IS 'HIGHEST GRADE LEVEL'
```

Example 4

Insert a label for two different columns of the EMPLOYEE table into the catalog.

```
LABEL ON EMPLOYEE
  (WORKDEPT IS 'DEPTNO IN EMPLOYEE',
         EDLEVEL  IS 'HIGHEST GRADE LEVEL ')
```

LOCK DBSPACE

The LOCK DBSPACE statement either prevents concurrent application processes from changing a dbspace or prevents concurrent application processes from using a dbspace.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the dbspace
- DBA authority.

Syntax

```
►►──LOCK DBSPACE──dbspace_name──IN──┬─SHARE─────┬──MODE──────────────────────────►◄
                                    └─EXCLUSIVE─┘
```

Description

dbspace_name
: Identifies the dbspace to be locked. The dbspace must exist at the application server. You cannot lock any dbspace containing the database manager's system catalog.

  The LOCK statement can be used to lock both private and public dbspaces. If the *dbspace_name* is unqualified, the database manager will first look for a private dbspace and, if that does not exist, it will look for a public dbspace with the same dbspace name.

IN SHARE MODE
: Prevents concurrent application processes from executing any but read-only operations on the dbspace.

IN EXCLUSIVE MODE
: Prevents concurrent application processes from executing any operations on the dbspace. This option requires a Z lock on the dbspace.

Locking prevents concurrent operations. A lock is not necessarily acquired during the execution of LOCK DBSPACE if a suitable lock already exists. The lock that prevents the concurrent operations is held until the termination of the unit of work.

Examples

Obtain a lock on the dbspace named DSP3. Allow others to read from the DSP3 while it is locked.

```
LOCK DBSPACE DSP3 IN SHARE MODE
```

LOCK TABLE

The LOCK TABLE statement either prevents concurrent application processes from changing a table or prevents concurrent application processes from using a table.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table
- The SELECT privilege for the table
- DBA authority.

Syntax

```
►►──LOCK TABLE──table_name──IN──┬─SHARE─────┬──MODE──────────────────────────────►◄
                                └─EXCLUSIVE─┘
```

Description

table_name
Identifies the table. The *table_name* must identify a base table that exists at the application server. If you lock a table in a private dbspace the entire dbspace is locked because locking is always performed at the dbspace level for private dbspaces.

IN SHARE MODE
Prevents concurrent application processes from executing any but read-only operations on the table.

IN EXCLUSIVE MODE
Prevents concurrent application processes from executing any operations on the table. This option requires an IX lock on the dbspace and a Z lock on the table.

Locking prevents concurrent operations. A lock is not necessarily acquired during the execution of LOCK TABLE if a suitable lock already exists. The lock that prevents the concurrent operations is held until the termination of the unit of work.

The lock is acquired when the LOCK TABLE statement is processed.

Examples

Obtain a lock on the DEPARTMENT table. Do not allow others to either update or read from DEPARTMENT while it is locked.

```
LOCK TABLE DEPARTMENT IN EXCLUSIVE MODE
```

OPEN

The OPEN statement opens a cursor.

Invocation

This statement can be embedded only in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

See “DECLARE CURSOR” on page 235 for the authorization required to use a cursor. The authorization for the OPEN statement is checked when the related DECLARE CURSOR statement is prepared.

Syntax

```
►►──OPEN──cursor_name──┬──────────────────────────────────┬──────────────►◄
                       ├─USING───host_variable_list───────┤
                       └─USING DESCRIPTOR──descriptor_name─┘
```

Description

cursor_name
: Identifies the cursor to be opened. The *cursor_name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement. When the OPEN statement is processed, the cursor must be in the closed state, and it must have been successfully prepared or declared.

If using an *insert-cursor* and the program is blocking, this statement tells the application server to prepare to block the rows to be inserted. If not blocking, the application server prepares to insert a single row into the database. Rows are not actually inserted into the database until one or more PUT statements have been processed.

If opening a *query-cursor*, the result table of the cursor is derived by evaluating that select-statement. The evaluation uses the current values of any special registers specified in the select-statement and the current values of any host variables or host structures specified in it or in the USING clause of the OPEN statement. The rows of the result table may be derived during the execution of the OPEN statement, and a temporary table created to hold them; or they may be derived during the execution of subsequent FETCH statements. In either case, the cursor is placed in the open state and positioned before the first row of its result table. If the table is empty, the position of the cursor is effectively “after the last row.”

USING
: Introduces a list of host variables or host structures or both whose values are substituted for the parameter markers (question marks) of a prepared statement. (For an explanation of parameter markers, see “PREPARE” on page 313.) If the DECLARE CURSOR statement names a prepared statement that includes parameter markers, you must use USING. If the prepared statement does not include parameter markers, USING is ignored. USING must not be used if the select-statement of the cursor is specified in the DECLARE CURSOR statement.

A USING clause cannot appear in the OPEN statement for an insert-cursor.

host_variable_list
Identifies a list of host variables, host structures, or both, that must be declared in the program in accordance with the rules for declaring host variables and host structures.

The total number of host variables and host structure subfields must be the same as the number of parameter markers in the prepared statement. The *n*th variable or subfield corresponds to the *n*th parameter marker in the prepared statement.

USING DESCRIPTOR *descriptor_name*
Identifies an input SQLDA that must contain a valid description of host variables.

Before the OPEN statement is processed, the user must set some fields in the SQLDA as described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

If the select-statement of the cursor was prepared (rather than declared) and that statement contains parameter markers, when that statement is evaluated each parameter marker in the statement is effectively replaced by its corresponding host variable. With the exception of the LIKE predicate, the replacement of a parameter marker is an assignment operation in which the source is the value of the host variable, and the target is a variable within the database manager. The attributes of the target variable are determined as follows:

- If the parameter marker was specified as the operand of a unary minus, the target is double-precision floating-point.
- If the parameter marker was specified as the operand of an arithmetic operator, the data type, scale, and precision of the target are the same as the other operand of that operator.
- If the parameter marker was specified as a comparison operand, the attributes of the target are the same as the other operand of the predicate. However, if the data type of the other operand is DATE, TIME, or TIMESTAMP, the target is effectively CHAR(254).
- When the parameter marker is specified as a comparison operand in the BETWEEN predicate,
  - If there is an operand that is specified solely as a column name (or a column function with the argument being a column with a field procedure defined on it), then the attributes of the leftmost operand are used.
  - Otherwise, the attributes of the leftmost operand that is not a parameter marker are used.
- When the parameter marker is specified as a comparison operand in the IN predicate,
  - The attributes of the leftmost operand that is not a parameter marker are used.

If the parameter marker is the pattern in a LIKE predicate, then:

- If the first operand in the predicate is a character string column, the target is VARCHAR(n), where n is 10 more than the length attribute of the column with this exception: if that length is greater than 246, then n is 256.
- If the first operand in the predicate is a graphic string column, the target is VARCHAR(n), where n is 5 more than the length attribute of the column with this exception: if that length is greater than 123, then n is 128.

Let V denote a host variable that corresponds to parameter marker P. The value of V is assigned to the target variable for P in accordance with the rules for assigning a value to a column. Thus:

- V must be compatible with the target.
- If V is a string, its length must not be greater than the length attribute of the target.
- If V is a number, the absolute value of its integer part must not be greater than the maximum absolute value of the integer part of the target.
- If the attributes of V are not identical to the attributes of the target, the value is converted to conform to the attributes of the target.

When the SELECT statement of the cursor is evaluated, the value used in place of P is the value of the target variable for P. For example, if V is CHAR(6), and the target is CHAR(8), the value used in place of P is the value of V padded with two blanks.

The USING clause is intended for a prepared SELECT statement that contains parameter markers. However, it can also be used when the SELECT statement of the cursor is part of the DECLARE CURSOR statement. In this case the OPEN statement is processed as if each host variable in the SELECT statement were a parameter marker, except that the attributes of the target variables are the same as the attributes of the host variables in the SELECT statement. The effect is to override the values of the host variables in the SELECT statement of the cursor with the values of the host variables specified in the USING clause.

Notes

Closed state of cursors

All cursors in a program are in the closed state when:

- The program is initiated
- A program initiates a new unit of work by executing a COMMIT or ROLLBACK statement.

A cursor can also be in the closed state because:

- A CLOSE statement was processed
- An error was detected that made the position of the cursor unpredictable.

To retrieve rows from the active set of a *query-cursor*, a FETCH statement must be processed while the cursor is open. To insert rows into the active set of an *insert-cursor*, a PUT statement must be processed while the cursor is open. The only way to change the state of a cursor from closed state to open is to process an OPEN statement.

Effect of temporary tables: If the result table of a query cursor is not read-only, its rows are derived during the execution of subsequent FETCH statements. The same method may be used for a read-only result table. However, if a result table is read-only, the database manager may choose to use the temporary table method instead. With this method the entire result table is inserted into a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in these two ways:

- An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
- An INSERT, UPDATE, and DELETE statement processed while the cursor is open cannot affect the result table.

Conversely, if a temporary table is not used, INSERT, UPDATE, and DELETE statements processed while the cursor is open can affect the result table if issued from the same application process. The effect of such operations is not always predictable. For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and you insert a row into T, the effect of that insert on the result table is not predictable because its rows are not ordered. A subsequent FETCH C might or might not retrieve the new row of T.

Examples

Example 1

Write the embedded statements in a COBOL program that will:

1. Define a cursor C1 that is to be used to retrieve all rows from the DEPARTMENT table for departments that are administered by (ADMRDEPT) department 'A00'
2. Place the cursor C1 before the first row to be fetched.

```
EXEC SQL  DECLARE C1 CURSOR FOR
            SELECT DEPTNO, DEPTNAME, MGRNO FROM DEPARTMENT
              WHERE ADMRDEPT = 'A00'  END-EXEC.

EXEC SQL  OPEN C1  END-EXEC.
```

Example 2

Code an OPEN statement to associate a cursor DYN_CURSOR with a dynamically defined select-statement in a PL/I program. Assume each prepared select-statement always has two parameter markers in its WHERE clause with the first having a data type of integer and the second having a data type of varchar(64). (The related host variable definitions, PREPARE statement and DECLARE CURSOR statement are also shown in the example below.)

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  HV_INT      BINARY    FIXED(31);
  DCL  HV_VCHAR64  CHAR(64)  VARYING;
  DCL  STMT1_STR   CHAR(200)  VARYING;
EXEC SQL  END DECLARE SECTION;

EXEC SQL  PREPARE STMT1_NAME FROM :STMT1_STR;

EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING :HV_INT, :HV_VCHAR64;
```

Example 3

Code an OPEN statement as in example 2, but in this case the number and data types of the parameter markers in the WHERE clause are not known.

```
EXEC SQL  BEGIN DECLARE SECTION;
  DCL  STMT1_STR   CHAR(200)  VARYING;
EXEC SQL  END DECLARE SECTION;
EXEC SQL  INCLUDE SQLDA;

EXEC SQL  PREPARE STMT1_NAME FROM :STMT1_STR;
EXEC SQL  DECLARE DYN_CURSOR CURSOR FOR STMT1_NAME;

EXEC SQL  OPEN DYN_CURSOR USING DESCRIPTOR :SQLDA;
```

Example 4

This example shows the SQL statements used with a cursor CURSOR3 in a PL/I program. In this program, CURSOR3 inserts a row into the MA_ACT view (and therefore into the EMP_ACT table, which is the base table for the view) based on the values in the host variables EMNUM *(char(6))*, PJNUM *(char(6))*, ACNUM *(smallint)*, EMTIM *(dec(5,2))*, STDAT *(date)*, and EMDAT *(date)*.

```
EXEC SQL  DECLARE CURSOR3 CURSOR FOR
            INSERT INTO MA_ACT
              VALUES (:EMNUM, :PJNUM, :ACNUM, :EMTIM, :STDAT, :EMDAT);

EXEC SQL  OPEN CURSOR3;

EXEC SQL  PUT CURSOR3;

EXEC SQL  CLOSE CURSOR3;
```

Extended OPEN

The Extended OPEN statement opens a cursor declared using an Extended DECLARE CURSOR statement for a previously prepared statement. The open cursor retrieves the results of a query, or inserts values into the database.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──OPEN──cursor_variable──┬──────────────────────────────────┬──────────►◄
                           └─USING DESCRIPTOR──descriptor_name─┘
```

Description

cursor_variable
Identifies the cursor that is to be opened. The cursor must have been defined by a preceding Extended DECLARE CURSOR statement in the same logical unit of work.

USING DESCRIPTOR *descriptor_name*
Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the statement was prepared.

Before the Extended OPEN statement is processed, the user must set the fields in the SQLDA described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

When the cursor is to be used for inserting data into a table, the USING DESCRIPTOR clause should not be included because the clause must be in the PUT statement.

Notes

In most respects, the Extended OPEN statement is similar to the OPEN statement (see "OPEN" on page 307). However, in the Extended OPEN statement, the *cursor_name* is a host variable, thereby making it possible for a user to provide the cursor name when the program is run and to open the cursor in a logical unit of work or program other than the one in which the statement was prepared. Extended DECLARE CURSOR and Extended OPEN must occur in the same logical unit of work.

Examples

```
OPEN :CURSOR1 USING DESCRIPTOR MYSQLDA
```

PREPARE

The PREPARE statement is used by application programs to dynamically prepare an SQL statement for execution. The PREPARE statement creates an executable SQL statement, called a *prepared statement*, from a character string form of the statement, called a *statement string*. The prepared statement is a named object that can be referred to only within the logical unit of work in which it is created.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization rules are those defined for the SQL statement specified by the PREPARE statement. For example, see Chapter 5, “Queries,” on page 121 for the authorization rules that apply when a select-statement is prepared. The authorization ID is the run-time authorization ID.

Syntax

```
►►──PREPARE──statement_name──FROM──┬─string_constant─┬──────────────────────────►◄
                                   └─host_variable───┘
```

Description

statement_name
: Provides a name for the prepared statement. No two prepared statements in a single source program may use the same statement name. In REXX, the *statement_name* must not be the same as the cursor_name declared in the program.

FROM
: Introduces the statement string. The statement string is the value of the specified *string_constant* or the identified *host_variable*.

string_constant
: String constants are supported in all languages except Assembler and C.

You should avoid using either delimited identifiers or DBCS strings in statements specified in string constants because results are unpredictable.

When the *string_constant* form of the PREPARE statement is used in Fortran programs:
- If the *statement_name* is referenced in a DECLARE CURSOR statement, the PREPARE statement must come first.
- Any unqualified objects are qualified with the authorization ID of the person preparing the program.

host_variable
: Identifies a host variable that is described in the program in accordance with the rules for declaring character string variables. An indicator variable must not be specified.

In Assembler, C, COBOL, and REXX, the host variable must be a varying-length string variable. In C, it cannot be a NUL-terminated string. In Fortran, the host variable must be a fixed-length string variable. In PL/I, the host variable can either be a fixed-length or varying-length string variable. The host variable must have a maximum length of 8192.

In a PL/I Version 2 program, a prepared statement containing DBCS characters must be coded as a mixed string using the new PL/I Mixed format.

For example:

```
DYNSTR = 'SELECT COL1 FROM TABLE   WHERE COL2 = G'<...>'M;
EXEC SQL PREPARE STMT1 FROM :DYNSTR;
```

Rules for statement strings

The *string_constant* or *host_variable* must contain one of the following SQL statements:

ACQUIRE DBSPACE	GRANT Package Privileges
ALLOCATE CURSOR	GRANT System Authorities
ALTER DBSPACE	GRANT Table Privileges
ALTER TABLE	INSERT
ASSOCIATE LOCATORS	LABEL ON
COMMENT ON	LOCK DBSPACE
CREATE INDEX	LOCK TABLE
CREATE SYNONYM	REVOKE Package Privileges
CREATE TABLE	REVOKE System Authorities
CREATE VIEW	REVOKE Table Privileges
DELETE	*select-statement*
DROP	UPDATE
EXPLAIN	UPDATE STATISTICS

Furthermore, the statement string must not:
- Begin with EXEC SQL and end with a statement terminator
- Include references to host variables
- Include comments.

Parameter markers: Although a statement string cannot include references to host variables, it may include *parameter markers*; those can be replaced by the values of host variables when the prepared statement is processed. A parameter marker is a question mark (?) that is used where a host variable could be used if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see "OPEN" on page 307 and "EXECUTE" on page 264.

Rules for parameter markers:
- Parameter markers must not be used:
  - In a select list (SELECT ? is incorrect)
  - As an operand of the concatenation operator
  - As both operands of a single arithmetic or comparison operator (WHERE ? = ? is incorrect)
  - As an operand in a datetime arithmetic expression
- At least one of the operands of the BETWEEN or IN predicates must *not* be a parameter marker.
- An argument of a scalar function cannot be specified solely as a parameter marker. For example, **VALUE**(COL1, COL2, ?) is not valid.
- If a scalar function is used in other than a SELECT list, and it has an argument that can be specified as an arithmetic expression, a parameter marker can be included in that expression, provided that it is the operand of an arithmetic operator and that the other operand is a number.

Notes

When a PREPARE statement is processed, the statement string is parsed and checked for errors. If the statement string is incorrect, a prepared statement is not created and the error condition that prevents its creation is reported in the SQLCA.

Prepared statements can be referred to in the following kinds of statements, with the following restrictions shown:

In ...	**The prepared statement ...**
DESCRIBE	has no restrictions
DECLARE CURSOR	must be a select-statement or an insert-statement
EXECUTE	must *not* be a select-statement

A prepared statement can be processed many times. Indeed, if a prepared statement is not processed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

All prepared statements created in a logical unit of work are destroyed when the logical unit of work is terminated.

Examples

Example 1

Prepare and process a non-select-statement in a COBOL program. Assume the statement is contained in a host variable HOLDER and that the program will place a statement string into the host variable based on some instructions from the user. The statement to be prepared does not have any parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER  END-EXEC.

EXEC SQL  EXECUTE STMT_NAME  END-EXEC.
```

Example 2

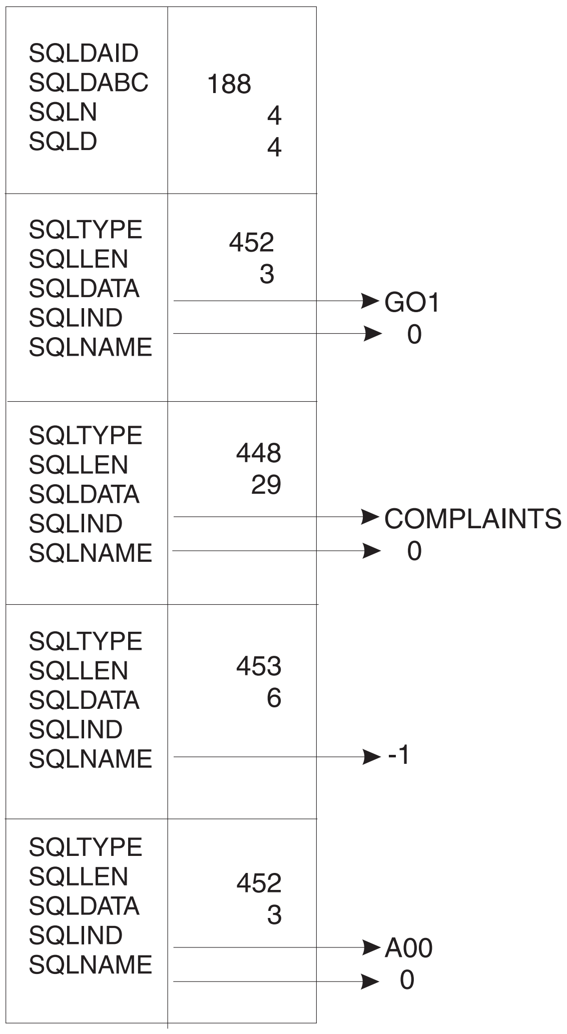
Prepare and process a non-select-statement as in example 1, except code it for a PL/I program. Also assume the statement to be prepared can contain any number of parameter markers.

```
EXEC SQL  PREPARE STMT_NAME FROM :HOLDER;

EXEC SQL  EXECUTE STMT_NAME USING DESCRIPTOR :INSERT_DA;
```

Assume that the following statement is to be prepared:

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

To insert department number G01 named COMPLAINTS, which has no manager and reports to department A00, the structure INSERT_DA should have the following values before running the EXECUTE statement.


SQLDAID
SQLDABC
SQLN
SQLD
188
4
4
SQLTYPE
SQLLEN
SQLDATA
SQLIND
SQLNAME
452
3
GO1
0
SQLTYPE
SQLLEN
SQLDATA
SQLIND
SQLNAME
448
29
COMPLAINTS
0
SQLTYPE
SQLLEN
SQLDATA
SQLIND
SQLNAME
453
6
-1
SQLTYPE
SQLLEN
SQLDATA
SQLIND
SQLNAME
452
3
A00
0

Extended PREPARE

The **Basic Extended PREPARE** and **Single Row Extended PREPARE** forms of the Extended PREPARE statement permit a statement to be prepared and stored in a package for later execution.

The **Empty Extended PREPARE** form of the Extended PREPARE statement provides support for dynamic SQL statements in non-modifiable packages. It is used in conjunction with the Temporary Extended PREPARE form of the Extended PREPARE statement.

The **Temporary Extended PREPARE** form of the Extended PREPARE statement provides support for dynamic SQL statements in non-modifiable packages.

The package you are preparing into must have been created with the CREATE PACKAGE statement.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the first three forms of the Extended PREPARE statement must have at least one of the following:
- ownership of the package
- DBA authority.

The authorization ID of the Temporary Extended PREPARE form must have at least one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

Basic Extended PREPARE

```
►►──PREPARE FROM──host_variable──────────────────────────────────────────────►◄

►►──SETTING──section_variable──IN──package_spec──────────────────────────────►

►──┬──────────────────────────────────────┬──────────────────────────────────►◄
   └─USING DESCRIPTOR──descriptor_name─┘
```

Single Row Extended PREPARE

```
►►──PREPARE SINGLE ROW FROM──host_variable───────────────────────────────────►◄

►►──SETTING──section_variable──IN──package_spec──────────────────────────────►

►──┬──────────────────────────────────────┬──────────────────────────────────►◄
   └─USING DESCRIPTOR──descriptor_name─┘
```

Empty Extended PREPARE

```
►►──PREPARE FROM NULL SETTING──section_variable──IN──package_spec────────────►◄
```

Temporary Extended PREPARE

```
►►──PREPARE FROM──host_variable──FOR──section_variable───────────────────────►◄

►►──IN──package_spec─────────────────────────────────────────────────────────►◄
```

Description

host_variable
: Specifies the statement that is to be prepared. *Host_variable* is a varying-length string host variable of maximum length 8192. It does not have an associated indicator variable.

SETTING *section_variable*
: In the Basic Extended PREPARE statement, the *section_variable* is set by the database manager to an identifier for the statement that is prepared. It is used in subsequent Extended DESCRIBE, DROP STATEMENT, Extended EXECUTE, and Extended DECLARE CURSOR statements to specify the corresponding prepared statement.

In the Single Row Extended PREPARE statement, the *section_variable* is set by the database manager to an identifier for the statement that is prepared. It is used in subsequent Extended DESCRIBE, DROP STATEMENT, and Extended EXECUTE (with the OUTPUT Descriptor clause) statements to specify the corresponding prepared statement.

In the Empty Extended PREPARE statement, the *section_variable* is set by the database manager to an identifier for the indefinite section that is created. It is used in subsequent Temporary Extended PREPARE, Extended DESCRIBE, Extended EXECUTE, DROP STATEMENT and Extended DECLARE CURSOR statements to specify the corresponding section.

FOR *section_variable*
Identifies a statement defined by an Empty Extended PREPARE statement. This should be set to the value returned by the database manager as a result of the Empty Extended PREPARE statement.

IN *package_spec*
Identifies the package in which the prepared statement is to be stored. If the qualified *package_spec* does not refer to an existing package, an error will result.

USING DESCRIPTOR *descriptor_name*
Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the statement was prepared. Extended PREPARE only utilizes the following fields in an SQLDA: SQLD, SQLTYPE, SQLLEN, and, optionally, SQLNAME (for CCSID override).

`USING DESCRIPTOR` may be specified for Temporary Extended PREPARE without an error indication, but it is ignored.

Normally if a prepared statement contains parameter markers (?), an SQLDA would be provided at run time by the Extended EXECUTE or Extended OPEN statement that references that prepared statement. However, an SQLDA can be used to improve run-time performance and reduce conversions in those cases where data types and lengths are known at statement preparation time for the parameter markers in the prepared SQL statement. Another reason for providing an SQLDA at statement preparation time is to override the restrictions on the use of parameter markers as outlined under “Rules for parameter markers” under the PREPARE statement. Also, if an SQLDA is not provided at statement preparation time, it is assumed that none of the variables used within predicates are nullable; therefore, an error results if a negative indicator value is provided at execution time.

An input SQLDA may also be specified on a subsequent Extended EXECUTE or Extended OPEN; in such cases, if the information does not match that of the PREPARE SQLDA, errors may result.

The fields described in the SQLDA should match the parameter markers (?) in the statement being prepared. If there are fewer fields specified in the SQLDA, an error will result. If there are more fields specified in the SQLDA, they will be ignored.

Before the Extended PREPARE statement is processed, the user must set the fields in the SQLDA described in the “Description” section of “EXECUTE” on page 264 and Table 20 on page 360.

The **Basic Extended PREPARE** form of the Extended PREPARE statement adds an SQL statement to an existing package. If the package is new, the Extended PREPARE statement must be preceded by a CREATE PACKAGE statement. Existing packages, created using the MODIFY option of CREATE PACKAGE, can be extended using this format of the PREPARE statement.

The USING DESCRIPTOR clause must be used when preparing a statement that contains parameter markers, if using the DRDA protocol.

The **Single Row Extended PREPARE** form of the Extended PREPARE statement indicates that the select-statement contained in the *host_variable* is a single row Select. Select-statements prepared using "PREPARE SINGLE ROW" must be processed using the Extended EXECUTE with OUTPUT DESCRIPTOR command.

The Single Row Extended PREPARE form of the Extended PREPARE statement is not supported with the DRDA protocol.

The **Empty Extended PREPARE** form of the Extended PREPARE statement allows for the creation of an indefinite section in a program. The section is subsequently used when a statement is dynamically prepared using a Temporary Extended PREPARE statement.

This format of the Extended PREPARE must follow the CREATE PACKAGE...USING NOMODIFY... format of the CREATE PACKAGE statement and must exist in the same logical unit of work as the CREATE PACKAGE statement.

If the above restriction is violated, execution of the statement will be unsuccessful.

The **Temporary Extended PREPARE** form of the Extended PREPARE statement prepares the statement contained in the created indefinite section. This section must have been created by an Empty Extended PREPARE statement. The section number for this section is contained in the *section_variable.*

This format of the Extended PREPARE may not be processed in a logical unit of work in which update to the package is already in progress. If the above restriction is violated, execution of the statement will be unsuccessful.

See "Rules for statement strings", "Parameter Markers", and "Rules for parameter markers" on page 314 for a list of the SQL statements which may be contained in the *host_variable* and the rules for using parameter markers in the *host_variable*.

Notes

The various formats to the Extended PREPARE statement permit statements to be created for different programs in different logical units of work.

Because a DBA can add a statement to a package on behalf of the owner (creator) of the module, where the owner is not authorized for the added function, the DBA should grant the proper authorization to the owner.

Examples

Example of **Basic Extended PREPARE**

```
PREPARE FROM :XSTRING SETTING :STMID
        IN :USERID.:PACKNAME USING DESCRIPTOR MYSQLDA
```

Example of **Single Row Extended PREPARE**

```
PREPARE SINGLE ROW FROM :XSTRING SETTING :STMID
        IN :USERID.:PACKNAME USING DESCRIPTOR MYSQLDA
```

Example of **Empty Extended PREPARE**

```
PREPARE FROM NULL SETTING :STMID
        IN :USERID.:PACKNAME
```

Example of **Temporary Extended PREPARE**

```
PREPARE FROM :XSTRING FOR :STMID
        IN :USERID.:PACKNAME
```

PUT

The PUT statement inserts a row into a table. It is most often used when blocking is in effect in order to create a block of rows to be inserted into a table at one time and thus improve performance.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

For an explanation of the authorization required to use a cursor, see “DECLARE CURSOR” on page 235.

Syntax

```
►►──PUT──cursor_name──┬─────────────────────────────────────┬──────────────►◄
                      ├─FROM────host_variable_list──────────┤
                      └─USING DESCRIPTOR──descriptor_name───┘
```

Description

cursor_name
Is an ordinary identifier that identifies the insert cursor to be used in the PUT operation. The *cursor_name* must identify a declared cursor as explained in “DECLARE CURSOR” on page 235. When the PUT statement is processed, the cursor must be in the open state.

FROM
This is only used in a PUT statement that is used in conjunction with a dynamic INSERT statement, in which case either FROM or USING DESCRIPTOR is required.

Introduces a list of host variables, host structure, or both, whose values are substituted for the parameter markers (question marks) in the dynamically-prepared INSERT statement. (For an explanation of parameter markers, see “PREPARE” on page 313.)

host_variable_list
Identifies a list of host variables, host structures, or both, that must be declared in the program in accordance with the rules for declaring host variables and host structures.

The total number of host variables and host structure subfields must be the same as the number of parameter markers in the prepared statement. The *n*th variable or subfield corresponds to the *n*th parameter marker in the prepared statement.

USING DESCRIPTOR *descriptor_name*
This is only used in a PUT statement that is used in conjunction with a dynamic INSERT statement, in which case either FROM or USING DESCRIPTOR is required.

Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the INSERT statement was prepared.

Before the PUT statement is processed, the user must set the fields in the SQLDA described in the "Description" section of "EXECUTE" on page 264 and Table 20 on page 360.

Notes

When blocking is used, every time a PUT statement is processed, a single row of data is added to an *insert-block*. Rows are not inserted into the database until the block is full, or, until a CLOSE statement is processed. The PUT statement can also be processed when blocking is not in effect. In this case, one data row is inserted directly into a table.

Insert blocking is available with the DRDA protocol if the application has been preprocessed with the BLOCK option.

The database manager does not notify your program of an insert error until the PUT that fills a block is processed. To determine when (or if) rows are actually inserted into the database, your program should examine SQLERRD(3) in the SQLCA when doing PUTs.

For example, suppose that 10 data rows to be inserted fit into one block, and that the data for the fourth insert is in error. PUTs 1 through 9 have successful SQLCA notifications, even though the insert for the fourth PUT has an error. On the tenth PUT, the block is full. The database manager tries to process the block of ten inserts, but encounters the error in the fourth row. It stops processing the block - that is, three rows are inserted successfully. SQLERRD(3) contains the number of rows that were successfully inserted. In this case, it contains a value of 3. If all rows were inserted successfully, it would contain 10. You can use SQLERRD(3) to determine where the error occurred.

Examples

Example 1

This example of statements from a PL/I program illustrates the use of a PUT statement with a static INSERT statement. The host variables EMPNO, FIRSTNME, MIDINIT, LASTNAME and EDLEVEL are compatible with the columns by the same name in the EMPLOYEE table. In this program, cursor PUTCUR inserts blocks of skeleton rows into the EMPLOYEE table.

```
EXEC SQL  DECLARE PUTCUR CURSOR FOR
            INSERT INTO EMPLOYEE  (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL)
              VALUES (:EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, :EDLEVEL);

EXEC SQL  OPEN PUTCUR;

... /* code to start a loop */
  ... /* code to pick up values and assign them to host variables */
  EXEC SQL  PUT PUTCUR;
... /* code to end a loop */


EXEC SQL  CLOSE PUTCUR;
```

Example 2: Similar to example 1, except that it uses a PUT statement with a dynamic INSERT statement.

```
EXEC SQL  PREPARE INSERT_STMT FROM
             'INSERT INTO EMPLOYEE  (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL)
              VALUES (? ? ? ? ?)';

EXEC SQL  DECLARE PUTCUR CURSOR FORINSERT_STMT;
EXEC SQL  OPEN PUTCUR;

... /* code to start a loop */
  ... /* code to pick up values and assign them to host variables  */
      /* and to the three subfields FIRSTNME, MIDINIT, LASTNAME    */
      /* of host structure EMPNAME.                                */
  EXEC SQL  PUT PUTCUR FROM :EMPNO, :EMPNAME, :EDLEVEL;
... /* code to end a loop */

EXEC SQL  CLOSE PUTCUR;
```

Extended PUT

The Extended PUT statement inserts a row into a table. It is most often used when blocking is in effect in order to create a block of rows to be inserted into a table at one time and thus improve performance. The cursor must have been opened with an Extended OPEN.

Invocation

This statement can only be embedded in an application program written in Assembler or REXX.

Authorization

The authorization ID of the statement must have one of the following:
- ownership of the package
- DBA authority
- EXECUTE privilege on the package.

Syntax

```
►►──PUT──cursor_variable──┬────────────────────────────────────┬──────────►◄
                          │        ┌─,───────────┐             │
                          │        ▼             │             │
                          ├─FROM────host_variable─┴────────────┤
                          └─USING DESCRIPTOR──descriptor_name──┘
```

Description

cursor_variable
: Identifies the insert cursor that is to be used. The cursor must have been defined by a preceding Extended DECLARE CURSOR statement in the same logical unit of work.

FROM *host_variable*,**...**
: Identifies variables in the program that will be used to provide the values that are to be inserted with the Extended PUT. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

USING DESCRIPTOR *descriptor_name*
: Identifies an input SQLDA structure that provides information concerning input variables that were specified as parameter markers (?) when the statement was prepared.

  Before the Extended PUT statement is processed, the user must set the fields in the SQLDA described in the “Description” section of “EXECUTE” on page 264 and Table 20 on page 360.

The indicated cursor must be declared and opened.

Notes

In most respects, the Extended PUT statement is identical to the PUT statement (see “PUT” on page 322); however, in the Extended PUT statement, the *cursor_variable* is a host variable. This feature makes it possible for a user to provide the cursor name when the program is run and to enter a PUT statement in a logical unit of work or program other than the one in which the statement was prepared. Extended DECLARE CURSOR, OPEN, and PUT must occur in the same logical unit of work.

Examples

```
PUT :CURSOR1 FROM :X, :Y

PUT :CURSOR2 USING DESCRIPTOR SQLDA
```

REVOKE (Package Privileges)

This form of the REVOKE statement revokes the privilege to process statements in a package.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

This authorization ID must previously have granted the specified privileges to every *authorization_name* (or PUBLIC) specified in the FROM clause.

Note that someone with DBA authority can indirectly revoke the EXECUTE privilege on a package by obtaining the owner′s password from the SYSTEM.SYSUSERAUTH catalog table and then connecting as the owner.

Syntax

```
                                                  ┌─,──────────────────┐
                  (1)                             ▼                    │
►►──REVOKE EXECUTE──────ON──package_name──FROM──┬─authorization_name─┴──────►◄
                                                │        (2)         │
                                                └─PUBLIC─────────────┘
```

Notes:

1 RUN can be used as a synonym for EXECUTE and is provided for compatibility with previous versions of SQL/DS.

2 PUBLIC is specified only once.

Description

EXECUTE ON *package_name*
Identifies the package from which the EXECUTE privilege is being removed. The *package_name* must identify a package that exists at the application server.

FROM *authorization_name***,...**
Identifies the user from whom the privilege is revoked. *authorization_name*,... is a list of one or more authorization IDs. Do not use the same *authorization_name* more than once.

You cannot use the *authorization_name* of the REVOKE statement itself. (You cannot revoke privileges from yourself.)

PUBLIC
Revokes the privilege from PUBLIC.

Examples

All users currently have the right to process the TREMAR package. PAYROLL, HANNA, and TREVOR have explicitly been granted this privilege. The other users have it because a GRANT EXECUTE TO PUBLIC statement was previously processed.

Remove the right to process the package from all users but PAYROLL.

```
REVOKE EXECUTE ON TREMAR FROM HANNA, PUBLIC, TREVOR
```

REVOKE (System Authorities)

This form of the REVOKE statement allows a user having DBA authority to revoke authorities from other users.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The authorization ID of the statement must have DBA authority.

Syntax

```
                                  ┌─,───────────────────────┐
                                  ▼                         │
►►──REVOKE──┬──CONNECT FROM───┬─authorization_name──┬──┴──┬────────────────────►◄
            │                 │              (1)    │     │
            │                 └─ALLUSERS────────────┘     │
            │                         ┌─,──────────────────┐ │
            │                         ▼                    │ │
            ├─┬─DBA──────┬──FROM───authorization_name──┴─┤
            │ └─RESOURCE─┘                                  │
            └─SCHEDULE FROM──subsystemid────────────────────┘
```

Notes:

1 ALLUSERS can only be specified once.

Description

CONNECT
Revokes CONNECT authority from the specified *authorization_names*. Revoking CONNECT causes all authorities to be revoked with it and the *authorization_name* to be deleted from the catalog SYSUSERAUTH.

Revoking CONNECT does not cause objects owned by that *authorization_name* to be dropped. Neither does it cause table privileges for that *authorization_name* to be revoked. A user with DBA authority can later drop the objects and revoke the privileges.

DBA
Revokes DBA authority from the specified *authorization_name*s. A user having DBA authority cannot revoke any authority from himself or herself. Revoking DBA authority automatically causes all authorities to be revoked except CONNECT.

RESOURCE
Revokes RESOURCE authority from the specified *authorization_name*s. No one can revoke RESOURCE authority from a user that has DBA authority. Revoking RESOURCE authority implies no other revocations.

FROM
Introduces a list of one or more *authorization_names*.

authorization_name
An authorization ID.

ALLUSERS
Specifies that implicit CONNECT authority is to be revoked for all system-defined users.

VSE Users
ALLUSERS is not a valid option because implicit CONNECT authority is not applicable to VSE application servers.

SCHEDULE
Allows the DBA to revoke access by a CICS subsystem. Used with the VSE Guest sharing facility of the DB2 Server for VM product. For more information see the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

FROM *subsystemid*
Is the subsystem ID of the CICS subsystem running under the VSE guest.

Notes

If you enter REVOKE for an authority that the user does not have, the revocation is ignored for that authority.

Examples

Example 1

Given that VEILLEUX, MARINA, and HEARST have DBA authority, enter the statements necessary to revoke all authority from VEILLEUX. Leave MARINA with only CONNECT authority and leave HEARST with both CONNECT and RESOURCE authority.

```
REVOKE DBA FROM VEILLEUX, MARINA, HEARST

REVOKE CONNECT FROM VEILLEUX

GRANT RESOURCE TO HEARST
```

Example 2

All users have previously been granted implicit connect authority from their VM user ID. PAYROLL, HANNA, and TREVOR have explicitly been granted this authority. The other users have it because a GRANT CONNECT TO ALLUSERS statement was previously processed.

Remove implicit connect authority from all users but PAYROLL.

```
REVOKE CONNECT FROM HANNA, TREVOR, ALLUSERS
```

VSE Users
Example 2 does not apply to VSE.

REVOKE (Table Privileges)

This form of the REVOKE statement revokes privileges on the table or view.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

This authorization ID must previously have granted the specified privileges to every *authorization_name* (or PUBLIC) specified in the FROM clause.

Note that someone with DBA authority can indirectly revoke privileges on a table or view by obtaining the owner's password from the SYSTEM.SYSUSERAUTH catalog table and then connecting as the owner.

Syntax

```
                      ┌─PRIVILEGES─┐
►►──REVOKE───┬─ALL─┴────────────┴─────┬──ON──┬─table_name─┬──────────────────────►
             │   ┌─,──────────────────┐ │      └─view_name──┘
             │   ↓        (1)         │ │
             └─────┬─ALTER────────────┬─┴─┘
                   ├─DELETE───────────┤
                   │        (1)       │
                   ├─INDEX────────────┤
                   ├─INSERT───────────┤
                   │             (1)  │
                   ├─REFERENCES───────┤
                   ├─SELECT───────────┤
                   └─UPDATE───────────┘


          ┌─,───────────────────────┐
          ↓                         │
►─FROM────┬─authorization_name─┬────┴─────────────────────────────────────────►◄
          │        (2)         │
          └─PUBLIC─────────────┘
```

Notes:

1 The ALTER, INDEX, and REFERENCES options are not applicable to views.

2 PUBLIC may only be specified once per statement.

Description

ALL or **ALL PRIVILEGES**
Revokes table privileges on the table or view identified in the ON clause. The privileges revoked are those possessed by the authorization ID of the REVOKE statement. ALL PRIVILEGES is the default.

ALTER
Revokes the privilege to use the ALTER TABLE statement. This privilege does not apply to views.

DELETE
Revokes the privilege to use the DELETE statement.

INDEX
Revokes the privilege to use the CREATE INDEX statement. This privilege does not apply to views.

INSERT
Revokes the privilege to use the INSERT statement.

REFERENCES
Revokes the privilege to either create referential constraints or to change existing referential constraints. This privilege does not apply to views.

SELECT
Revokes the privilege to use the SELECT statement or the CREATE VIEW statement.

UPDATE
Revokes the privilege to use the UPDATE statement. Note that a list of column names can be used only with GRANT, not with REVOKE. You must therefore revoke UPDATE on all columns.

ON *table_name*
ON *view_name*
Identifies the table or view from which the privileges are being revoked. The *table_name* or *view_name* must identify a table or view that exists at the application server.

FROM *authorization_name***,...**
Identifies from whom the privileges are revoked. *authorization_name*,... is a list of one or more authorization IDs.

You cannot use the *authorization_name* of the REVOKE statement itself. (You cannot revoke privileges from yourself.)

PUBLIC
Revokes a grant of privileges to PUBLIC.

Dependent Privileges

When a privilege is revoked from a user, every privilege dependent on that privilege is also revoked.

A privilege P2 possessed by user U2 is dependent on privilege P1 possessed by user U1 if all of these are true:
- P1 and P2 are the same privilege.
- U1 granted the privilege to U2.
- No other user granted the same privilege to U2 before U1 granted it.

Also, table privilege P2 is dependent on table privilege P1 if P2 was derived from P1 as a result of a CREATE VIEW statement.

Revoking a privilege that was used to create a package invalidates the package.

Multiple Grants: If you granted the same privilege to the same user more than once, revoking that privilege from that user negates all those grants. It does not negate any grant of that privilege made by others.

If a user has more than one source for a privilege, that privilege is not revoked until it is revoked by all sources (see example 2 below).

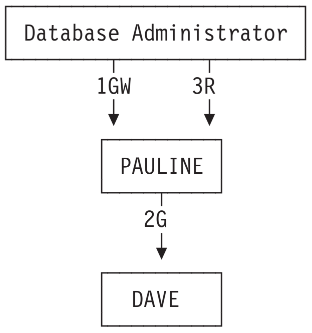
Notes

The only way to revoke the WITH GRANT OPTION is to revoke the privilege itself and then to grant it again without the WITH GRANT OPTION.

Examples

Example 1

This example shows the effect of revoking a privilege that has a dependent privilege. To illustrate this process, the diagram that follows shows a sequence of GRANT and REVOKE statements.



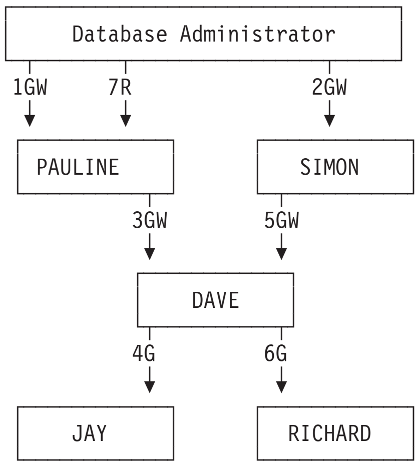
The statements illustrated in the above diagram are:

```
  1GW) from DBA:        GRANT SELECT ON TBLA TO PAULINE WITH GRANT OPTION

  2G)  from PAULINE:    GRANT SELECT ON TBLA TO DAVE

  3R)  from DBA:        REVOKE SELECT ON TBLA FROM PAULINE
```

Following this sequence of statements neither PAULINE nor DAVE has the SELECT privilege on TBLA. The explicit revoking of PAULINE's privilege implicitly revokes DAVE's as well.

Example 2

This extends example 1 in order to show the effect of having received a privilege from more than one source.



Following this sequence of statements from the users indicated:

```
1GW) from DBA:       GRANT SELECT ON TBLA TO PAULINE WITH GRANT OPTION

2GW) from DBA:       GRANT SELECT ON TBLA TO SIMON   WITH GRANT OPTION

3GW) from PAULINE    GRANT SELECT ON TBLA TO DAVE    WITH GRANT OPTION

4G)  from DAVE:      GRANT SELECT ON TBLA TO JAY

5GW  from SIMON:     GRANT SELECT ON TBLA TO DAVE    WITH GRANT OPTION

6G)  from DAVE:      GRANT SELECT ON TBLA TO RICHARD

7R)  from Admin:     REVOKE SELECT ON TBLA FROM PAULINE
```

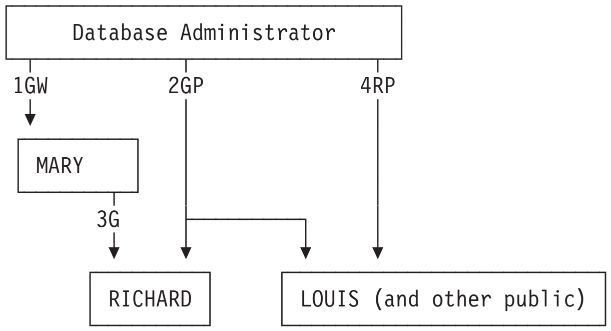
PAULINE loses her SELECT privilege on TBLA, but DAVE retains his (having obtained it from SIMON as well).

JAY loses his SELECT privilege because he obtained it from DAVE at a time when DAVE had only obtained the SELECT WITH GRANT privilege from PAULINE.

RICHARD retains his SELECT privilege because he obtained it from DAVE at a time when DAVE had obtained the SELECT WITH GRANT privilege from both PAULINE and SIMON.

Example 3

This example shows how the revocation of a PUBLIC privilege varies depending on whether: that privilege was granted specifically to that user **or** that privilege was obtained using a GRANT TO PUBLIC.



Following this sequence of statements from the users indicated:

```
1GW) from DBA:        GRANT SELECT ON TBLA TO MARY WITH GRANT OPTION

2GP) from DBA:        GRANT SELECT ON TBLA TO PUBLIC

3G)  from MARY:       GRANT SELECT ON TBLA TO RICHARD

4RP) from DBA:        REVOKE SELECT ON TBLA FROM PUBLIC
```

RICHARD retains the SELECT privilege on TBLA even though he was originally granted it as a member of the public. LOUIS only had the SELECT privilege as a member of the public, so loses that privilege.

ROLLBACK

The ROLLBACK statement ends a logical unit of work and back out the database changes that were made by that logical unit of work.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

Syntax

```
                  ┌─WORK─┐
►►──ROLLBACK──────┴──────┴──┬─────────┬──────────────────────────────────────►◄
                            └─RELEASE─┘
```

Description

RELEASE
Re-establishes the default user ID and default database for a subsequent logical unit of work. If this default user ID had been overridden with an explicit CONNECT, in the terminating logical unit of work that explicitly established user ID is replaced by the default user ID. By not specifying RELEASE, the user ID and database at termination of the logical unit of work are retained for a subsequent logical unit of work. For VSE interactive users connected to a remote DRDA application server, when the next SQL statement is entered, you are automatically connected with your CICS signon user ID to the same application server.

ROLLBACK terminates the logical unit of work in which ROLLBACK is processed. All changes made by the following statements during a logical unit of work, are backed out:

ACQUIRE DBSPACE
ALTER DBSPACE
ALTER PROCEDURE
ALTER PSERVER
ALTER TABLE
COMMENT ON
CREATE INDEX
CREATE PROCEDURE
CREATE PSERVER
CREATE SYNONYM
CREATE TABLE
CREATE VIEW
DELETE
DROP
DROP PROCEDURE
DROP PSERVER
EXPLAIN
GRANT Package Privileges
GRANT System Authorities

GRANT Table/View Privileges
INSERT
LABEL ON
PUT
REVOKE Package Privileges
REVOKE System Authorities
REVOKE Table/View Privileges
UPDATE
UPDATE STATISTICS

All locks acquired by the logical unit of work are released. All cursors that were opened during the logical unit of work are closed. All statements that were prepared during the logical unit of work are destroyed. Any cursors associated with a prepared statement that is destroyed cannot be opened until the statement is prepared again.

Notes

If a COMMIT or ROLLBACK does not immediately precede the termination of an application process, the database manager attempts to commit the work (it may, however, not always be successful). It is strongly recommended that each application process explicitly ends its logical unit of work before terminating.

ROLLBACK should not be issued after a severe error has occurred (one which sets the SQLWARN0 field in the SQLCA to 'S'). In this situation, the only statement that can be issued is a CONNECT statement to another application server.

The logical unit of work must be completed by using the COMMIT or ROLLBACK statements before the CONNECT statement can be used to switch to another user ID or application server.

TCP/IP does not perform any security checking during a physical connect. The Batch application requester will use the DRDA security handshaking flows during the logical connect to perform user ID and password verification. The physical TCP/IP connection will be deallocated and reallocated whenever the application switches to a different user ID or server name (using the CONNECT statement), and DRDA security handshaking flows will be used again during the logical connect. Either of these switches will not require the application to issue a COMMIT RELEASE or ROLLBACK RELEASE. The Batch Resource Adapter will retain and use the current user ID, password, and server name (unless different ones are specified with a new CONNECT statement) after the new TCP/IP physical connection is established. If a COMMIT RELEASE or ROLLBACK RELEASE was issued prior to a CONNECT statement, then all user ID, password and server name information is lost and must be supplied with the next CONNECT.

Examples

The PL/I program in “COMMIT” on page 182 illustrates how the ROLLBACK statement is used.

SELECT INTO

The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables. If the table is empty, the statement assigns +100 to SQLCODE and '02000' to SQLSTATE and does not assign values to the host variables. If more than one row satisfies the search condition, statement processing is terminated and an error occurs.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

In Fortran, REXX, and programs prepared using extended dynamic SQL, SELECT INTO cannot be used with the DRDA protocol.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- DBA authority, or
- For each table or view identified in the SELECT INTO statement:
  - The SELECT privilege on the table or view, or
  - Ownership of the table or view.

Syntax

```
►►──select_clause──INTO──host_variable_list──from_clause──┬──────────────┬──────►
                                                          └─where_clause─┘

►──┬─────────────┬───────────────────────────────────────────────────────────►◄
   └─with_clause─┘
```

Description

The result table is derived by evaluating the *from_clause*, *where_clause*, and *select_clause*, in this order.

See Chapter 5, “Queries,” on page 121 for a description of the *select_clause*, *from_clause*, and *where_clause*.

INTO
Introduces a list of host variables, host structures, or both.

host_variable_list
Identifies a list of host variables, host structures, or both, that must be declared in the program in accordance with the rules for declaring host variables and host structures.

The first value in the result row is assigned to the first *host_variable* or host structure subfield in the list, the second value to the second variable, and so on. If the number of host variables and host structure subfields is less than the number of select_list values, the value W is assigned to the SQLWARN3 field of the SQLCA. (See “SQL Communication Area (SQLCA)” on page 353.) Note that there is no warning if there are more variables than the number of select_list values. For a datetime value, the variable must be a character string variable of a minimum length as defined in Chapter 3.

If the value is null, an indicator variable must be specified.

Each assignment to a variable is made according to the rules described in Chapter 3.

```
►►──WITH──┬─RR─┬──────────────────────────────────────────────────────────────►◄
          ├─CS─┤
          └─UR─┘
```

WITH
Specifies the isolation level at which the statement is executed.

RR
Repeatable read

CS
Cursor stability

UR
Uncommitted read

If an error occurs, no value is assigned to the host variable or to variables later in the list, though any values that have already been assigned to variables remain assigned.

If an error occurs because the result table has more than one row, values may or may not be assigned to the host variables. If values are assigned to the host variables, the row that is the source of the values is undefined and not predictable.

See the *DB2 Server for VSE & VM Application Programming* manual for a description of the possible errors when SELECT INTO is processed.

Examples

Example 1

Using a COBOL program statement, put the maximum salary (SALARY) from the EMPLOYEE table into the host variable MAX-SALARY (dec(9,2)).

```
EXEC SQL  SELECT MAX(SALARY)
            INTO :MAX-SALARY
            FROM EMPLOYEE
END-EXEC.
```

Example 2

Using a PL/I program statement, select the row from the EMPLOYEE table with a employee number (EMPNO) value the same as that stored in the host variable HOST_EMP char(6)). Then put the first name (FIRSTNME) and last name (LASTNAME) into the host structure HOST_NAME, and education level (EDLEVEL) into the host variable HOST_EDUCATE (integer) from that row.

```
EXEC SQL  SELECT FIRSTNME, LASTNAME, EDLEVEL
            INTO :HOST_NAME, :HOST_EDUCATE
            FROM EMPLOYEE
            WHERE EMPNO = :HOST_EMP;
```

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of its base table.

There are two forms of this statement:
- The *Searched* UPDATE form updates zero or more rows (optionally determined by a search condition).
- The *Positioned* UPDATE form updates exactly one row (as determined by the current position of a cursor).

Invocation

A Searched UPDATE statement can be embedded in an application program or issued interactively. A Positioned UPDATE must be embedded in an application program. Both Searched UPDATE and Positioned UPDATE are executable statements that can be dynamically prepared.

A Positioned UPDATE in Fortran, and programs prepared using extended dynamic SQL cannot be used with the DRDA protocol.

Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:
- Ownership of the table
- The UPDATE privilege for the table or columns in the table or view
- DBA authority.

The UPDATE privilege on a view is only inherent in DBA authority. Ownership of a view does not necessarily include the UPDATE privilege on the view because the privilege may not have been granted when the view was created, or it may have been granted, but subsequently revoked.

If the *search_condition* includes a subquery, the privileges designated by the authorization ID of the statement must also include at least one of the following:
- Ownership of the tables or views identified in the subquery
- The SELECT privilege on every table or view identified in the subquery
- DBA authority.

Syntax

Searched UPDATE:

```
►►──UPDATE──┬─table_name─┬──┬──────────────────┬──────────────────────────────►
            └─view_name──┘  └─correlation_name─┘


      ┌─,──────────────────────────────┐
      ▼                                │
►─SET───column_name── = ──┬─expression─┬──┬─────────────────────────┬─────────►
                          └─NULL───────┘  └─WHERE──search_condition─┘

►──┬─────────────────┬───────────────────────────────────────────────────────►◄
   └─WITH──┬─RR─┬────┘
           └─CS─┘
```

Positioned UPDATE:

```
                                    ┌─,──────────────────────────────┐
                                    ▼                                │
►►──UPDATE──┬─table_name─┬──SET─────column_name── = ──┬─expression─┬──────────►
            └─view_name──┘                            └─NULL───────┘

►─WHERE CURRENT OF──cursor_name──────────────────────────────────────────────►◄
```

Description

table_name or view_name
Identifies the table or view to be updated. The name must identify a table or view that exists at the application server, but must not identify a catalog table, a view of a catalog table, or a read-only view. For an explanation of read-only views, see "CREATE VIEW" on page 231.

Note: Someone with DBA authority may update rows from a few of the catalog tables. See "Updateable Columns" on page 371.

correlation_name
Can be used within *search_condition* to designate the table or view. (For an explanation of *correlation_name*, see "Correlation Names" on page 64.)

SET
Introduces a list of column names and values.

column_name
Identifies a column to be updated. The *column_name* must identify a column of the specified table or view, but must not identify a view column derived from a scalar function, constant, or expression. The column names must not be qualified, and a column must not be specified more than once.

For a Positioned UPDATE, allowable column names can be further restricted to those in a certain list. This list appears in the UPDATE clause of the select statement for the associated cursor. The column names need not be in the *select-list* of the select statement for the associated cursor If the select statement is dynamically prepared, the UPDATE clause must always be present. Otherwise, the clause can be omitted under the conditions described in “The NOFOR Option” on page 239.

A view column derived from the same column as another column of the view can be updated, but both columns cannot be updated in the same UPDATE statement.

expression or **NULL**
Indicates the new value of the column. The *expression* is any expression of the type described in Chapter 3. It must not include a column function. NULL specifies the null value.

A *column_name* in an expression must name a column of the named table or view. For each row that is updated, the value of the column in the expression is the value of the column in the row before the row is updated.

If the *column_name* on the left hand side of the SET identifies a long string column, the only type of expression allowed is a host-variable.

WHERE
Specifies the rows to be updated. You can omit the clause, give a search condition, or name a cursor. If the clause is omitted, all rows of the table or view are updated.

search_condition
Is any search condition described in Chapter 3. Each *column_name* in the search condition, other than in a subquery, must name a column of the table or view. The search condition must not include a subquery where the base object of both the UPDATE and the subquery is the same table.

The *search_condition* is applied to each row of the table or view and the updated rows are those for which the result of the *search_condition* is true.

If the search condition contains a subquery, the subquery can be thought of as being processed each time the search condition is applied to a row, and the results used in applying the search condition. In actuality, the subquery is processed for each row only if it contains a correlated reference to a column of the table or view.

WITH
Specifies the isolation level used when locating the rows to be updated by the statement.

RR
Repeatable read

CS
Cursor stability

The default isolation level of the statement is the isolation level of the package. WITH can only be specified on a SEARCHED update; it is incompatible with the WHERE CURRENT OF clause.

CURRENT OF *cursor_name*
Identifies the cursor to be used in the update operation. The *cursor_name* must identify a declared cursor as explained in “DECLARE CURSOR” on page 235. The *cursor_name* can be a delimited identifier. If *cursor_name* is a reserved word, it must be a delimited identifier.

The table or view specified must also be identified in the FROM clause of the select-statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR" on page 235.)

When the UPDATE statement is processed, the cursor must be positioned on a row and that row is updated.

Update values must satisfy the following rules. If they do not, or if any other errors occur during the execution of the UPDATE statement, no rows are updated.

- *Assignment*:

  Update values are assigned to columns under the assignment rules described in Chapter 3.
- *Validity*:

  If the identified table, or the base table of the identified view, has one or more unique indexes, each row updated in the table must conform to the constraints imposed by those unique indexes.

  In the case of a multiple-row update of a unique key, the uniqueness constraint is effectively checked at the end of the operation.

  If a view is used that is defined using the WITH CHECK OPTION, each updated row must conform to the definition of the view. If a view is used that is not defined using WITH CHECK OPTION, rows can be changed so that they no longer conform to the definition of the view. Such rows are updated in the base table of the view and no longer appear in the view.

  If a view is used that is dependent on other views whose definitions include WITH CHECK OPTION, the updated rows must also conform to the definition of those views.
- *Referential Integrity*:

  The value of the primary key in a parent row must not be changed by a Positioned UPDATE. A primary key value may be changed using a Searched UPDATE if there are no rows that are dependent on the old key value and if the new value of the primary key is unique. A non-null update value of a foreign key must be equal to a value of the primary key of the parent table of the relationship.

When an UPDATE statement completes execution, the value of SQLERRD(3) in the SQLCA is the number of rows updated. (For a description of the SQLCA, see "SQL Communication Area (SQLCA)" on page 353.)

Differences Between Searched Updates in Recoverable and Non-Recoverable Storage Pools

Recoverable Storage Pool: Uniqueness is checked after all rows are updated.

Non-Recoverable Storage Pool: When multiple-row updates are performed against a column that has a unique index, the database manager is sensitive to the order (ascending or descending) of the data. Since the database manager automatically creates a unique index on a primary key column, a Searched UPDATE cannot be used to perform multiple-row updates against the primary key column. This is to ensure that updates to the primary key are independent of the order of the data. For the same reason, a Positioned UPDATE cannot be used to update primary key columns.

Locking: Unless appropriate locks already exist, one or more exclusive locks are acquired by the execution of a successful UPDATE statement. Until the locks are released, the updated row can only be accessed by the application process that performed the update. For further information on locking, see the descriptions of the COMMIT, ROLLBACK, LOCK TABLE, and LOCK DBSPACE statements.

Blocking: The blocking options, SBLocK or BLocK, in the SQLPREP command and the CREATE PACKAGE statement improves performance as they insert and retrieve rows in groups. However, if a program was preprocessed with the NOFOR option, query cursors referenced in Positioned UPDATE statements are unavailable for blocking. If a Positioned UPDATE is coded in a program and NOFOR is not in effect, then a FOR UPDATE OF clause must be included in the select-statement. See the *DB2 Server for VSE & VM Application Programming* manual for more information on blocking when preprocessing and running a program.

Error Conditions: It is possible for an error to occur that makes the state of the cursor unpredictable. If an error occurs during the execution of a Positioned UPDATE that makes the position of a cursor unpredictable, the cursor is closed.

If an error occurs during the execution of a Searched UPDATE, you must inspect SQLWARN6 to determine the extent of the error. The following are the current settings of SQLWARN6 along with possible responses:

1. SQLWARN6 is set to 'S'. A severe error has occurred, leaving the system in an unusable state.
   - No further requests are possible. The application must end, or, in a DB2 Server for VSE & VM environment, may switch to another database.
2. SQLWARN6 is set to 'W'. An error occurred causing the LUW to be rolled back automatically. The system is still in a usable state. The application can:
   - begin a new LUW and proceed
   - stop.
3. SQLWARN6 is blank. An error has occurred, but the LUW is still active. Any changes made by the request have been rolled back, hence the failing request has not left any partial results in the database. The application can:
   - continue forward processing of the LUW
   - commit the changes made before the failing request
   - roll back the LUW.

Examples

Example 1

Change the job (JOB) of employee number (EMPNO) ‘000290’ in the EMPLOYEE table to ‘LABORER’.

```
UPDATE EMPLOYEE
  SET JOB = 'LABORER'
  WHERE EMPNO = '000290'
```

Example 2

Increase the project staffing (PRSTAFF) by 1.5 for all projects that department (DEPTNO) ‘D21’ is responsible for in the PROJECT table.

```
UPDATE PROJECT
  SET PRSTAFF = PRSTAFF + 1.5
  WHERE DEPTNO = 'D21'
```

Example 3

All the employees except the manager of department (WORKDEPT) ‘E21’ have been temporarily laid off. Indicate this by changing their job (JOB) to NULL and their pay (SALARY, BONUS, COMM) values to zero in the EMPLOYEE table.

```
UPDATE EMPLOYEE
  SET JOB=NULL, SALARY=0, BONUS=0, COMM=0
  WHERE DEPTNO = 'E21'
  AND JOB <> 'MANAGER'
```

Example 4

In a PL/I program display the rows from the EMPLOYEE table and then, if requested to do so, change the job (JOB) of certain employees to the new job keyed in.

```
EXEC SQL  DECLARE C1 CURSOR FOR
            SELECT *
              FROM EMPLOYEE
              FOR UPDATE OF JOB;

EXEC SQL  OPEN C1;

EXEC SQL  FETCH C1 INTO ...     ;

PUT ...      ;
GET LIST (CHANGE, NEWJOB);
IF CHANGE = 'YES' THEN
  EXEC SQL  UPDATE EMPLOYEE
              SET JOB = :NEWJOB
              WHERE CURRENT OF C1;

EXEC SQL  CLOSE C1;
```

UPDATE STATISTICS

The UPDATE STATISTICS statement causes internal statistics of tables and indexes to be updated with current information.

Invocation

This statement can be embedded in an application program, or it can be issued interactively.

Authorization

The privileges held by the authorization ID of the statement must include CONNECT authority.

Syntax

```
►►──UPDATE──┬─────┬──STATISTICS FOR──┬─TABLE──table_name──────┬─────────────────────►◄
            └─ALL─┘                  └─DBSPACE──dbspace_name──┘
```

Description

Invoking UPDATE STATISTICS can improve performance on statements that access data from tables. These statistics, contained in the catalog tables, include the table size, various index characteristics, and other information.

ALL
: Updates statistics for all columns. In the case of a column which is not a first column of any index, the column statistics are an approximation. If ALL is not specified, statistics are only updated for a column which is the first column of any index.

FOR TABLE
: Indicates the table for which you want the statistics updated. If the table name is qualified, the qualifier is the owner of the table. Otherwise, the authorization ID of the statement is the owner of the table.

  table_name
  : Identifies the table whose statistics you want updated. The name must identify a base table that exists at the application server.

FOR DBSPACE
: Updates the statistics for all tables in the designated dbspace. If the dbspace name is qualified, the qualifier is the owner of the dbspace. Otherwise, the authorization ID of the statement is the owner of the dbspace.

  dbspace_name
  : Identifies the dbspace containing the tables whose statistics you want updated. The name must identify a dbspace that exists at the application server.

Examples

This shows the statements that are embedded in a PL/I program in order to add an index on project name (PROJNAME) to the PROJECT table and to update the statistics on that table. This is so that programs using that table that are subsequently reprepared can consider those statistics when determining an access strategy.

```
EXEC SQL  CREATE INDEX PROJNAME
            ON PROJECT(PROJNAME);

EXEC SQL  UPDATE STATISTICS FOR TABLE PROJECT;
```

WHENEVER

The WHENEVER statement specifies the next host language statement to which execution will be transferred when a specified exception condition occurs.

Invocation

This statement can only be embedded in an application program. It is not an executable statement. It is not supported in REXX.

Authorization

None required.

Syntax

```
►►──WHENEVER──┬─┬─SQLERROR───┬──┬─CONTINUE─────────────────────┬─┬──────────────►◄
              │ └─SQLWARNING─┘  │     (1)                      │ │
              │                 ├─STOP─────────────────────────┤ │
              │                 └─┬─GOTO──┬──┬───┬─host_label──┘ │
              │                   └─GO TO─┘  └─:─┘               │
              └─NOT FOUND──┬─CONTINUE────────────────────────────┤
                           └─┬─GOTO──┬──┬───┬─host_label─────────┘
                             └─GO TO─┘  └─:─┘
```

Notes:

1 STOP is not valid for C, and Fortran.

Description

The SQLERROR, SQLWARNING or NOT FOUND, clause identifies the type of exception condition.

SQLERROR
Identifies any condition that results in a negative value in SQLCODE.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is 'W'), or that results in a positive value other than +100 in SQLCODE.

NOT FOUND
Identifies any condition that results in an SQLCODE of +100 and an SQLSTATE of '02000'.

The CONTINUE, GO TO, or STOP clause specifies the next statement to be processed when the identified type of exception condition exists.

CONTINUE
Causes the next sequential instruction of the source program to be processed.

GOTO *host_label*
GO TO *host_label*
Causes control to pass to the statement identified by *host_label*. For *host_label*, substitute a host identifier optionally preceded by a colon. The form of the host identifier depends on the host language. In COBOL, for example, it can be a section-name or an unqualified paragraph-name. In a Fortran program, it is an unsigned integer variable not preceded by a colon.

STOP
Causes program termination. If a logical unit of work is in progress, it is rolled back.

Notes

There are three types of WHENEVER statements:
WHENEVER SQLERROR
WHENEVER SQLWARNING
WHENEVER NOT FOUND

Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which CONTINUE is specified.

Examples

Write the statements that need to be embedded in a COBOL program in order to:
1. Go to the label HANDLER for any statement that produces an error
2. Continue processing for any statement that produces a warning
3. Go to the label ENDDATA for any statement that does not return data when expected to do so.

```
EXEC SQL  WHENEVER SQLERROR GOTO HANDLER  END-EXEC.
EXEC SQL  WHENEVER NOT FOUND GOTO ENDDATA  END-EXEC.
```

Appendix A. SQL Limits

The tables that follow describe certain limits imposed by this product.

Table 14. Identifier Length Limits

Identifier Limits	DB2 Server for VSE & VM
Longest authorization name	8
Longest constraint name	18
Longest correlation name	18
Longest cursor name	18
Longest host identifier	256 [a]
Longest long identifier	18
Longest short identifier	8
Longest server name	18
Longest statement name	18
Longest unqualified column name	18
Longest unqualified package name	8
Longest unqualified table/view/index name	18

Table 15. Numeric Limits

Numeric Limits	DB2 Server for VSE & VM
Smallest INTEGER value	-2147483648
Largest INTEGER value	+2147483647
Smallest SMALLINT value	-32768
Largest SMALLINT value	+32767
Largest decimal precision	31
Smallest FLOAT value	$-7.2\text{x}10^{75}$
Largest FLOAT value	$+7.2\text{x}10^{75}$
Smallest positive FLOAT value	$+5.4\text{x}10^{-79}$
Largest negative FLOAT value	$-5.4\text{x}10^{-79}$
Smallest REAL value	$-7.2\text{x}10^{75}$
Largest REAL value	$+7.2\text{x}10^{75}$
Smallest Positive REAL value	$+5.4\text{x}10^{-79}$
Largest Negative REAL value	$-5.4\text{x}10^{-79}$

Table 16. String Limits

String Limits	DB2 Server for VSE & VM
Maximum byte count of CHAR	254
Maximum byte count of VARCHAR	32767
Maximum character count of GRAPHIC	127
Maximum character count of VARGRAPHIC	16383

Table 16. String Limits (continued)

String Limits	DB2 Server for VSE & VM
Maximum byte count of character constant	254
Longest concatenated character string	254
Longest concatenated graphic string	127
Maximum character count of a graphic constant [b]	127

Table 17. Datetime Limits

Datetime Limits [c]	DB2 Server for VSE & VM
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 18. Database Manager Limits

Database Manager Limits	DB2 Server for VSE & VM
Most columns in a table	255
Most columns in a view	140 [d]
Maximum byte count of a row including all overhead	4080 [e]
Maximum byte count of a table [f]	32×10^9
Maximum byte count of an index [f]	32×10^9
Most rows in a table	2×10^9
Longest index key	255
Most columns in an index key	16
Most indexes on a table	255
Most tables referenced in an SQL statement or a view [g]	15
Most host variable declarations in a preprocessed program	storage
Most host variables in an SQL statement	256
Longest host variable used for insert or update	32767
Longest SQL statement	8192
Most elements in a select list	255
Most predicates in a WHERE or HAVING clause	200
Most JOIN columns	40
Maximum number of columns in a GROUP BY clause	16
Maximum total length of columns in a GROUP BY clause	255
Maximum number of columns in an ORDER BY clause	16
Maximum total length of columns in an ORDER BY clause	255
Maximum size of an SQLDA	22524
Maximum number of prepared statements	512 [h]

Table 18. Database Manager Limits (continued)

Database Manager Limits	DB2 Server for VSE & VM
Most declared cursors in a program	512 [h]
Maximum number of cursors opened at one time	storage
Most tables in a relational database	storage
Most CCSID overrides in an INSERT or SELECT statement [i]	80

Notes

a	Individual host language compilers may further restrict this. The database manager, and not the Fortran compiler, places a limit of 18 on host identifiers in Fortran programs.
b	May be further restricted by preprocessors and utilities.
c	Shown in ISO format.
d	
e	The row length of a formatted data row is 4080 bytes including overhead items such as the data value of the row, null byte, and the varchar length field. These items and others affecting the length of a row in a table are discussed in *DB2 Server for VSE & VM Database Administration*.
f	The numbers shown are architectural limits. The practical limits may be less.
g	In a complex select-statement, the number of tables that can be joined may be significantly less.
h	In C, COBOL and PL/I the sum of the number of declared cursors and the number of prepared statements that are not referenced by a cursor must not be greater than 512. In REXX, the sum of the number of declared cursors and the number of prepared statements that are not referenced by a cursor must not be greater than 40.
i	Though a table may be created with more than 80 different combinations of CCSID and datatype, insert-statement and select-statement impose a limitation of 80 CCSID overrides. For overrides above 80, use a second insert-statement or select-statement.

Appendix B. SQLCA and SQLDA

SQL Communication Area (SQLCA)

An SQLCA is a structure or a collection of variables that is updated at the end of the execution of every SQL statement. A program that contains executable SQL statements must provide either an SQLCA structure or a standalone SQLCODE field.

In all host languages except REXX, the SQL INCLUDE statement can be used to provide the declaration of the SQLCA. A similar set of variables is used for this purpose in REXX (see the *DB2 REXX SQL for VM/ESA Installation and Reference* manual for details).

In COBOL and Assembler

The name of the storage area must be SQLCA.

In PL/I and C

The name of the structure must be SQLCA. Every executable SQL statement must be within the scope of its declaration.

In Fortran

The name of the COMMON area for the INTEGER and SMALLINT variables of the SQLCA must be SQLCA1; the name of the COMMON area for the CHARACTER and VARCHAR variables must be SQLCA2.

Description of Fields

Table 19. Fields of SQLCA

Assembler, COBOL, or PL/I Name [1]	**C Name** [1]	**Fortran Name** [1]	**Data Type**	**Usage**
SQLCAID	sqlcaid	Not used	CHAR(8)	An 'eye catcher' for storage dumps, containing 'SQLCA'.
SQLCABC	sqlcabc	Not used	INTEGER	Contains the maximum length of the SQLCA: 136.
SQLCODE	sqlcode	SQLCOD	INTEGER	Contains an SQL return code.[2] **Code** — **Means** **0** — Successful execution, although SQLWARN indicators (see below) might have been set. **positive** — Successful execution, but with a warning message. **negative** — Error condition.

Table 19. Fields of SQLCA (continued)

Assembler, COBOL, or PL/I Name [1]	C Name [1]	Fortran Name [1]	Data Type	Usage
SQLERRML[3]	sqlerrml[3]	SQLTXL	SMALLINT	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.
SQLERRMC[3]	sqlerrmc[3]	SQLTXT	VARCHAR (70)	Contains one or more tokens, separated by X'FF', that are substituted for variables in the descriptions of error and warning conditions.[2] In some cases the last token appears as "FOnn". This token specifies the format number of the SQLCODE message text. "FO" represents the word "format"; "nn" identifies the version of the message that applies in this particular case. After a CONNECT statement is issued, the authorization-ID and server-name are returned.
SQLERRP	sqlerrp	SQLERP	CHAR(8)	For DRDA, after a CONNECT statement is issued, SQLERRP begins with a three-letter identifier indicating the product (DSN for DB2 for MVS, SQL for DB2 for OS/2 and DB2 for AIX, QSQ for OS/400, and ARI for DB2 Server for VSE & VM). For non-DRDA, SQLERRP begins with the three letter identifier *ARI*. If the SQLCODE field indicates an error or warning condition, this field will contain the name of the module that returned the error.
SQLERRD(1)	sqlerrd[0]	SQLERR(1)	INTEGER	Contains the Relational Data System (RDS) error code.
SQLERRD(2)	sqlerrd[1]	SQLERR(2)	INTEGER	Contains the Database Storage System (DBSS) return code.
SQLERRD(3)	sqlerrd[2]	SQLERR(3)	INTEGER	Contains the number of rows affected after INSERT, UPDATE, and DELETE. With blocking, the SQLERRD(3) associated with the final row in the block contains the number of rows in the block.

Table 19. Fields of SQLCA (continued)

Assembler, COBOL, or PL/I Name [1]	C Name [1]	Fortran Name [1]	Data Type	Usage
SQLERRD(4)	sqlerrd[3]	SQLERR(4)	INTEGER	When preprocessing a SELECT, INSERT by subselect, searched UPDATE, or searched DELETE statement, this field contains *timerons*, a short floating point value that indicates a rough relative estimate of resources required; it does not reflect an estimate of the time required. When preparing a dynamically defined SQL statement, use this value as an indicator of the relative cost of the prepared SQL statement. For a particular statement, this number can vary with changes to the statistics in the catalog. It is also subject to change between releases of DB2 Server for VSE & VM. For other conditions, the content of this field is not predictable.
SQLERRD(5)	sqlerrd[4]	SQLERR(5)	INTEGER	Following the execution of a successful DELETE statement, this field will contain the number of dependent rows affected. This includes the rows that were set to null as a result of the SET NULL rule, and the rows that were deleted as a result of the CASCADE rule. If the object table is not part of a referential structure, this field is set to zero. If processing of a datetime local exit fails, this field will contain the datetime local exit function number. This is a fullword number describing the function to be performed. Datetime local exits are discussed in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration*.
SQLERRD(6)	sqlerrd[5]	SQLERR(6)	INTEGER	Reserved for future use.
SQLWARN	sqlwarn	SQLWRN (0:10)	ARRAY	A set of indicators each containing either a blank or a setting as indicated below.
SQLWARN0	sqlwarn0	SQLWRN(0)	CHAR(1)	Blank if all other indicators are blank. Contains 'W' if at least one other indicator contains 'V', 'W', or 'Z'. Contains 'S' if SQLWARN6 is set to 'S', which overrides any 'W'.

Table 19. Fields of SQLCA (continued)

Assembler, COBOL, or PL/I Name [1]	C Name [1]	Fortran Name [1]	Data Type	Usage
SQLWARN1	sqlwarn1	SQLWRN(1)	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'Z' if, on truncation of mixed character data, the data does not follow the proper rules regarding mixed data. If both types of truncation occur, 'Z' overrides 'W'.
SQLWARN2	sqlwarn2	SQLWRN(2)	CHAR(1)	Contains 'W' if null values were eliminated from the argument of a function.
SQLWARN3	sqlwarn3	SQLWRN(3)	CHAR(1)	Contains 'W' if the number of columns in a select list is greater than the number of host variables supplied for the INTO clause of a SELECT or FETCH statement.
SQLWARN4	sqlwarn4	SQLWRN(4)	CHAR(1)	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
SQLWARN5	sqlwarn5	SQLWRN(5)	CHAR(1)	Contains 'W' if the SQL statement would cause a performance degradation.
SQLWARN6	sqlwarn6	SQLWRN(6)	CHAR(1)	Contains 'W' if the database manager was forced to end a logical unit of work. Contains 'S' when the database manager issues a severe SQLCODE; that is, one which predicates that the database manager is in an unusable state.
SQLWARN7	sqlwarn7	SQLWRN(7)	CHAR(1)	Contains 'W' if an adjustment was made to a date or timestamp value for the last day of the month. Contains 'Z' if the conversion of an operand with a decimal data type caused the loss of any non-zero digits in the fractional part of the number.
SQLWARN8	sqlwarn8	SQLWRN(8)	CHAR(1)	Contains 'W' if a statement has been disqualified for blocking. Contains 'Z' if a character that could not be converted was replaced with a substitute character.
SQLWARN9	sqlwarn9	SQLWRN(9)	CHAR(1)	Contains 'W' if blocking was canceled for a cursor because of insufficient storage in the user's virtual machine.

Table 19. Fields of SQLCA (continued)

Assembler, COBOL, or PL/I Name [1]	C Name [1]	Fortran Name [1]	Data Type	Usage
SQLWARNA	sqlwarna	SQLWRN(A)	CHAR(1)	Contains 'W' if blocking was canceled because a blocking factor of at least 2 rows could not be maintained. Contains 'V' if there was a conversion error when converting the value of one of the fields in the SQLCA at the application requester.
SQLSTATE	sqlstate	SQLSTT	CHAR(5)	Contains a return code for the outcome of the most recent execution of an SQL statement[4]. This return code conforms to the SQL92 standard.
1: The field names are those present in an SQLCA obtained from using an INCLUDE statement. **2**: For the specific meanings of DB2 Server for VSE & VM return codes and of variables in error messages, see the *DB2 Server for VM Messages and Codes* or the *DB2 Server for VSE Messages and Codes* manual for your database manager. **3**: In COBOL and C, SQLERRM includes SQLERRML and SQLERRMC. In PL/I, the varying-length string SQLERRM is equivalent to SQLERRML prefix to SQLERRMC. In Assembler, the storage area SQLERRM is equivalent to SQLERRML and SQLERRMC. **4**: For a description of SQLSTATE values, see the *DB2 Server for VM Messages and Codes* or *DB2 Server for VSE Messages and Codes* manual.				

INCLUDE SQLCA Declarations

The description of the SQLCA that is given by `INCLUDE SQLCA` is shown for each of the host languages.

Assembler

```
SQLCA    DS   0F
SQLCAID  DS   CL8
SQLCABC  DS   F
SQLCODE  DS   F
SQLERRM  DS   H,CL70
SQLERRP  DS   CL8
SQLERRD  DS   6F
SQLWARN  DS   0C
SQLWARN0 DS   C
SQLWARN1 DS   C
SQLWARN2 DS   C
SQLWARN3 DS   C
SQLWARN4 DS   C
SQLWARN5 DS   C
SQLWARN6 DS   C
SQLWARN7 DS   C
SQLWARN8 DS   C
SQLWARN9 DS   C
SQLWARNA DS   C
SQLSTATE DS   CL5
```

C

```
#ifndef SQLCODE
struct sqlca
{
       unsigned char  sqlcaid[8];
       long           sqlcabc;
       long           sqlcode;
       short          sqlerrml;
       unsigned char  sqlerrmc[70];
       unsigned char  sqlerrp[8];
       long           sqlerrd[6];
       unsigned char  sqlwarn[11];
       unsigned char  sqlstate[5];
};
#define  SQLCODE   sqlca.sqlcode
#define  SQLWARN0  sqlca.sqlwarn[0]
#define  SQLWARN1  sqlca.sqlwarn[1]
#define  SQLWARN2  sqlca.sqlwarn[2]
#define  SQLWARN3  sqlca.sqlwarn[3]
#define  SQLWARN4  sqlca.sqlwarn[4]
#define  SQLWARN5  sqlca.sqlwarn[5]
#define  SQLWARN6  sqlca.sqlwarn[6]
#define  SQLWARN7  sqlca.sqlwarn[7]
#define  SQLWARN8  sqlca.sqlwarn[8]
#define  SQLWARN9  sqlca.sqlwarn[9]
#define  SQLWARNA  sqlca.sqlwarn[10]
#define  SQLSTATE  sqlca.sqlstate
#endif
struct sqlca sqlca;
```

COBOL

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) COMPUTATIONAL.
   05 SQLCODE      PIC S9(9) COMPUTATIONAL.
   05 SQLERRM.
      49 SQLERRML  PIC S9(4) COMPUTATIONAL.
      49 SQLERRMC  PIC X(70).
   05 SQLERRP      PIC X(8).
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) COMPUTATIONAL.
   05 SQLWARN.
      10 SQLWARN0  PIC X(1).
      10 SQLWARN1  PIC X(1).
      10 SQLWARN2  PIC X(1).
      10 SQLWARN3  PIC X(1).
      10 SQLWARN4  PIC X(1).
      10 SQLWARN5  PIC X(1).
      10 SQLWARN6  PIC X(1).
      10 SQLWARN7  PIC X(1).
      10 SQLWARN8  PIC X(1).
      10 SQLWARN9  PIC X(1).
      10 SQLWARNA  PIC X(1).
   05 SQLSTATE     PIC X(5).
```

Fortran

```
 INTEGER*4        SQLCOD,
*                 SQLERR(6),
*                 SQLTXL*2
 COMMON /SQLCA1/  SQLCOD,SQLERR,SQLTXL

 CHARACTER        SQLERP*8,
*                 SQLWRN(0:10),
*                 SQLTXT*70
*                 SQLSTT*5
 COMMON /SQLCA2/  SQLERP,SQLWRN,SQLTXT,SQLSTT
```

PL/I

```
DCL 1 SQLCA,
      2 SQLCAID      CHAR(8),
      2 SQLCABC      BIN FIXED(31),
      2 SQLCODE      BIN FIXED(31),
      2 SQLERRM      CHAR(70) VAR,
      2 SQLERRP      CHAR(8),
      2 SQLERRD(6)   BIN FIXED(31),
      2 SQLWARN,
        3 SQLWARN0   CHAR(1),
        3 SQLWARN1   CHAR(1),
        3 SQLWARN2   CHAR(1),
        3 SQLWARN3   CHAR(1),
        3 SQLWARN4   CHAR(1),
        3 SQLWARN5   CHAR(1),
        3 SQLWARN6   CHAR(1),
        3 SQLWARN7   CHAR(1),
        3 SQLWARN8   CHAR(1),
        3 SQLWARN9   CHAR(1),
        3 SQLWARNA   CHAR(1),
      2 SQLSTATE     CHAR(5);
```

SQL Descriptor Area (SQLDA)

An SQLDA is a structure or collection of variables that is required for execution of the SQL DESCRIBE statement, and may optionally be used by the OPEN, FETCH, EXECUTE, and PUT statements. An SQLDA communicates with dynamic and extended SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH statement. The *DB2 Server for VSE & VM Application Programming* manual describes the use of an SQLDA.

The meaning of the information in an SQLDA depends on its use. In DESCRIBE and Extended DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In EXECUTE, OPEN, PUT, and Extended EXECUTE, Extended OPEN, and Extended PUT an SQLDA provides information to the database manager about input host variables. In Extended EXECUTE, FETCH and Extended FETCH, an SQLDA provides output information.

SQLDAs are supported in all languages, however predefined declarations are only provided by Assembler, C, and PL/I. In these languages the SQL INCLUDE statement can be used to provide a SQLDA declaration. A similar set of variables is used for this purpose in REXX (see the *DB2 REXX SQL for VM/ESA* manual for details).

Description of Fields

An SQLDA consists of four variables followed by an arbitrary number of occurrences of a sequence of five variables collectively named SQLVAR. In OPEN, FETCH, PUT, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE, they describe columns of a result table.

Table 20. Fields of SQLDA

Assembler or PL/I Name [2]	C Name [2]	Data Type	Usage in DESCRIBE and Extended DESCRIBE (set by the database manager except for SQLN)	Usage in EXECUTE, FETCH, OPEN, PUT, and extended dynamic statements of the same name (set by the user prior to executing the statement)
SQLDAID	sqldaid	CHAR(8)	An 'eye catcher' for storage dumps, containing 'SQLDA '.	For CCSID when the protocol is SQLDS, the sixth position of this field must be set to '+'; for example, 'SQLDA+ '[1] (See SQLNAME in Table 21) Not used otherwise.
SQLDABC	sqldabc	INTEGER	Length of the SQLDA, equal to SQLN*44+16.	Number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to contain SQLN occurrences. SQLDABC must be set to a value greater than or equal to 16+SQLN*44.
SQLN	sqln	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero before the DESCRIBE statement is processed. 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	sqld	SMALLINT	For a SELECT statement, the number of columns described by occurrences of SQLVAR (or, if USING BOTH was specified on DESCRIBE, twice the number of columns). For a non-SELECT statement, 0.	The number of host variables described by occurrences of SQLVAR to be used in the SQLDA when executing this statement. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.
1: When SQLDS or DRDA protocols are being used, the database manager calculates the CCSID value of the user data area before the first FETCH (or PUT) cursor operation. This CCSID value is ONLY recalculated on subsequent FETCHes (or PUTs) if position 6 of the SQLDAID field has been set to '+'. Using this mechanism, it is possible to dynamically change the CCSID value for an open cursor. **2:** The field names are those present in an SQLCA obtained from an INCLUDE statement.				

Fields in an Occurrence of SQLVAR

Table 21. Fields in SQLVAR

Assembler or PL/I Name	C Name	Data Type	Usage in DESCRIBE and Extended DESCRIBE (set by the database manager except for SQLN)	Usage in EXECUTE, FETCH, OPEN, PUT, and extended dynamic statements of the same name (set by the user prior to executing the statement)
SQLTYPE	sqltype	SMALLINT	Indicates the data type of the column and whether it can contain nulls. For a description of the type codes, see Table 22 on page 362.	Indicates the data type of the host variable and whether an indicator variable is provided. For a description of the type codes, see Table 22 on page 362.

Table 21. Fields in SQLVAR (continued)

Assembler or PL/I Name	C Name	Data Type	Usage in DESCRIBE and Extended DESCRIBE (set by the database manager except for SQLN)	Usage in EXECUTE, FETCH, OPEN, PUT, and extended dynamic statements of the same name (set by the user prior to executing the statement)
SQLLEN	sqllen	SMALLINT	The length attribute of the column. For datetime columns, the length of the string representation of the values. See Table 22 on page 362.	The length attribute of the host variable. See Table 22 on page 362.
SQLDATA	sqldata	pointer	For string columns, SQLDATA contains the CCSID of the column. For character string columns, SQLDATA can alternatively contain the value X'FFFF' indicating bit data. For datetime columns, the SQLDATA contains the CCSID of the string representation of the values. See Table 23 on page 363 for more information.	A pointer to the storage area that either holds the parameter value (if SQLDA is used for input), or is to hold a select list result (if the SQLDA is used for output). For varying-length character strings, the actual data should be preceded by a halfword field that specifies the length of the character string. (The value should not include the length of the halfword.) The data must be aligned on a halfword boundary.
SQLIND	sqlind	pointer	For character and datetime data, byte 1 of SQLIND is set as follows: • X'FF' for a bit value • X'01' for a SBCS value • X'02' for a mixed value This information is not available when using the DRDA protocol.	Contains the address of the indicator variable where applicable. The indicator variable must be declared as a 15-bit integer. For an Input SQLDA, the indicator should be set to 0 to indicate that the parameter value is not null and to a negative value to indicate that the parameter value is null. For an Output SQLDA, the database manager fills in the indicator using the following rules: **0** Denotes that the parameter is not null, and is in the associated storage area. **<0** Denotes that the parameter value is null. **>0** Denotes that a returned value was truncated because the storage area provided was not large enough. If the truncated item was a DBCS or character string, the indicator variable contains the length in characters before truncation. (Applies only for the FETCH statement.) When a time value is truncated at its seconds part on output, the seconds are placed in the SQLIND.

Table 21. Fields in SQLVAR (continued)

Assembler or PL/I Name	C Name	Data Type	Usage in DESCRIBE and Extended DESCRIBE (set by the database manager except for SQLN)	Usage in EXECUTE, FETCH, OPEN, PUT, and extended dynamic statements of the same name (set by the user prior to executing the statement)
SQLNAME	sqlname	VARCHAR (30)	Contains the name or label associated with the column used in the select list of the DESCRIBE statement. For more information, see "SQLNAME" on page 248.	For character, datetime and graphic data, SQLNAME may be used to override [1] the default CCSID. An override is indicated by the following: • the length of the SQLNAME field is 8 • the first 2 bytes of SQLNAME have a value of X'0000' • for SQLDS protocol, the sixth position of the SQLAID field must be '+'; for example, 'SQLDA+ ' (see SQLDAID in Table 20 on page 360). The override itself is present in bytes 3 and 4 of the SQLNAME. See Table 23 on page 363 for the relation between the character subtypes, the graphic data type, and the CCSID. Note that bytes 5 to 8 of the SQLNAME field are reserved by IBM for future use in override situations for character and graphic data. For all other data, SQLNAME is not used.

1: It is important to note that this use of the SQLNAME field is only for overrides. Applications that use the defaults and have properly initialized SQLDAs need not be concerned.

Note: Note that in a remote unit of work application a DESCRIBE of a SELECT statement will return the application server's CCSIDs and that these will not necessarily be the same as any host variables that may be in the select list (these will have CCSIDs from the application requester).

SQLTYPE and SQLLEN

The following table shows the values that may appear in the SQLTYPE and SQLLEN fields of the SQLDA. In DESCRIBE, an even value of SQLTYPE means the column does not allow nulls, and an odd value means the column does allow nulls. In EXECUTE, FETCH, OPEN, and PUT, an even value of SQLTYPE means no indicator variable is provided, and an odd value means that SQLIND contains the address of an indicator variable.

Table 22. SQLTYPE and SQLLEN Values for DESCRIBE, EXECUTE, FETCH, OPEN, and PUT

	For DESCRIBE		For EXECUTE, FETCH, OPEN, and PUT	
SQLTYPE	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
384/385	date	10 or length of LOCAL date format	fixed-length character string representation of a date	length attribute of the host variable
388/389	time [1]	8 or length of LOCAL time format	fixed-length character string representation of a time	length attribute of the host variable

Table 22. SQLTYPE and SQLLEN Values for DESCRIBE, EXECUTE, FETCH, OPEN, and PUT (continued)

	For DESCRIBE		For EXECUTE, FETCH, OPEN, and PUT	
SQLTYPE	COLUMN DATA TYPE	SQLLEN	HOST VARIABLE DATA TYPE	SQLLEN
392/393	timestamp [1]	26	fixed-length character string representation of a timestamp	length attribute of the host variable
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 varying-length graphic string	length attribute of the host variable
480/481	floating point	4 for single precision, 8 for double precision	floating point	4 for single precision, 8 for double precision
484/485	packed decimal	precision in byte 1; scale in byte 2	packed decimal	precision in byte 1; scale in byte 2
488/489	zoned decimal [2]	precision in byte 1; scale in byte 2	zoned decimal [2]	precision in byte 1; scale in byte 2
496/497	large integer	4	large integer	4
500/501	small integer	2	small integer	2
504/505	N/A	N/A	DISPLAY SIGN LEADING SEPARATE	precision in byte 1; scale in byte 2

1: Since host variables do not have datetime data types, character string variables must be used to retrieve datetime values. Thus, when the SQLDA describes host variables, these type-codes denote fixed-length character string variables.

2: Zoned decimal is not supported for local operations.

CCSID Usage

The following table describes the SQLDATA field for the DESCRIBE statement and the SQLNAME field for host variables.

Table 23. CCSID Values for SQLDATA and SQLNAME

Data Type	Subtype	Bytes 1 & 2	Bytes 3 & 4
Character	SBCS data	X'0000'	The CCSID value
Character	mixed data	X'0000'	The CCSID value
Datetime	SBCS data	X'0000'	The CCSID value
Datetime	mixed data	X'0000'	The CCSID value

Table 23. CCSID Values for SQLDATA and SQLNAME (continued)

Data Type	Subtype	Bytes 1 & 2	Bytes 3 & 4
Character	bit data	X'0000'	X'FFFF'
Graphic	N/A	X'0000'	The CCSID value
Any other data type	N/A	N/A	N/A

INCLUDE SQLDA Declarations

The description of the SQLDA that is given by `INCLUDE SQLDA` is shown for assembler, PL/I and C. Though you can use an SQLDA in VS COBOL II, and Fortran, the `INCLUDE` statement does not provide the code; you must provide it, as shown in the *DB2 Server for VSE & VM Application Programming* manual.

Assembler

```
SQLDA    DSECT
SQLDAID  DS     CL8
SQLDABC  DS     F
SQLN     DS     H
SQLD     DS     H
SQLVAR   DS     0F
SQLVARN  DSECT
SQLTYPE  DS     H
SQLLEN   DS     0H
SQLPRCSN DS     CL1
SQLSCALE DS     CL1
SQLDATA  DS     A
SQLIND   DS     A
SQLNAME  DS     H,CL30
&SYSECT  CSECT
```

C

```
#ifndef SQLDASIZE
struct sqlda {
   unsigned char sqldaid[8];
   long sqldabc;
   short sqln;
   short sqld;
      struct sqlvar {
      short sqltype;
      short sqllen;
      unsigned char *sqldata;
      short *sqlind;
      struct sqlname {
         short length;
         char data[30];
      } sqlname;
   } sqlvar[1];
};

#define SQLDASIZE(n)
      (sizeof(struct sqlda)+((n)-1)*sizeof(struct sqlvar))
#endif
```

Note: SQLDA character array variables sqldaid and sqlname.data are not NUL-terminated. They cannot be directly used by C string manipulation functions.

The SQLDA must not be declared within the SQL declare section.

Using the defined preprocessor function SQLDASIZE, your program can dynamically allocate an SQLDA of adequate size for use with each EXECUTE statement. For example, the code fragment below allocates an SQLDA adequate for five fields and uses it in an EXECUTE statement S3:

```
struct sqlda *sqlptr;

   sqlptr = (struct sqlda *)malloc(SQLDASIZE(5));
   sqlptr->SQLN=5;
   /* Add code to set the rest of values and pointers in the SQLDA */
   EXEC SQL EXECUTE S3 USING DESCRIPTOR *sqlptr;
```

Note: The variable used to point to the SQLDA is not defined in a SQL declare section. Its context within an SQL statement (following INTO or USING DESCRIPTOR) is enough to identify it.

You can use a similar technique to allocate an SQLDA for use with a DESCRIBE statement. The following program fragment illustrates the use of SQLDA with DESCRIBE for three fields and a 'prepared' statement S1:

```
struct sqlda *sqlptr;

   EXEC SQL DECLARE C1 CURSOR FOR S1;
   sqlptr = (struct sqlda *)malloc(SQLDASIZE(3));
   sqlptr->sqln=5;
   EXEC SQL DESCRIBE S1 INTO *sqlptr;
   if (sqlptr->sqld > sqlptr->sqln)
      --get a bigger one
   Set sqldata and sqlind
   EXEC SQL OPEN C1;
   EXEC SQL FETCH C1 USING DESCRIPTOR *sqlptr;
```

There is no standard C to support packed decimal data. If data in packed decimal format is required, the SQLDA must be filled in with an SQLTYPE of 484 or 485, with the appropriate values for precision and scale in SQLLEN. The C program would then deal with the data in its packed format.

PL/I

```
DCL 1 SQLDA BASED(SQLDAPTR),
      2 SQLDAID     CHAR(8),
      2 SQLDABC     BIN FIXED(31),
      2 SQLN        BIN FIXED(15),
      2 SQLD        BIN FIXED(15),
      2 SQLVAR      (SQLSIZE REFER(SQLN)),
        3 SQLTYPE   BIN FIXED(15),
        3 SQLLEN    BIN FIXED(15),
        3 SQLDATA   PTR,
        3 SQLIND    PTR,
        3 SQLNAME   CHAR(30) VAR;
DCL SQLSIZE BIN FIXED(15);
DCL SQLDAPTR PTR;
```

The SQLDA must not be declared within the SQL declare section.

In addition to the structure above, you should also declare an additional mapping for the same area. The SQLPRCSN and SQLSCALE fields of the second mapping are used when decimal data is used. An example of this mapping follows.

```
DCL 1 SQLDA BASED(SQLDAPTR),
      2 SQLDAIDX    CHAR(8),
      2 SQLDABCX    BIN FIXED(31),
      2 SQLNX       BIN FIXED(15),
      2 SQLDX       BIN FIXED(15),
      2 SQLVARX(SQLSIZE REFER(SQLNX)),
         3 SQLTYPEX  BIN FIXED(15),
         3 SQLPRCSN  format 1 or format 2
         3 SQLSCALE  format 1 or format 2
         3 SQLDATAX  PTR,
         3 SQLINDX   PTR,
         3 SQLNAMEX  CHAR(30) VAR:
```

You can declare the SQLPRCSR and SQLSCALE fields in one of two formats:

Format 1

```
         3 SQLPRCSN  BIT(8),
         3 SQLSCALE  BIT(8),
```

The fields must be set by 8-bit strings. For example, for a precision of 5 and a scale of 2, the following assignments are required:

```
SQLDAPTR->SQLPRCSN = '00000101'B,
SQLDAPTR->SQLSCALE = '00000010'B,
```

Format 2

```
         3 SQLPRCSN  CHAR(1),
         3 SQLSCALE  CHAR(1),
```

This format requires the declaration of additional variables. These are a CHAR(2) variable and a BASED FIXED BIN (15) variable for both precision and scale. For example:

```
DCL PRCSNC CHAR(2)
DCL PRCSNN FIXED BIN(15) BASED (ADDR(PRCSNC));
DCL SCALEC CHAR(2);
DCL SCALEN FIXED BIN(15) BASED (ADDR(SCALEC));
```

For a precision of 5 and a scale of 2, the following assignments are required:

```
PRCSNN = 5;
SCALEN = 2;
```

The SQLDAX fields for a precision of 5 and a scale of 2 would be:

```
SQLDAPTR->SQLPRCSN = SUBSTR(PRCSNC,2,1);
SQLDAPTR->SQLSCALE = SUBSTR(SCALEC,2,1);
```

This format, though more complex than Format 1, allows PL/I manipulation of the precision and scale fields. For example, the value of the SQLPRCSN field can be determined by simply reversing the substring operation above. That is:

```
SUBSTR(PRCSNC,2,1) = SQLDAPTR->SQLPRCSN;
```

Such an operation is not possible using Format 1.

Because the PL/I SQLDA is declared as a based structure, your program can dynamically allocate an SQLDA of adequate size with each EXECUTE statement. For example, the code fragment below allocates an SQLDA adequate for five fields and uses it to EXECUTE statement S3:

```
SQLSIZE=5;
ALLOCATE SQLDA SET(SQLDAPTR);
/*Add code to set values and pointers in the SQLDA*/
EXEC SQL EXECUTE S3 USING DESCRIPTOR SQLDA;
```

The statement SQLSIZE=5 determines the size of the SQLDA to be allocated by means of the PL/I REFER feature. The ALLOCATE statement allocates an SQLDA of the size desired, and sets SQLDAPTR to point to it. (Before an EXECUTE statement is issued using this SQLDA, your program must fill its contents.)

You can use a similar technique to allocate an SQLDA for use with a DESCRIBE statement. The following program fragment illustrates the use of SQLDA with DESCRIBE for three fields and a 'prepared' statement S1:

```
EXEC SQL DECLARE C1 CURSOR FOR S1;
SQLSIZE=3;
ALLOCATE SQLDA SET(SQLDAPTR);
EXEC SQL DESCRIBE S1 INTO SQLDA;
IF SQLID>SQLN THEN
    - get a bigger one;
Set SQLDATA and SQLIND;
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;
```

Appendix C. DB2 Server for VSE & VM Catalog

This appendix is intended to help you to use the catalog for your database manager. It contains Product-Sensitive Programming Interface and Associated Guidance Information.

The DB2 Server for VSE & VM database manager automatically maintains information about the database in a set of tables called the *catalog*. The catalog tables are created by the database manager during database generation. They describe tables, columns, indexes, keys, packages, authorities, and other objects in the database. Data in the catalog tables is available to authorized users through normal SQL query facilities; however, the catalog is primarily intended for use by the database manager.

During database generation, the catalog is defined as normal tables with PUBLIC read authorization. After database generation, a user with DBA authority can revoke the select privilege from PUBLIC. Usually all users are allowed to access the catalog, so you can use SQL statements to retrieve information in the catalog. For example, this SQL statement finds what column names in table SALARY begin with the letter 'D':

```
SELECT CNAME FROM SYSTEM.SYSCOLUMNS
   WHERE TNAME = 'SALARY'
   AND CNAME LIKE 'D%'
```

SYSTEM is the owner of all catalog tables except SYSLANGUAGE (which is owned by SQLDBA). You must qualify all references to catalog tables with the owner name, unless you have a synonym defined.

After database generation, the only information in the tables not available to everyone is password information. You must have DBA authority to access the catalog table that contains passwords (SYSUSERAUTH). A view, called SYSUSERLIST, is defined on SYSUSERAUTH when the catalog tables are created. The owner of the view is SQLDBA, so you must refer to the view as SQLDBA.SYSUSERLIST. This view is accessible to all users and contains all the columns of SYSUSERAUTH except the passwords. If you do not have DBA authority, you must query the view (SYSUSERLIST) instead of the underlying table (SYSUSERAUTH).

Some of the information in the catalog is of little interest to most users. Statistics maintained in the catalog, for example, are used by the database manager to determine optimal access paths. These statistics may be quite meaningless to you. If you wish, you can define views on the catalog tables containing only columns that are meaningful to you.

Some of the information in the catalog is maintained in a form for internal use by the database manager and is provided as additional guidance on database administration tasks. Two special data types are used: DBAINT and DBAHW. These appear externally like INTEGER and SMALLINT data. However, DBAINT and DBAHW do not sort as expected when they contain negative values. Consequently, queries that use ORDER BY, GROUP BY, or predicates that involve > or < operations on these values may not work as expected.

The database manager updates its catalog during normal operation in response to SQL data definition and control statements. It also updates its catalog when programs are preprocessed.

You can create and maintain your own installation-dependent catalog tables using SQL statements.

Note: Data in the catalog tables is available to authorized users through normal SQL query facilities; however, the catalog is primarily intended for use by the database manager, and is therefore subject to change.

"Roadmap" to Catalog

Item	**Catalog Table**	
authorization	SYSUSERAUTH SYSUSERLIST	406
character conversion	SYSSTRINGS	400
character set	SYSCHARSETS	378
coded character set identifiers	SYSCCSIDS SYSSTRINGS	378 400
column	SYSCOLUMNS SYSKEYCOLS	381 390
column update privilege	SYSCOLAUTH	379
column with field procedure	SYSFIELDS	386
constraint	SYSKEYS	391
dbspace	SYSDBSPACES SYSUSAGE SYSDROP	384 405 385
dbspace waiting to be dropped	SYSDROP	385
default	SYSOPTIONS	393
dropped dbspace	SYSDROP	385
dropped table	SYSDROP	385
field procedures	SYSFPARMS SYSFIELDS	387 386
foreign key	SYSKEYS	391
index	SYSINDEXES SYSUSAGE	388 405
index column statistics	SYSCOLSTATS SYSCOLUMNS SYSINDEXES	380 381 388
key	SYSKEYS	391
key column	SYSKEYCOLS	390
language for character set	SYSLANGUAGE	392
option	SYSOPTIONS	393
package	SYSACCESS SYSUSAGE	373 405
package run privilege	SYSPROGAUTH	396
password	SYSUSERAUTH	406

Item	Catalog Table	
privilege	SYSCOLAUTH SYSPROGAUTH SYSTABAUTH	379 396 403
primary key	SYSKEYS	391
statistics	SYSCATALOG SYSCOLSTATS SYSCOLUMNS SYSDBSPACES SYSINDEXES	375 380 381 384 388
synonym	SYSSYNONYMS	402
stored procedures	SYSPARMS SYSROUTINES SYSPSERVERS	395 398 397
table	SYSCATALOG SYSCOLUMNS SYSUSAGE	375 381 405
table privilege	SYSTABAUTH	403
table waiting to be dropped	SYSDROP	385
unique constraint	SYSKEYS	391
view	SYSVIEWS SYSCATALOG SYSCOLUMNS SYSACCESS SYSUSAGE	406 375 381 373 405
view privilege	SYSTABAUTH	403

Updateable Columns

Only someone with DBA authority may enter UPDATE, INSERT and DELETE statements against catalog tables. Furthermore, only the following columns may be altered. It is not possible to add columns to the catalog.

Catalog Column	Update	Insert	Delete
SYSACCESS			
VALID [1]	X		
SYSCATALOG			
CLUSTERTYPE	X		
CLUSTERROW	X		
AVGROWLEN	X		
ROWCOUNT	X		
NPAGES	X		
PCTPAGES	X		
SYSCCSIDS			
all columns	X	X	X
SYSCHARSETS			
all columns	X	X	X

Updateable Columns

Catalog Column	Update	Insert	Delete
SYSCOLSTATS			
VAL10	X		
VAL50	X		
VAL90	X		
FREQ1VAL	X		
FREQ1PCT	X		
FREQ2VAL	X		
FREQ2PCT	X		
SYSCOLUMNS			
COLCOUNT	X		
HIGH2KEY	X		
LOW2KEY	X		
AVGCOLLEN	X		
COLINFO	X		
SUBTYPE [2]	X		
SYSDBSPACES			
NACTIVE	X		
NPAGES [3]	X		
SYSDROP			
all columns			X
SYSINDEXES			
CLUSTER	X		
KEYLEN	X		
FIRSTKEYCOUNT	X		
FULLKEYCOUNT	X		
NLEAF	X		
NLEVELS	X		
CLUSTERRATIO	X		
SYSLANGUAGE			
all columns	X	X	X
SYSOPTIONS			
all columns	X	X	X
SYSSTRINGS			
all columns	X	X	X

Notes:

1. It is advisable to enter a REBIND command, rather than updating VALID to force dynamic re-preprocessing,

2. Updating this field is only effective if the corresponding CCSID field value for the row is null. Upon successful update of the SUBTYPE value, all packages which reference the column whose SUBTYPE has been updated must be re-preprocessed.
3. **CAUTION:**
Changing NPAGES makes the dbspace appear to be a different size without actually changing it. NPAGES should not be changed in a production environment, to do so may cause errors to occur. It is intended for testing purposes only.

SYSACCESS

Packages are stored in tables. The database manager uses SYSACCESS to record information about the tables in which packages are stored. For package tables that are *in use*, SYSACCESS records information about:
- Packages created by the preprocessors or by a CREATE PACKAGE statement.
- View definitions (views are stored as packages).

When a package table is not in use, SYSACCESS indicates whether the table is available or unavailable.

The columns in SYSACCESS are:

Column Name	Data Type	Description and Comments
TNAME	VARCHAR(18) NOT NULL	When the package table is in use, TNAME is either the name of the package or the name of a view. A view definition is stored as a package; the name of the package is the name of the view. The TABTYPE field indicates whether this row describes a real package or a view. When the package table is unused, TNAME is either '!0**x** AVAILABLE' or '¢0**x** UNAVAILABLE' to indicate whether the table is available or unavailable. A package table is available when it is unused and the DBSPACE is not full. A package table is unavailable when it is unused and the DBSPACE is full. A package table may also be marked as unavailable when a package is dropped from a DBSPACE that was previously marked full. Such package tables are marked as available the next time the database manager pre-allocates packages. The **x** is a number from one to five that is used internally.
CREATOR	CHAR(8) NOT NULL	The owner of the package or view who either preprocessed the program associated with this package, explicitly created the package (by CREATE PACKAGE), or created this view. If the package table is unused, CREATOR is a non-readable unique value that is based on the system clock. (The database manager generates this value for unused package tables because TNAME and CREATOR serve as a key for an index on SYSTEM.SYSACCESS.)
DBSPACENO	DBAHW NOT NULL	The number of the DBSPACE that contains this package. (When DBSPACEs are defined by database generation or by ADD DBSPACE processing, the database manager assigns each DBSPACE a number for internal use.)
TABID	DBAHW NOT NULL	Packages are stored as tables. TABID contains the internal identifier of that table. (In the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual, this identifier is known as the DBSS RID.)

Column Name	Data Type	Description and Comments
LINKID	DBAHW NOT NULL	A package may occupy more than one row of the table in which it is stored. The database manager connects these rows in the correct order by a mechanism called a *unary link*. LINKID is the identifier of that unary link.
FIRSTROW	DBAINT NOT NULL	The internal identifier for the first row of the unary link. (In the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual, row identifiers are known as DBSS TIDs.) FIRSTROW is 0 for unused package tables.
TIMESTAMP	CHAR(17) NOT NULL	The date and time when this package was created. The field has the format MM/DD/YY HH:MM:SS. It is updated when the database manager automatically preprocesses the package. (The database manager attempts to preprocess a package when some dependency is lost; for example, when a package tries to use an index that was dropped.) TIMESTAMP is blank if the package table is unused.
VALID	CHAR(1) NOT NULL	The possible values are: **Y** if the package is valid. **N** if the package is not valid because: • a view, index, table, or DBSPACE has been dropped • the application server CHARNAME has been changed and the package or view definition has a dependency on a changed system table. In this case, please refer to the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual for a list of all affected tables. **blank** if the package table is unused.
TABTYPE	CHAR(1) NOT NULL	The possible values are: **X** if this row describes a package. **V** if this row describes a view definition. **blank** if the package table is unused.
CONSTKN	CHAR(8) FOR BIT DATA	The consistency token for this package. The field is one of: **eight blanks** If this entry is for a view, the view was created or repreprocessed on an SQL/DS database Version 3 Release 1 or later. If not for a view, CTOKEN(NO) was specified or allowed to default in the preprocessor options. **timestamp** CTOKEN(YES) was specified in the preprocessor options. **null** The package or view was migrated from an SQL/DS database prior to Version 3 Release 1.

Column Name	Data Type	Description and Comments
PLABEL	VARCHAR(30)	The label for this package. The field is one of: **thirty blanks** If this entry is for a view, the view was created or repreprocessed on an SQL/DS database Version 3 Release 1 or later. If for a package, the LABEL option was not specified in either the preprocessor options or in a CREATE PACKAGE statement. **label-text** LABEL(label-text) was specified in the preprocessor options or for CREATE PACKAGE. **null** The package or view was migrated from an SQL/DS database prior to Version 3 Release 1.

SYSCATALOG

The SYSCATALOG table contains a row for each table or view in the database, including itself and other catalog tables.

The columns in SYSCATALOG are:

Column Name	Data Type	Description and Comments
TNAME	VARCHAR(18) NOT NULL	The name of the table or view being described.
CREATOR	CHAR(8) NOT NULL	The owner of the table or view. (The CREATOR of the catalog is SYSTEM.)
TABLETYPE	CHAR(1) NOT NULL	The possible values are: **V** if the object is a view. **R** if the object is a real table.
NCOLS	SMALLINT NOT NULL	The number of columns in the table or view.
REMARKS	VARCHAR(254) NOT NULL	The information from a COMMENT statement entered for the table or view. The remarks are deleted from SYSCATALOG when the table or view is dropped. If the DBCS option is enabled, users can store mixed data (EBCDIC and DBCS) in the REMARKS column.
DBSPACENO	DBAHW NOT NULL	The possible values are: **0** if the object is a view. **number** if the object is a real table. This number is the internal number of the DBSPACE in which the table is stored. This is the DBSPACE number to which some of the SHOW operator commands refer (such as SHOW DBSPACE).
DBSPACENAME	VARCHAR(18) NOT NULL	The name of the DBSPACE containing the table.

Column Name	Data Type	Description and Comments
TABID	DBAHW NOT NULL	The possible values are: **0** if the object is a view. **number** if the object is a real table. This number is the internal identifier of the table. (In the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual, the internal identifier is referred to as the DBSS RID.)
CLUSTERTYPE	CHAR(1) NOT NULL	The possible values are: **I** if the rows are clustered by an index. **D** if the rows are clustered by default rules. This is its initial value. Internally, the physical placement of rows is determined by an index or (if no index is available) by default rules. The default rules place each new row near the previously inserted row. CLUSTERTYPE is updated by CREATE and DROP INDEX statement on this table. In addition to the above values, the possible values for catalog tables are: **L** if the rows are clustered by link rules. For certain catalog tables, a direct addressing link is set up to enable faster access to a specific row. A CLUSTERTYPE value of 'L' indicates that rows are clustered in link order. **N** if the rows are clustered by internal RDS rules.
CLUSTERROW	DBAINT NOT NULL	The possible values are: **0** if the object is a view. This is also an initial value. **number** if the object is a real table. This number is the highest internal row identifier (DBSS TID) for any row in the table. The database manager uses this value when it is clustering rows by default rules. See Note 1 for update rules on this column.
AVGROWLEN	DBAHW NOT NULL	The average length of the rows in this table, rounded to the nearest integer. This field is set to -1 when the table is created. See Note 1 for update rules on this column.
ROWCOUNT	DBAINT NOT NULL	The total number of rows in this table. This is updated to the following values by CREATE TABLE, UPDATE STATISTICS, and DATALOAD/RELOAD as indicated: **-2** When a DATALOAD with COMMITCOUNT option reaches the commit threshold and commits the loaded rows if statistics are collected while data is being loaded. **-1** When the table is initially created with CREATE TABLE. **>=0** When an UPDATE STATISTICS is performed, or when a CREATE/REORGANIZE INDEX is performed, or when data is loaded using DATALOAD/RELOAD (and update statistics is not set off). The integer is equal to the total number of rows in this table and it is updated only if one of the previously listed operations is performed.

Column Name	Data Type	Description and Comments
NPAGES	DBAINT NOT NULL	The number of pages on which rows of this table appear. This number is approximate because it does not contain those pages that contain only long fields. Thus, the sum of the NPAGES for all tables in a DBSPACE might be less than NACTIVE in SYSDBSPACES. (NACTIVE is the total number of active data pages in a DBSPACE.) This field is set to -1 when the table is created, and is updated to a non-negative integer according to the same rules as the ROWCOUNT column.
PCTPAGES	DBAHW NOT NULL	The approximate percentage of the total active pages in the DBSPACE that have rows from this table on them. The initial value in this field is -1. Loading, updating or dropping of any table can affect the actual percentage of used pages for all tables in a given DBSPACE and PCTPAGES may not reflect this. The database manager takes this into account and dynamically calculates (but does not update) PCTPAGES whenever it is used. The calculated value is NPAGES/ SYSDBSPACES.NACTIVE. See Note 1 for update rules on this column.
NOVERFLOW	DBAINT NOT NULL	The number of rows in this table that have overflowed from their original page in storage to another page. If this number is large, it may be time to reorganize the table by dumping it out of the database and reloading it. See Note 1 for update rules on this column.
LFDTABID	DBAHW NOT NULL	The internal table identification (referred to in the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual as the DBSS RID) of a secondary table the database manager uses to store any long fields that exist in this table. The secondary table is transparent to users. This field is zero if the described table has no long fields.
LFDLINK	DBAHW NOT NULL	The rows in the secondary table that contains long field data are linked together by an internal mechanism called a *unary link*. LFDLINK is the identifier of that unary link. (In the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual, the internal identifier is referred to as a DBSS LID.) If the described table contains no long fields, LFDLINK is zero.
LFDDBSPACE	DBAHW NOT NULL	The number of the DBSPACE that contains the long field data table. LFDDBSPACE is zero if there are no long fields in the described table.
TLABEL	VARCHAR(30)	A table label supplied by a user using a LABEL statement. The table labels are deleted from SYSCATALOG when the table or view is dropped. If the DBCS option is enabled, users can store mixed data (EBCDIC and DBCS) in the TLABEL column.
PARENTS	SMALLINT	The number of parent relationships in which the table is a dependent. Can be NULL if migrated from SQL/DS Version 2 Release 1 or earlier.
DEPENDENTS	SMALLINT	The number of dependent relationships in which the table is a parent. Can be NULL if migrated from SQL/DS Version 2 Release 1 or earlier.
INACTIVE	SMALLINT	The number of inactive keys for the table. This includes inactive primary keys, inactive foreign keys, and foreign keys that reference an inactive primary key in another table. Can be NULL if migrated from SQL/DS Version 2 Release 1 or earlier.

Column Name	Data Type	Description and Comments
DATACAPTURE	CHAR(1)	Records the value of the DATA CAPTURE specification for a table. This value can be NULL if the database was migrated from Version 3 Release 5 or earlier, 'blank' if DATA CAPTURE NONE was specified for the table, or Y if DATA CAPTURE CHANGES was specified.

Note 1: The value is always updated by an UPDATE STATISTICS statement on this table or by the DATALOAD/RELOAD DBS Utility commands if the collecting of statistics has not been turned off by SET UPDATE STATISTICS OFF.

SYSCCSIDS

The SYSCCSIDS table contains a row for every CCSID supported by the installation.

Column Name	Data Type	Description and Comments
CCSID	INTEGER NOT NULL	Identifies the CCSIDs supported by the installation. The values in this field identify valid CCSIDs when columns are created by the CREATE TABLE or ALTER TABLE statement. This column is defined with a UNIQUE constraint.
SUBTYPE	CHAR(1) NOT NULL	Identifies the subtype of the CCSID. The possible values are: **B** for bit data. **M** for mixed data. **S** for SBCS data. **blank** for anything other than non-character.
SBCSID	INTEGER NOT NULL	Identifies the SBCS portion of a mixed CCSID.
DBCSID	INTEGER NOT NULL	Identifies the DBCS portion of a mixed CCSID.
CHARNAME	CHAR(18) NOT NULL	The name of the character set specified by the SQLINIT EXEC (for example, FRENCH, INTERNATIONAL, 937).

More information on CCSIDs can be found in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

SYSCHARSETS

The rows in SYSCHARSETS contain information about various EBCDIC character sets. The database manager reads a row from this table during initialization based on the name specified by the CHARNAME parameter of the SQLSTART command.

The database manager uses the character sets to identify valid characters, to fold lowercase characters to uppercase properly, and for the TRANSLATE function.

IBM supplies sample DBS Utility control files that you can use for loading character set information into SYSCHARSETS. Or, you can define your own character sets and have them loaded by someone with DBA authority. For more information on how to define your own character set, see the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

SYSCHARSETS is only for SBCS character sets.

Column Name	Data Type	Description and Comments
NAME	VARCHAR(18) NOT NULL	The name used to identify the character set. NAME is usually the national language name of the character set (for example, FRENCH) and corresponds exactly to the CHARNAME in the SYSCCSIDS catalog table. A CCSID is associated with each name.
CHARCLASS	CHAR(192) NOT NULL FOR BIT DATA	This contains the character classifications for this character set.
CHARTRANS	CHAR(192) NOT NULL FOR BIT DATA	This contains the character translation values for this character set. The character translation values are used for lowercase to uppercase folding.

SYSCOLAUTH

SYSCOLAUTH records grants of the UPDATE privilege on tables and views when the privilege is granted on a column-by-column basis. Each entry in SYSCOLAUTH has a corresponding entry in SYSTABAUTH with a matching timestamp. (SYSTABAUTH records privileges granted on entire tables, but not on individual columns.) A SYSCOLAUTH entry identifies a particular column on which an UPDATE privilege has been granted. For example, if the UPDATE privilege is granted on several columns in one GRANT statement, the grant is represented as one entry in SYSTABAUTH, and several entries in SYSCOLAUTH, all having matching timestamps.

Some of the entries in SYSCOLAUTH represent privileges that are exercised by preprocessed programs. These entries appear as though the creator of the program (the user who preprocessed the program) granted the privilege to the program itself. The columns in SYSCOLAUTH are:

Column Name	Data Type	Description and Comments
GRANTOR	CHAR(8) NOT NULL	The user ID of the person who granted the UPDATE privilege on this column.
GRANTEE	CHAR(8) NOT NULL	The user ID of the person who holds the UPDATE privilege. If the *userid* is PUBLIC, the privilege is held by all users.
CREATOR	CHAR(8) NOT NULL	The owner of the table that contains the column.
TNAME	VARCHAR(18) NOT NULL	The name of the table that contains the column. (CREATOR.TNAME uniquely identifies the table that contains the column.)
TIMESTAMP	CHAR(12) NOT NULL	The value of the System/390 time of day clock when the grant was made. This value is used internally when privileges are revoked, and is stored as a string of numbers and letters.
COLNAME	VARCHAR(18) NOT NULL	The name of the column on which the UPDATE privilege has been granted.

Note: The authorization for update by column appears in a separate table, one column per row, only because it is possible to grant the UPDATE privilege on specific columns of the table. If the user has the UPDATE privilege on all columns of a table, that information does not appear in SYSCOLAUTH; rather, UPDATECOLS in SYSTABAUTH is set to ‘ ’. Otherwise, UPDATECOLS contains ‘*’ to indicate that more information is in the SYSCOLAUTH table.)

SYSCOLSTATS

The SYSCOLSTATS table keeps the column statistics listed below for a column which is the first column of an index. SYSCOLSTATS is updated whenever an index is created, reorganized or dropped, or UPDATE STATISTICS is run. The statistics in SYSCOLSTATS are used internally by the database manager.

Because SYSCOLSTATS records the first and second-most frequent values in the first column used by every index on every table in the database, you should consider revoking public access to SYSCOLSTATS if any of these values could be sensitive data.

The columns in SYSCOLSTATS are:

Column Name	Data Type	Description and Comments
CNAME	VARCHAR(18) NOT NULL	The name of the column described.
TNAME	VARCHAR(18) NOT NULL	The name of the table in which the column (CNAME) is located.
CREATOR	CHAR(8) NOT NULL	The owner who created the table identified by TNAME. (Thus, CNAME is the name of a column in the table identified by CREATOR.TNAME.)
VAL10	VARCHAR(12) NOT NULL FOR BIT DATA	The value of column CNAME at the tenth percentile. If the table TNAME has N rows and all N values of CNAME are arranged in ascending order, then VAL10 is at position 0.1 * N in this sequence.
VAL50	VARCHAR(12) NOT NULL FOR BIT DATA	The value of column CNAME at the 50th percentile. If the table TNAME has N rows and all N values of CNAME are arranged in ascending order, then VAL50 is at position 0.5 * N in this sequence.
VAL90	VARCHAR(12) NOT NULL FOR BIT DATA	The value of column CNAME at the 90th percentile. If the table TNAME has N rows and all N values of CNAME are arranged in ascending order, then VAL90 is at position 0.9 * N in this sequence.
FREQ1VAL	VARCHAR(12) NOT NULL FOR BIT DATA	The most frequent value in the column. If there is more than one value, the smaller is used. If the column is not a character data type, FREQ1VAL may be unprintable.
FREQ1PCT	SMALLINT NOT NULL	The percent frequency of FREQ1VAL
FREQ2VAL	VARCHAR(12) NOT NULL FOR BIT DATA	The second most frequent value in the column. If there is more than one value, the smaller is used. If the column is not a character data type, FREQ2VAL may be unprintable.
FREQ2PCT	SMALLINT NOT NULL	The percent frequency of FREQ2VAL

SYSCOLUMNS

The SYSCOLUMNS table contains a more detailed description of the database than that contained in SYSCATALOG. Recall that SYSCATALOG contains a row for each table or view in the database; SYSCOLUMNS contains a row for every *column* of every table or view in the database (including the columns of the catalog tables).

The columns in SYSCOLUMNS are:

Column Name	Data Type	Description and Comments
CNAME	VARCHAR(18) NOT NULL	The name of the column described.
TNAME	VARCHAR(18) NOT NULL	The name of the table or view in which the column (CNAME) is located.
CREATOR	CHAR(8) NOT NULL	The owner of the table or view identified by TNAME. (Thus, CNAME is the name of a column in the table or view identified by CREATOR.TNAME.)
COLNO	SMALLINT NOT NULL	The number of the column in the table. The value in COLNO corresponds to the sequence that columns are specified in the CREATE TABLE statement or added in the ALTER TABLE statement.
COLTYPE	CHAR(8) NOT NULL	The data type of the column: INTEGER, SMALLINT, CHAR, VARCHAR, LNGVCHAR, DATE, TIME, TIMESTMP, GRAPHIC, VARGRAPH, LONGVARG, FLOAT, DECIMAL, DBAINT, or DBAHW. These last two data types are for information used only internally by the database manager. (DBAINT data appears externally as INTEGER data; DBAHW data appears externally as SMALLINT data.)
LENGTH	CHAR(7) NOT NULL	The size of the column as specified in the CREATE TABLE or ALTER TABLE statements. If the column has a data type of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC, LENGTH contains the value specified on the CREATE TABLE or ALTER TABLE statements. If the data type is LONG VARCHAR, LENGTH contains 32767. If the data type is LONG VARGRAPHIC, LENGTH contains 16383. If the data type is DATE, TIME, or TIMESTAMP, LENGTH is blank. For INTEGER and SMALLINT, LENGTH is blank. For FLOAT, LENGTH contains the length supplied for FLOAT in the CREATE TABLE or ALTER TABLE statements. If no value was supplied for a FLOAT column, LENGTH is blank. If the data type is DECIMAL, the precision and scale of the column are in this field, in the form: (pp,ss). For example: (11, 2).

Column Name	Data Type	Description and Comments
SYSLENGTH	DBAHW NOT NULL	If COLTYPE is CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC, then this field contains the (maximum) length of the data, in bytes. If COLTYPE is DATE, then SYSLENGTH contains 4. If COLTYPE is TIME, then SYSLENGTH contains 3. If COLTYPE is TIMESTAMP, then SYSLENGTH contains 10. If COLTYPE is DECIMAL, then the first byte gives the number of digits in the number, and the second gives the number of digits after the decimal point. In other words, if COLTYPE is DECIMAL and LENGTH is (pp,ss), then SYSLENGTH contains (256 x pp) + ss. If COLTYPE is any one of the other numeric types, SYSLENGTH contains the number of bytes occupied by a datum of that type: 2, 4, or 8. SYSLENGTH never reflects the additional byte used internally to indicate nulls or the halfword prefix for the length of VARCHAR or VARGRAPHIC fields.
NULLS	CHAR(1) NOT NULL	The possible values are: **Y** if null values are allowed in this column. **N** if null values are not allowed in this column.
REMARKS	VARCHAR(254) NOT NULL	Information about the column supplied by a user by a COMMENT statement. The remarks are deleted from SYSCOLUMNS when the table or view is dropped. If the application server default CHARNAME setting supports mixed data (that is, CCSIDMIXED is not 0), users can store mixed data (both SBCS and DBCS characters) in the REMARKS column.
COLCOUNT	DBAINT NOT NULL	The number of unique values in the column. COLCOUNT initially contains -1 or 0. COLCOUNT captures only an approximate value for a column which is not the first column of any index. See Note 1 for update rules on this column.
HIGH2KEY	VARCHAR(12) NOT NULL FOR BIT DATA	The first eight bytes of the second highest value in the column. (This value is needed internally.) This field initially contains blanks. If COLTYPE is not CHAR, VARCHAR, LNGVCHAR, GRAPHIC, VARGRAPH, or LONGVARG, then HIGH2KEY may be unprintable. See Note 1 for update rules on this column.
LOW2KEY	VARCHAR(12) NOT NULL FOR BIT DATA	The first eight bytes of the second lowest value in this column. (This value is needed internally.) This field initially contains blanks. If COLTYPE is not CHAR, VARCHAR, LNGVCHAR, GRAPHIC, VARGRAPH, or LONGVARG, then LOW2KEY may be unprintable. See Note 1 for update rules on this column.
AVGCOLLEN	DBAHW NOT NULL	The average length of the values in this column. (This value is needed internally.) This value is initially -1. See Note 1 for update rules on this column. The value is always -1 for long fields.
ORDERFIELD	CHAR(1) NOT NULL	The possible values are: **Y** if the rows are physically clustered in accord with the values in this column. **N** when the table is created, then set to 'Y' if further information about this column exists. This value is reset to X'FF' when the index is dropped. This entry is valid only for a column in a single-column index.

Column Name	Data Type	Description and Comments
CLABEL	VARCHAR(30)	A column label supplied by a user using a LABEL statement. The column labels are deleted from SYSCOLUMNS when the table or view is dropped. If the application server default CHARNAME setting supports mixed data (that is, CCSIDMIXED is not 0), users can store mixed data (both SBCS and DBCS characters) in the CLABEL column.
COLINFO	CHAR(1)	This indicates the presence of additional information about the column which will be found in the table SYSTEM.SYSCOLSTATS. **Y** further information exists, see SYSTEM.SYSCOLSTATS. **NULL** when the table is created, then set to 'Y' if further information about this column exists. If this is the first column of an index and the index is dropped, and there is no other index with this column as the first column, this value is also set to NULL.
SUBTYPE [2]	CHAR(1)	The subtype is applicable for CHAR, VARCHAR, and LONG VARCHAR columns only. The possible values are: . **B** for bit data. **M** for mixed data. **S** for SBCS data. **NULL** if any of the following cases is true: • The column is in a table migrated from SQL/DS Version 2 Release 2 or earlier and has a data type of CHAR, VARCHAR, or LONG VARCHAR. • The data type is not character.
CCSID [3]	INTEGER	If CCSID conversion is required, it is done before data is stored in the column. CCSID is applicable for CHAR, VARCHAR, LONG VARCHAR, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC columns only. The possible values are: **1 to 65534** a valid CCSID for character or graphic data **65535** uniquely identifies bit character data **NULL** if any of the following conditions exist: • the data type is character or graphic and the column is in a table migrated from a release previous to SQL/DS Version 3 Release 1 • the data type is neither character nor graphic.
FLDPROC	CHAR(1)	Indicates whether the column has a field procedure. The possible values are: **NULL** if a column belongs to a table migrated to SQL/DS Version 3 Release 1 or later **Y** if the column has a field procedure **N** if the column does not have a field procedure.

Note 1:
The value is updated for all columns of the table by an UPDATE ALL

STATISTICS statement on the table or a DBSPACE containing the table. UPDATE STATISTICS has the same effect but only for the columns which are the first columns of an index. CREATE or REORGANIZE INDEX updates the value for one column, the first one in the index.

Note 2:
The SUBTYPE value is only used when the CCSID value is null. If a SUBTYPE is encountered that is not valid and the CCSID value is null, a SUBTYPE of SBCS is assumed.

Note 3:
More information on CCSIDs can be found in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

SYS**DBSPACES**

The SYSDBSPACES table contains a row for each PUBLIC and PRIVATE DBSPACE in the database, including those DBSPACEs that no user has yet acquired. The number of DBSPACEs available is determined during database generation. The size of each DBSPACE is also specified at that time.

Additional DBSPACEs may be added from time to time by the ADD DBSPACE operation.

The columns in SYSDBSPACES are:

Column Name	**Data Type**	**Description and Comments**
DBSPACENAME	VARCHAR(18) NOT NULL	The name given to the DBSPACE by the user who acquired it. If the DBSPACE has not been acquired, the field contains an empty string.
DBSPACENO	DBAHW NOT NULL	When the database is generated, the database manager assigns each DBSPACE a number (for internal use). DBSPACENO is that number. This is the number you would use in the SHOW DBSPACE operator command.
OWNER	CHAR(8) NOT NULL	The possible values are: **blank** if the DBSPACE is not yet assigned. **owner** if the DBSPACE is PRIVATE. **PUBLIC** if the DBSPACE is PUBLIC. **¬¬¬¬¬¬¬¬** if the DBSPACE has been dropped, but has not yet been removed from the database. In this case, DBSPACENAME will contain the number of the DBSPACE to be removed. Note that the hex value for the symbols displayed is ′5F′X. Depending on the active CHARNAME the symbol may not display as ′¬′ on your terminal.
DBSPACETYPE	SMALLINT NOT NULL	The possible values are: **1** if the DBSPACE is PUBLIC. **2** if the DBSPACE is PRIVATE.
NTABS	DBAHW NOT NULL	The number of tables contained in this DBSPACE. This field is updated when CREATE TABLE and DROP TABLE statements are issued.

Column Name	Data Type	Description and Comments
NPAGES	INTEGER NOT NULL	The number of usable pages in the DBSPACE. NPAGES is specified in the PAGES parameter of the ACQUIRE DBSPACE statement. **CAUTION:** **Changing NPAGES makes the dbspace appear to be a different size without actually changing it. NPAGES should not be changed in a production environment, to do so may cause errors to occur. It is intended for testing purposes only.**
NRHEADER	DBAHW NOT NULL	The number of pages to be used for the DBSPACE header. This number must be between 1 and 8. NRHEADER is specified in the NHEADER parameter of the ACQUIRE DBSPACE statement.
PCTINDX	DBAHW NOT NULL	The percentage of pages to be used for indexes. PCTINDX is specified in the PCTINDEX parameter of the ACQUIRE DBSPACE statement.
FREEPCT	SMALLINT NOT NULL	The percentage of space on each page to be kept free when rows are inserted. Initially, the database manager gets the value for FREEPCT from the PCTFREE parameter of the ACQUIRE DBSPACE statement. FREEPCT can be updated by the PCTFREE parameter of the ALTER DBSPACE statement. For package DBSPACEs, FREEPCT is either 0 or 1. A FREEPCT of 0 indicates that the package DBSPACE is full. A FREEPCT of 1 indicates that it is not full.
LOCKMODE	CHAR(1) NOT NULL	The possible values are: **S** if the entire DBSPACE is to be locked. **P** if page locking is to be done in this DBSPACE. **T** if row locking is to be done in this DBSPACE. LOCKMODE is updated by the LOCK parameter of the ACQUIRE DBSPACE and ALTER DBSPACE statements.
NACTIVE	DBAINT NOT NULL	The number of active data pages in this DBSPACE (set to -1 when the DBSPACE is acquired). This value is the number of data pages that must be read for a complete DBSPACE scan. The value includes all data pages that contain stored rows for this DBSPACE. NACTIVE is set by an UPDATE STATISTICS statement issued for this DBSPACE, or for any table in this DBSPACE, or by creating or reorganizing any index in this DBSPACE, or by the DATALOAD/RELOAD DBS Utility commands if the statistics collection has not been turned off by SET UPDATE STATISTICS OFF. NACTIVE is set to zero for an UPDATE STATISTICS statement issued on a package DBSPACE.
POOL	DBAHW NOT NULL	The number of the storage pool into which the database manager places pages that belong to this DBSPACE. POOL is specified using the STORPOOL parameter of the ACQUIRE DBSPACE statement. If the value of POOL is negative, the storage pool is nonrecoverable. (The absolute value of POOL is the storage pool number.)

SYSDROP

SYSDROP contains a list of tables and DBSPACEs waiting to be dropped. The database manager uses this table when tables or DBSPACEs are dropped from the database. When a DBSPACE or table is dropped, its description is dropped from the catalog immediately, but the object is not dropped until the end of a logical unit of work (LUW). Instead, the database manager makes an entry in SYSDROP identifying the dropped table or DBSPACE. When the LUW is committed (implicitly or explicitly), all objects identified in SYSDROP are dropped. This allows the LUW containing the DROP statement to proceed without waiting on any locks held against the object being dropped, while guaranteeing that the object will be dropped. It also minimizes the performance cost if the LUW must be rolled back. Any LUW accessing the table or DBSPACE when the DROP statement is issued will complete, but no further access to the object is possible. For information on diagnosing problems associated with the DROP statement, see the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual.

Column Name	Data Type	Description and Comments
DBSPACENO	DBAHW NOT NULL	The internal number of the DBSPACE containing an object to be dropped.
TABID	DBAHW NOT NULL	The internal identifier of a table to be dropped. (In the *DB2 Server for VSE & VM Diagnosis Guide and Reference* manual, an internal table identifier is known as a DBSS RID.)
QUALF	CHAR(1) NOT NULL	The possible values are: **S** if the object to be dropped is a DBSPACE. **T** if the object to be dropped is a table.

SYSFIELDS

The SYSFIELDS table contains a row for each column that has a field procedure associated with it. The columns in SYSFIELDS are:

Column Name	Data Type	Description and Comments
CREATOR	CHAR(8) NOT NULL	The owner of the package or view who created the table containing the column with the field procedure.
TNAME	VARCHAR(18) NOT NULL	The name of the table containing this column.
COLNO	SMALLINT NOT NULL	The number of this column in the table.
CNAME	VARCHAR(18) NOT NULL	The name of this column
FLDTYPE	CHAR(8) NOT NULL	The data type of the encoded value in the field. Possible values are: **INTEGER** for large integer. **SMALLINT** for small integer. **DECIMAL** for decimal. **FLOAT** for floating-point. **CHAR** for fixed length character string. **VARCHAR** for varying length character string. **GRAPHIC** for fixed length graphic string. **VARGRAPH** for varying length graphic string.

Column Name	Data Type	Description and Comments
FLDLENGTH	SMALLINT NOT NULL	The length attribute of the field, or the precision for decimal fields. The number does not include the internal prefixes that may be used to record the actual length and null state. The value in this column depends on the data type of the field as follows: **For INTEGER** 4 **For SMALLINT** 2 **For DECIMAL** 1 byte - precision of number 1 byte - scale of number **For FLOAT** 8 **For CHAR** Length of the string **For VARCHAR** Maximum length of the string **For GRAPHIC** Number of DBCS characters **For VARGRAPHIC** Maximum number of DBCS characters.
FPNAME	CHAR(8) NOT NULL	Name of the field procedure. Field procedure names are unique within an installation.
FPWORKAREA	SMALLINT NOT NULL	Size, in bytes, of the work area required for the encoding and decoding functions of the field procedure.
FPEXITPARML	SMALLINT NOT NULL	Length of the field procedure parameter value block.
FPPARMLIST	VARCHAR(254) NOT NULL	The parameter list given after FIELDPROC in the statement that created the column. Insignificant blanks are removed.

SYSFPARMS

The SYSFPARMS table holds the field procedure value block contents for each field procedure. Blocks longer than 254 characters will have more than one row in the table. All field procedure value blocks of length greater than 0 will be recorded in this table. The columns in SYSFPARMS are:

Column Name	Data Type	Description and Comments
FPNAME	CHAR(8) NOT NULL	Name of the field procedure. Field procedure names are unique within an installation.
CREATOR	CHAR(8) NOT NULL	The owner of the package or view who created the table that contains the column with the field procedure.
TNAME	VARCHAR(18) NOT NULL	Name of the table that contains the column with the field procedure.
CNAME	VARCHAR(18) NOT NULL	Name of the column that has the field procedure.
SEQNO	SMALLINT NOT NULL	Indicates the sequence of the portion of the parameter value block contained in this row. A long block may be divided among several rows of the SYSFPARMS table. The value for the first portion of a block is 1. Successive rows have sequential values.
FPEXITPARM	VARCHAR(254) NOT NULL	The parameter value block (or a portion of it) of the field procedure. This control block is passed to the field procedure when it is invoked.

SYSINDEXES

The SYSINDEXES table contains a row for every index currently in existence, including the indexes that the database manager maintains on its own catalog tables.

The columns in SYSINDEXES are:

Column Name	Data Type	Description and Comments
INAME	VARCHAR(18) NOT NULL	The name of the index.
ICREATOR	CHAR(8) NOT NULL	The user ID of the person who created the index. The combination of INAME and ICREATOR uniquely identifies the index.
TNAME	VARCHAR(18) NOT NULL	The table on which the index is defined.
CREATOR	CHAR(8) NOT NULL	The owner of the package or view who created the table on which the index is defined.
COLNAMES	VARCHAR(100) NOT NULL	This contains the first 100 characters of the names of the columns on which the index is defined. Each name is preceded by + (for ascending) or - (for descending), and separated by commas and blanks. For example: `+AGE, +SALARY, -NEXM`
INDEXTYPE	CHAR(1) NOT NULL	The possible values are: **U** if the index is unique (duplicates not allowed). **D** if duplicates are allowed.
CLUSTER	CHAR(1) NOT NULL	The possible values are: **C** if the index is clustered. **N** if the index is not clustered. **F** if the index was the first index created and is now clustered (used for default insert clustering). **W** if the index was the first index created and is now *not* clustered (still the default insert index). **blank** if this is an inactive primary key index. The value of CLUSTER is not directly related to the CLUSTERRATIO value. See Note 1 for update rules on this column.
IID	DBAHW NOT NULL	The internal index identifier assigned to the index by DBSS.
COLNUMBERS	VARCHAR(34) NOT NULL FOR BIT DATA	An indicator array of binary integers of 15 bits (plus sign); it has one more element than the number of columns in the index. The first element is the number of columns in the index. The second element is the column number defining the major ordering of the index; the remaining elements define the minor orders of the index. The column number is positive if the index is ascending on that column, and negative if it is descending. Each of these binary integer halfwords is stored in internal format. The size of this field restricts index definitions to 16 columns.

Column Name	Data Type	Description and Comments
KEYLEN	DBAHW NOT NULL	This is used internally by the database manager. It is the average length of the key field. See Note 1 for update rules on this column.
FIRSTKEYCOUNT	DBAINT NOT NULL	This is used internally by the database manager. It gives the number of distinct values for the index, considering the first column only. See Note 1 for update rules on this column.
FULLKEYCOUNT	DBAINT NOT NULL	This is used internally by the database manager. It gives the number of distinct values for the index, considering all key columns. See Note 1 for update rules on this column.
LOCKMODE	CHAR(1) NOT NULL	This is used internally by the database manager. It is: **K** if key-interval locking is being performed on the index. **P** if the pages of the index are being locked. LOCKMODE is updated using the LOCK parameter of the ACQUIRE DBSPACE and ALTER DBSPACE statements. When you specify LOCK=ROW on either the ACQUIRE DBSPACE or ALTER DBSPACE statements, the database manager internally uses key-interval locking for that DBSPACE. (Note that this applies only to PUBLIC DBSPACEs because you cannot specify a different lock size for PRIVATE DBSPACEs.)
NLEAF	DBAINT NOT NULL	This is used internally by the database manager. It is the number of lowest-level pages in the index. See Note 1 for update rules on this column.
NLEVELS	DBAHW NOT NULL	This is used internally by the database manager. It is the number of levels in the index tree. See Note 1 for update rules on this column.
IPCTFREE	SMALLINT NOT NULL	The amount of free space reserved in the index for later insertions and updates. IPCTFREE is specified in the CREATE INDEX statement and the REORGANIZE INDEX command via the PCTFREE parameter.
CLUSTERRATIO	SMALLINT	This is used internally by the optimizer. The value here is a measure of how clustered an index is. The value is a number between 0 and 10 000 where 10 000 represents a totally clustered index and 0 represents a totally unclustered index. The value in this column is not directly related to the CLUSTER value. See Note 1 for update rules on this column.
RELEASE	CHAR(5)	This column identifies the release for which the index was created. It contains the value “2.1.0” for any release up to and including SQL/DS Version 2 Release 1. For later releases, possible values are: • 2.2.0 for SQL/DS Version 2 Release 2 • 3.1.0 for SQL/DS Version 3 Release 1 • 3.2.0 for SQL/DS Version 3 Release 2 • 3.3.0 for SQL/DS Version 3 Release 3 • 3.4.0 for SQL/DS Version 3 Release 4 • 3.5.0 for SQL/DS Version 3 Release 5 • 5.1.0 for DB2 Server for VSE & VM Version 5 Release 1 • 6.1.0 for DB2 Server for VSE & VM Version 7 Release 5 Non-unique indexes created under SQL/DS Version 2 Release 2 or later have better performance characteristics than non-unique indexes from previous releases.

Column Name	Data Type	Description and Comments
KEYTYPE	CHAR(1)	This identifies whether the index is used for a primary key. The possible values are: **P** if the index was created for a primary key which is active. **I** if the index was created for a key which is inactive. **U** if the index was created for a unique constraint which is active. **blank** if the index was not created for a primary key or unique constraint.

Note 1: The value is updated for all indexes on a table or in the DBSPACE by the UPDATE STATISTICS and UPDATE ALL STATISTICS statements. CREATE and REORGANIZE INDEX updates the value for the created or reorganized index.

SYSKEYCOLS

This table contains a row for every column in every key.

Column Name	Data Type	Description and Comments
TNAME	VARCHAR(18) NOT NULL	The name of the table on which the key is defined.
TCREATOR	CHAR(8) NOT NULL	The owner of the package or view who created the table on which the key is defined.
KEYTYPE	CHAR(1) NOT NULL	The possible values are: **P** primary key **F** foreign key **U** unique constraint
KEYNAME	CHAR(18) NOT NULL	This is the key name specified in the FOREIGN KEY clause (KEYTYPE = F) or the constraint name specified in the UNIQUE clause (KEYTYPE = U). If the key or constraint name is not specified or it is a primary key, the system generated name will be stored here.
CNAME	VARCHAR(18) NOT NULL	The column name.
KEYORD	SMALLINT NOT NULL	The position of the column within the key.
TABLEORD	SMALLINT NOT NULL	The position of the column within the table.
DATACODE	SMALLINT NOT NULL	The data type of the column in internal form.
SYSLENGTH	SMALLINT NOT NULL	This contains the length of the column. Its interpretation is the same as the SYSLENGTH column in the SYSCOLUMNS table.
TIMESTAMP	TIMESTAMP NOT NULL	The date and time when this key was activated.

Column Name	Data Type	Description and Comments
FLDPROC	CHAR(1)	Indicates whether this column has a field procedure associated with it. Possible values are: **Y** if yes. **N** if no. **NULL** if the table containing the key was migrated from SQL/DS Version 2 Release 2 or earlier.
CCSID [1]	INTEGER	If CCSID conversion is required, it is done before data is stored in the column. CCSID is applicable for CHAR, VARCHAR, LONG VARCHAR, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC columns only. The possible values are: **1 to 65534** a valid CCSID for character or graphic data **65535** uniquely identifies bit character data **NULL** if any of the following conditions exist: • the data type is character or graphic and the column is in a table migrated from a release previous to SQL/DS Version 3 Release 1 • the data type is neither character nor graphic.

Note 1:
More information on CCSIDs can be found in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

SYSKEYS

This table contains a row for each primary key, each foreign key, and each unique constraint.

Column Name	Data Type	Description and Comments
TNAME	VARCHAR(18) NOT NULL	The name of the table on which the key or constraint is defined.
TCREATOR	CHAR(8) NOT NULL	The owner of the package or view who created the table on which the key or constraint is defined.
KEYTYPE	CHAR(1) NOT NULL	The possible values are: **P** primary key **F** foreign key **U** unique constraint
KEYNAME	CHAR(18) NOT NULL	This is the key name specified in the FOREIGN KEY clause (KEYTYPE = F) or the constraint name specified in the UNIQUE clause (KEYTYPE = U). If the key name or the unique constraint name is not specified or it is a primary key, the system generated name will be stored here. The format of the generated name is 'PKEY', 'FKEY' or 'UKEY' followed by a special 12-bytes timestamp. The timestamp is the value of the System/390 time of day clock when the key is defined and it is a string of numbers and letters in base 35 representation. It is the same type of timestamp that the database manager uses in the SYSTABAUTH table.

Column Name	Data Type	Description and Comments
KEYCOLS	SMALLINT NOT NULL	This is the number of columns that form the primary or foreign key, or unique constraint.
INAME	VARCHAR(18) NOT NULL	For a primary key this contains the name of the primary key index. For a foreign key, this field is blank. For a unique constraint this contains the name of the index.
REFTNAME	VARCHAR(18) NOT NULL	For a foreign key, this field contains the name of the parent table. For a primary key and unique constraint, this field is blank.
REFTCREATOR	CHAR(8) NOT NULL	For a foreign key, this field contains the owner of the package or view who created the parent table. For a primary key and unique constraint this field is blank.
DELETERULE	CHAR(1) NOT NULL	For a foreign key, this column gives the associated DELETE rule. The possible values are: **R** if the delete rule is RESTRICT. **C** if the delete rule is CASCADE. **N** if the delete rule is SET NULL. For a primary key and unique constraint this field is blank.
STATUS	CHAR(1) NOT NULL	The current status of the key. The possible values are: **A** if the key is active. **I** if the key is inactive. **D** if the foreign key is implicitly inactive. (Dependent on an inactive primary key).
TIMESTAMP	TIMESTAMP NOT NULL	The date and time when this key or constraint was activated.

SYSLANGUAGE

The SYSLANGUAGE table contains the names of all currently installed national languages (for example, English or French); that is, it is not a programming language such as COBOL. A unique four-character code identifies each language and a brief description, if necessary, is contained in the REMARKS column.

Unlike all other catalog tables, the owner of SYSLANGUAGE is SQLDBA. To view SYSLANGUAGE use the statement:

```
SELECT * FROM SQLDBA.SYSLANGUAGE
```

The columns in SYSLANGUAGE are:

Column Name	Data Type	Description and Comments
LANGUAGE	VARCHAR(40) NOT NULL	The name of the national language.
LANGKEY	CHAR(4) NOT NULL	The language key.
REMARKS	VARCHAR(254)	Comments or description of the language.
LANGID	VARCHAR(5)	The VM-compatible language ID of the installed language.

SYSOPTIONS

The SYSOPTIONS table describes the options and defaults that may be implemented for this database. The table summarizes the information contained in the text that follows the table.

Column Name	Data Type	Description and Comments
SQLOPTION	VARCHAR(18) NOT NULL	The name of the option being described by this row. SQLOPTION can be: **RELEASE** if this row describes the release level of the database manager. **CHARNAME** if this row describes the name of the character set that is currently in effect. **DBCS** if this row describes the setting of the DBCS option. **CHARSUB** if this row describes the default subtype for character columns. **DATE** if this row describes the default format of DATE for the database manager. **TIME** if this row describes the default format of TIME for the database manager. **LDATELEN** if this row describes the length of the local (user defined) DATE format. **LTIMELEN** if this row describes the length of the local (user defined) TIME format. **DEFAULT LANGUAGE** if this row describes the default language used for ISQL HELP text. **CCSIDSBCS** if this row describes the default CCSID for SBCS character data and newly-created SBCS character columns. **CCSIDMIXED** if this row describes the default CCSID for mixed character data and newly-created mixed character columns. **CCSIDGRAPHIC** if this row describes the default CCSID for graphic data and newly-created graphic columns. **MCCSIDSBCS** if this row describes the default CCSID for migrated SBCS character columns. **MCCSIDMIXED** if this row describes the default CCSID for migrated mixed character columns. **MCCSIDGRAPHIC** if this row describes the default CCSID for migrated graphic columns.

SYSOPTIONS

Column Name	Data Type	Description and Comments
VALUE	VARCHAR(18) NOT NULL	This describes the option. If SQLOPTION is **RELEASE**, then VALUE indicates the release level, such as "7.5.0" for DB2 Server for VSE & VM Version 6 Release 1. If SQLOPTION is **CHARNAME**, then VALUE indicates the value of CHARNAME that was specified when the database manager was last started. The database management system is shipped with CHARNAME set to INTERNATIONAL. (Character set information is stored in the SYSCHARSETS catalog table.) If SQLOPTION is **DBCS**, then VALUE indicates whether the DBCS option is enabled. YES, in this case, indicates that the DBCS option is enabled. NO indicates that it is not enabled. Note that a "YES" does not necessarily mean that the DBCS option is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the DBCS option will not take effect until the database manager is restarted. If SYSOPTION is **CHARSUB**, then VALUE indicates the default character subtype to be used for the database. The default is SBCS. MIXED is the other possible value. Note that the value indicated here does not mean that value is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the value indicated will not be in effect until the database manager is restarted. If SQLOPTION is **DATE**, then VALUE indicates the data format to be used for the database. The default for DATE is ISO. However, JIS, USA, EUR or LOCAL may be used. Note that the value indicated here does not necessarily mean that the value indicated for the option is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the value indicated will not be in effect until the database manager is restarted. If SQLOPTION is **TIME**, then VALUE indicates the TIME format to be used for the database. The default for TIME is ISO. However, JIS, USA, EUR or LOCAL may be used. Note that the value indicated here does not necessarily mean that the value indicated for the option is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the value indicated will not be in effect until the database manager is restarted. If SQLOPTION is **LDATELEN**, then VALUE indicates the length of the local DATE format. The default for LDATELEN is 0, if no local DATE format is used. If a local DATE format is used, the length LDATELEN must be greater than 9 and less than 255. Note that the value indicated here does not necessarily mean that the value indicated for the option is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the value indicated will not be in effect until the database manager is restarted. If SQLOPTION is **LTIMELEN**, then VALUE indicates the length of the local TIME format. The default for LTIMELEN is 0, if no local TIME format is used. If a local TIME format is used, the length LTIMELEN must be greater than 7 and less than 255. Note that the value indicated here does not necessarily mean that the value indicated for the option is currently in effect. It could be that the database administrator just updated the value. If the value was just updated, then the value indicated will not be in effect until the database manager is restarted. If SQLOPTION is **DEFAULT LANGUAGE**, then VALUE indicates the default language to be used for ISQL HELP text. If SQLOPTION is **CCSIDSBCS**, then VALUE indicates the default CCSID for SBCS character data and newly-created SBCS character columns. If SQLOPTION is **CCSIDMIXED**, then VALUE indicates the default CCSID for mixed character data and newly-created mixed character columns. If SQLOPTION is **CCSIDGRAPHIC**, then VALUE indicates the default CCSID for graphic data and newly-created graphic columns. If SQLOPTION is **MCCSIDSBCS**, then VALUE indicates the default CCSID for migrated SBCS character columns. If SQLOPTION is **MCCSIDMIXED**, then VALUE indicates the default CCSID for migrated mixed character columns. If SQLOPTION is **MCCSIDGRAPHIC**, then VALUE indicates the default CCSID for migrated graphic columns.
REMARKS	VARCHAR(254) NOT NULL	This contains remarks describing each row. The database manager places remarks in this column when the SYSOPTIONS table is created.

The following table shows the actual entries you would see in a newly-installed SYSOPTIONS table.

SQLOPTION	VALUE	REMARKS
RELEASE	7.5.0	VERSION, RELEASE, MODIFICATION
CHARNAME	INTERNATIONAL [1]	CHARACTER SET FOR SQL STATEMENTS
DBCS	NO [1]	WHETHER SO/SI CHARACTERS ARE RECOGNIZED
CHARSUB	SBCS	DEFAULT CHARACTER SUBTYPE COLUMNS. POSSIBLE VALUES: SBCS,MIXED
DATE	ISO	DEFAULT DATE: ISO, JIS, USA, EUR, LOCAL
TIME	ISO	DEFAULT TIME: ISO, JIS, USA, EUR, LOCAL
LDATELEN	0	LOCAL DATE LENGTH: 0 OR 9 < LEN < 255
LTIMELEN	0	LOCAL TIME LENGTH: 0 OR 7 < LEN < 255
DEFAULT LANGUAGE	S001	DEFAULT LANGUAGE FOR HELP TEXT
CCSIDSBCS	500	DEFAULT CCSID FOR SBCS DATA AND NEWLY CREATED SBCS CHARACTER COLUMNS [2]
CCSIDMIXED	0	DEFAULT CCSID FOR MIXED DATA AND NEWLY CREATED MIXED CHARACTER COLUMNS
CCSIDGRAPHIC	0	DEFAULT CCSID FOR GRAPHIC DATA AND NEWLY CREATED GRAPHIC COLUMNS
MCCSIDSBCS	37 [3]	DEFAULT CCSID FOR MIGRATED SBCS CHARACTER COLUMNS
MCCSIDMIXED	0	DEFAULT CCSID FOR MIGRATED MIXED CHARACTER COLUMNS
MCCSIDGRAPHIC	0	DEFAULT CCSID FOR MIGRATED GRAPHIC COLUMNS

Note 1:
The CHARNAME value must be a mixed CHARNAME and DBCS value must be set to YES for DBCS character support.

Note 2:
More information on CCSIDs can be found in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

Note 3:
Though 37 is the default CCSID migration value, 500 is the installation default.

SYSPARMS

The SYSPARMS table describes the parameters for the stored procedures defined. It contains a row for each parameter of each stored procedure. Table Table 24 shows the definition of this catalog table.

Table 24. Definition of SYSTEM.SYSPARMS

Column Name	Data Type	Description
NAME	CHAR(18) NOT NULL	The name of the STORED procedure with which this parameter is associated.
AUTHID	CHAR(8) NOT NULL	The authorization ID associated with this version of the stored procedure. See Table 28 on page 400 for an example of using the AUTHID column.

Table 24. Definition of SYSTEM.SYSPARMS (continued)

Column Name	Data Type	Description
PARMNAME	CHAR(18) NOT NULL	The name of the parameter, or blank. This column is included for compatibility with other database products. It is ignored by DB2 Server for VSE & VM.
ROUTINEID	INTEGER NOT NULL	Internal identifier of the stored procedure.
ROWTYPE	CHAR(1) NOT NULL	The type of the parameter. Possible values are: **P** Input parameter; **O** Output parameter; **B** Both input and output
ORDINAL	SMALLINT NOT NULL	The ordinal number of the parameter within the parameter list.
TYPENAME	CHAR(18) NOT NULL	The name of the data type of the parameter.
DATATYPEID	SMALLINT NOT NULL	The internal ID of the data type of the parameter.
LENGTH	INTEGER NOT NULL	Maximum length of the data type or the precision of the parameter.
SCALE	SMALLINT NOT NULL	Scale of the parameter, if the data type is decimal. 0 otherwise.
SUBTYPE	CHAR(1) NOT NULL	If the data type of the parameter is character, this column contains the character subtype. Possible values are: **B** The subtype is FOR BIT DATA. **S** The subtype is FOR SBCS DATA. **M** The subtype is FOR MIXED DATA. **blank** The data type of the parameter is not character.
CCSID	INTEGER NOT NULL	For all character and graphic data types, this column contains the CCSID that the stored procedure assumes this parameter will be tagged with. 0 for datatypes other than character and graphic.

SYSPROGAUTH

SYSPROGAUTH records privileges of users to run packages, and to grant these privileges to other users. For the DB2 Server for VSE & VM database manager, a program is a package stored in the database. The columns in SYSPROGAUTH are:

Column Name	Data Type	Description and Comments
GRANTOR	CHAR(8) NOT NULL	The user ID of the person who granted the RUN privilege.
GRANTEE	CHAR(8) NOT NULL	The user ID of the person who holds the RUN privilege. If the *userid* is PUBLIC, the program may be run by all users.
CREATOR	CHAR(8) NOT NULL	The owner who preprocessed the program. CREATOR.PROGNAME uniquely identifies the package that may be run by the grantee.

Column Name	Data Type	Description and Comments
PROGNAME	VARCHAR(8) NOT NULL	The name of the package that may be run by the grantee. The name is obtained from the PREPNAME preprocessor parameter. CREATOR.PROGNAME is the complete name of the package.
TIMESTAMP	CHAR(12) NOT NULL	The value of the System/390 time of day clock when the grant was made. This value is used internally when privileges are revoked; it is stored as a string of numbers and letters.
RUNAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the user is allowed only to run the package. **G** if the user may also grant the RUN privilege on the package to someone else.

SYSPSERVERS

SYSTEM.SYSPSERVERS is added to allow the database administrator to define the stored procedure servers at which stored procedure run, and to put them in groups. This allows the database administrator to tune the stored procedure workload. This table is unique to DB2 Server for VSE & VM. Table Table 25 shows the definition of this catalog table.

Table 25. Definition of SYSTEM.SYSPSERVERS

Column Name	Data Type	Description
PSERVER	CHAR(8) NOT NULL	The name of the stored procedure server. This name must not contain any embedded blanks. If it does, any attempt to start the stored procedure server fails. Note that a PSERVER can be in only one group. See “SYSPSERVERS” for more information on the PSERVER column.
SERVGROUP	CHAR(18)	The group that this server is in. Grouping the stored procedures enables the database administrator to tune the stored procedure workload. For example, if the database manager wanted to dedicate servers ACT1, ACT2, and ACT3 to accounting-related stored procedures, he could define them all in the same group, perhaps called ACCOUNT. In SYSTEM.SYSROUTINES, the row for any accounting-related stored procedure would specify ACCOUNT in the SERVGROUP column. When one of these stored procedures is invoked, the database manager will select a free server from the ACCOUNT group to run the stored procedure. Any server for which the SERVGROUP column is NULL is in the default server group.
AUTOSTART	CHAR(1)	Indicates whether the stored procedure server should be started when the database manager is started. A value of ’Y’ indicates that the server should be autostarted. ’N’ or NULL indicates that it should not be autostarted. The default value for this column is NULL.

Table 25. Definition of SYSTEM.SYSPSERVERS (continued)

Column Name	Data Type	Description
DESCRIPTION	CHAR(254)	An optional column, in which the database manager can provide information such as the stored procedures that use this server group, specifications (for example virtual storage requirements) for servers in this group, and so on. The default value for this column is NULL.

Table 26 shows an example of a SYSTEM.SYSPSERVERS table.

Table 26. Sample SYSTEM.SYSPSERVERS Table

	PSERVER	SERVGROUP	AUTOSTART	DESCRIPTION
1	PROCSRV1		Y	Default server group
2	PROCSRV2	BILLING	Y	
3	PROCSRV3	BILLING	Y	
4	PROCSRV4	DAY_RPT	Y	

In Table 26, the first row identifies the only server in the default server group. The second and third rows identify the servers that are in the group BILLING. The fourth row identifies the only server in the group DAY_RPT.

SYSROUTINES

SYSROUTINES allows the database administrator to specify the load module or phase name and package name for a given stored procedure, and to specify the stored procedure server at which it will run. Note that several of the columns in SYSTEM.SYSROUTINES in DB2 Server for VSE & VM correspond to columns in stored procedure related catalog tables in DB2 for MVS and have been given the same name. However, both tables have system-unique columns, and as a result the definitions of the tables are not identical. Table 27 shows the definition of SYSTEM.SYSROUTINES.

Table 27. Definition of SYSTEM.SYSROUTINES

Column Name	Data Type	Description
NAME	CHAR(18) NOT NULL	The name of the STORED procedure. This is the name that is specified in the SQL CALL statement.
AUTHID	CHAR(8) NOT NULL	The authorization ID that will be running this stored procedure. The AUTHID column can be used to qualify which SYSTEM.SYSROUTINES row is used to determine the load module, run time options, and so on, to use when a particular stored procedure is invoked. Possible reasons to use the AUTHID column include: • To restrict the use of a stored procedure to a particular authorization ID • To enable a particular authorization ID to test a new version of a stored procedure • To allow different authorization IDs to use different versions of a stored procedure If AUTHID for a stored procedure is blank, any authorization ID can run that stored procedure. See Table 28 on page 400 for an example of using the AUTHID column.

Table 27. Definition of SYSTEM.SYSROUTINES (continued)

Column Name	Data Type	Description
LOADMOD	CHAR(8) NOT NULL	The name of the load module or phase associated with the stored procedure.
ROUTINEID	INTEGER NOT NULL	Internal identifier of the routine.
PARMCOUNT	SMALLINT NOT NULL	The number of parameters for the routine.
LANGUAGE	CHAR(8) NOT NULL	Specifies the programming language used to create the stored procedure. Possible values are ASSEMBLE, PLI, COBOL, and C.
PARAMETERSTYLE	CHAR(1) NOT NULL	Specifies which parameter linkage convention should be used for this stored procedure. There are two possibilities: **blank** The GENERAL linkage convention is used. When a stored procedure is called, input parameters cannot be null. Also, the stored procedure cannot nullify output parameters. **N** The GENERAL WITH NULLS linkage convention is used. The input parameters can be null, and an array of indicator variables is passed to the stored procedure by DB2 Server for VSE & VM This is the default.
STAYRESIDENT	CHAR(1) NOT NULL	Determines whether the stored procedure load module or phase is removed from memory when the stored procedure ends. Possible values are **Y** The load module or phase remains in memory after the stored procedure ends. **blank** The load module or phase is removed from memory after the stored procedure ends.
PROGRAMTYPE	CHAR(1) NOT NULL	Indicates whether the stored procedure runs as a main routine or as a subroutine. The possible values are: **M** The routine runs as a LE main routine. **S** The routine runs as a LE subroutine.
COMMITON RETURN	CHAR(1) NOT NULL	If 'Y', a COMMIT WORK will be issued on return from the stored procedure. The default is 'N'. Note that since DB2 Server for VSE & VM does not have CURSOR WITH HOLD support, any cursors that are open on return will be closed if COMMITONRETURN is 'N'. This means that stored procedures that are to return result sets must have a value of 'N' in this column.
RESULTSETS	SMALLINT NOT NULL	Specifies the maximum number of result sets that the stored procedure can return to a DRDA client. A value of 0 indicates that no result sets will be returned.
SERVGROUP	CHAR(18)	Contains the name of the group of stored procedure servers that is used to run this stored procedure. The servers are defined in the SYSTEM.SYSPSERVERS catalog table, in the SERVGROUP column. If this column is blank or NULL, the stored procedure must run in the default server group. This implies that the column DEFSERV cannot contain the value 'N' in a row in which the column SERVGROUP is blank or NULL.

Table 27. Definition of SYSTEM.SYSROUTINES (continued)

Column Name	Data Type	Description
DEFSERV	CHAR(1)	Determines whether the stored procedure can run in the default stored procedure server group. **'Y' or NULL** Indicates that the procedure can run in the default server group. **'N'** Indicates that the procedure cannot run in the default server group. If this column contains 'N' then the SERVGROUP column must contain the name of a stored procedure server group.
RUNOPTS	VARCHAR(254) NOT NULL	The IBM Language Environment run-time options to use for this stored procedure. If RUNOPTS is blank, the installation default IBM Language Environment run-time options are used.
REMARKS	VARCHAR(254) NOT NULL	A character string provided by the user with the COMMENT ON statement.

Table 28 shows an example of a SYSTEM.SYSROUTINES table.

Table 28. Sample SYSTEM.SYSROUTINES Table

	PROCEDURE	AUTHID	SPECIFICNAME	SERVGROUP	DEFSERV
1	PROC1		PROG1	GROUP1	Y
2	PROC1	USER1	PROG2	GROUP1	N
3	PROC2	USER2	PROG3	Y	

Note that in Table 28 rows 1 and 2 refer to the PROC1 stored procedure. By creating multiple rows in the SYSTEM.SYSROUTINES table with the same value for the NAME column, you can indicate that specified users have access to different versions of the stored procedure. In this case, in row 1 the AUTHID column is blank. Any user without a specific entry can use row 1. Row 2 applies only to SQL CALL requests coming from AUTHID USER1. When this user invokes the PROC1 stored procedure, a different load module or phase (PROG2) is loaded. The load module or phase can be a test version of the stored procedure or a version that is specific for that user.

Row 3 applies to stored procedure PROC2 and AUTHID USER2. Because there is no other row for stored procedure PROC2, user USER2 is the only one who can call this stored procedure.

As shown in Table 28, it is possible to have more than one row in SYSTEM.SYSROUTINES for a given stored procedure. The search precedence used to determine which row is selected for a specific client is as follows:

1. A row with AUTHID matching the caller's AUTHID
2. A row with AUTHID blank

SYS**STRINGS**

The SYSSTRINGS table contains a list of the valid conversion combinations of source and target CCSIDs. More information on CCSIDs can be found in the *DB2 Server for VM System Administration* or the *DB2 Server for VSE System Administration* manual.

Column Name	Data Type	Description and Comments
INCCSID	INTEGER NOT NULL	The CCSID of the string that is a candidate for conversion.
OUTCCSID	INTEGER NOT NULL	The CCSID to which the string is to be converted.
TRANSTYPE	CHAR(2) NOT NULL	Classifies the CCSIDs as follows: **SS** for EBCDIC and ASCII SBCS to EBCDIC SBCS data conversion; **SM** for EBCDIC and ASCII SBCS to EBCDIC mixed data conversion; **MS** for EBCDIC mixed to EBCDIC SBCS data conversion; **MM** for EBCDIC mixed to EBCDIC mixed data conversion; **PS** for ASCII mixed to EBCDIC SBCS data conversion; **PM** for ASCII mixed to EBCDIC mixed data conversion; **GG** for ASCII graphic to EBCDIC graphic data conversion; **US** for UCS-2 to EBCDIC SBCS data conversion; **UI** for UCS-2 to a single byte component of an EBCDIC mixed data conversion; **UM** for UCS-2 to EBCDIC mixed data conversion; **UG** for UCS-2 to EBCDIC graphic data conversion
ERRORBYTE	CHAR(1) FOR BIT DATA	Specifies the byte that is used in the conversion table as an error indicator. An error occurs whenever a code point maps to the byte specified in this field.[1] Null indicates the absence of an error indicator.
SUBBYTE	CHAR(1) FOR BIT DATA	Specifies the byte that is used in the conversion table as a substitution character.[2] Null indicates the absence of a substitution character.
TRANSPROC	CHAR(8) NOT NULL	Name of the conversion procedure for MM, PM and GG TRANSTYPE. This procedure only applies to the DBCS portion of mixed data for the MM and PM TRANSTYPE.

Column Name	Data Type	Description and Comments
TRANSTAB1	CHAR(64)	The first 64 bytes of the 256-byte conversion table or a blank. Used for SS, SM, MM, MS, PS and PM TRANSTYPEs.
TRANSTAB2	CHAR(192)	The last 192 bytes of the 256-byte conversion table or a blank. If either TRANSTAB1 or TRANSTAB2 contains an empty string, then both are considered an empty string. Used for SS, SM, MM, MS, PS and PM TRANSTYPEs.

Note 1:
If ERRORBYTE is X'3E', for example, that error byte indicates that no conversion is defined for the code points that map to X'3E'. An error (-330 or -331 assigned to SQLCODE and 22021 assigned to SQLSTATE) or warning (+331 assigned to SQLCODE and 01520 assigned to SQLSTATE) occurs whenever a code point maps to it.

Note 2:
If SUBBYTE is X'3F', for example, that byte is substituted for the code points that map to X'3F'. A warning (Z assigned to SQLWARN8, W assigned to SQLWARN0, and 01517 assigned to SQLSTATE) occurs whenever a code point maps to it.

SYS**SYNONYMS**

The SYSSYNONYMS table contains a row for every synonym currently in effect. (A synonym is effective *only* for the user who defined it.) The columns in SYSSYNONYMS are:

Column Name	Data Type	Description and Comments
USERID	CHAR(8) NOT NULL	The owner who defined the synonym. The synonym is effective for this user only.
ALTNAME	VARCHAR(18) NOT NULL	The user's synonym for a table or view. USERID.ALTNAME uniquely identifies the synonym.
CREATOR	CHAR(8) NOT NULL	The owner of the table or view for which user USERID defined a synonym.
TNAME	VARCHAR(18) NOT NULL	CREATOR.TNAME is the real name of the table or view for which user USERID defined a synonym.

Note: SYSSYNONYMS helps resolve unqualified table references in SQL statements. If a user does not qualify the name of the table (by preceding it with "creator."), the preprocessor first looks to see if the user has a table by that name. If the user does not, the name is assumed to be a synonym for another user's table, and the preprocessor consults SYSSYNONYMS to determine the real table. The object of the synonym must be a table or a view; it cannot be another synonym.

SYSTABAUTH

SYSTABAUTH records:

- Privileges owned by users to access tables and views. For each privilege, it also records the source of the privilege (for example, a grant from another user).
- Privileges on tables and views exercised by packages. Each such privilege appears in SYSTABAUTH as if it were *granted* to the program by the user who preprocessed the program. The database manager uses SYSTABAUTH to find and invalidate packages when the necessary privileges are revoked from the creator of a program.

The columns in SYSTABAUTH are:

Column Name	Data Type	Description and Comments
GRANTOR	CHAR(8) NOT NULL	The owner who granted the privileges. If this row records the privileges exercised by a package, then GRANTOR is the creator of the corresponding program.
GRANTEE	VARCHAR(8) NOT NULL	The owner who holds the privileges or the name of the package that exercises the privileges. If the value of GRANTEE is 'PUBLIC', the privileges are held by all users.
GRANTEETYPE	CHAR(1) NOT NULL	The possible values are: **blank** if the grantee is a user. **P** if the grantee is a package.
SCREATOR	CHAR(8) NOT NULL	The owner who created the source table or source view on which privileges have been granted.
STNAME	VARCHAR(18) NOT NULL	The name of the source table or source view on which privileges have been granted. SCREATOR.STNAME uniquely identifies the source table or view.
TCREATOR	CHAR(8) NOT NULL	The owner who created the target table or view on which the grantee possesses some privileges.
TTNAME	VARCHAR(18) NOT NULL	The name of the target table or view on which the grantee possesses some privileges. TCREATOR.TTNAME uniquely identifies the target table or view. Usually, TCREATOR.TTNAME is the same as SCREATOR.STNAME. An exception occurs when a view is defined: an entry is made in SYSTABAUTH, showing the underlying table(s) in SCREATOR.STNAME and the view in TCREATOR.TTNAME.
TIMESTAMP	CHAR(12) NOT NULL	The value of the System/390 time of day clock when the grant was made. This value is used internally when privileges are revoked; it is stored as a string of numbers and letters.
UPDATECOLS	CHAR(1) NOT NULL	The possible values are: **blank** if the grant did not involve the UPDATE privilege, or if the UPDATE privilege was granted on all of the columns. * if the UPDATE privilege was granted on some of the columns. In this case, the SYSCOLAUTH table gives the names of the columns on which the UPDATE privilege was granted.

Column Name	Data Type	Description and Comments
SELECTAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the user is allowed to select rows from this table. **G** if the user is allowed to grant this SELECT privilege. **blank** otherwise. The SELECT privilege is not automatically granted because a user might be authorized to insert into a table, but not to read it.
INSERTAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the user is allowed to insert into this object. **G** if the user is allowed to grant this INSERT privilege. **blank** otherwise.
UPDATEAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the user is allowed to update this object. **G** if the user is allowed to grant this UPDATE privilege. **blank** otherwise. The field UPDATECOLS, possibly together with several rows of the SYSCOLAUTH table, identifies the columns on which the UPDATE privilege was granted.
DELETEAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the user is allowed to delete rows. **G** if the user is allowed to grant this DELETE privilege. **blank** otherwise.
ALTERAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the object is a base table and the user is allowed to alter it. **G** if the user is allowed to grant this ALTER privilege. **blank** otherwise.
INDEXAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if the object is a table and the user is allowed to create an index on it. **G** if the user is allowed to grant this INDEX privilege. **blank** otherwise.
REFAUTH	CHAR(1)	The possible values are: **Y** if the user is allowed to form, drop, activate, or deactivate a relationship where the object table is the parent table. **G** if the user is allowed to grant this REFERENCES privilege. **NULL** if the table or view was created prior to SQL/DS Version 2 Release 2. **blank** otherwise.

Note: For information on updating columns see “Updateable Columns” on page 371.

SYSUSAGE

SYSUSAGE records dependencies of one object on another. For example, a package is dependent on the tables and indexes that it uses, or a view is dependent on the tables on which it is defined. Each entry in SYSUSAGE describes one dependent object and one base object. (The base object is the object that is depended upon.) The columns in SYSUSAGE are:

Column Name	Data Type	Description and Comments
BNAME	VARCHAR(18) NOT NULL	The name of the base object (table, view, index, or DBSPACE).
BCREATOR	CHAR(8) NOT NULL	The owner of the creator of the table or index, or the owner of the DBSPACE. BCREATOR.BNAME uniquely identifies the base object.
BTYPE	CHAR(1) NOT NULL	A code indicating what the base object is: **R** real table. **V** view. **I** index. **S** DBSPACE.
DNAME	VARCHAR(18) NOT NULL	The name of the dependent view or package that is derived from or that uses the object BNAME.
DCREATOR	CHAR(8) NOT NULL	The owner who defined the dependent view or package DNAME. DCREATOR.DNAME uniquely identifies the dependent object.
DTYPE	CHAR(1) NOT NULL	The possible values are: **V** if the dependent object is a view. **X** if the dependent object is a package. Views can depend on tables and other views; packages can depend on any object.
TIMESTAMP	CHAR(8) NOT NULL	For packages, it is the value of the System/390 time of day clock when the package was created; the value is used internally and is represented as a string of numbers and letters. For views, it is the date when the view was created, in the format MM/DD/YY.

For each view or package defined, at least one entry is normally made in SYSUSAGE.

Note: If you preprocess an application program that SELECTs an undefined table, a package will be defined, but no entry will be made in the SYSUSAGE table. In this case, the preprocessor will issue a warning.

If the view or package involves only one base object (for example, `CREATE VIEW V AS SELECT * FROM EMP`), then one entry is made, with BNAME being the name of that base object. If the view or package involves more than one base object, then an entry is made for each such base object involved. SYSUSAGE enables the database manager to find the packages and views that are affected if a given base object is dropped.

SYSUSERAUTH and SYSUSERLIST

The database manager uses SYSUSERAUTH to record system authorizations. The system authorizations are DBA, RESOURCE, SCHEDULE, and CONNECT authority. As in SYSTABAUTH, an entry in SYSUSERAUTH indicates either a system authorization held by a user or a special privilege exercised by a program.

Only users with DBA authority can access SYSUSERAUTH; other users must access the view SYSUSERLIST. The creator of the view is SQLDBA; thus, you must refer to the view as SQLDBA.SYSUSERLIST. The SYSUSERLIST view contains all columns of SYSUSERAUTH except PASSWORD. The columns in SYSUSERAUTH (and SYSUSERLIST) are:

Column Name	Data Type	Description and Comments
NAME	CHAR(8) NOT NULL	Either the user ID of a user, or the name of a program. The two possibilities are distinguished by the contents of the AUTHOR field: if AUTHOR is blank, this field contains a user ID; if not, it contains the name of a program, and AUTHOR contains the user ID of the creator of the program.
AUTHOR	CHAR(8) NOT NULL	This is blank (ignored) if NAME is the name of a DB2 Server for VSE & VM user; if NAME is the name of a program, then this field contains the user ID of the person who preprocessed the program.
RESOURCEAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if this user is authorized to create new tables and authorized to acquire a private dbspace by issuing the ACQUIRE DBSPACE statement. **blank** otherwise.
DBAAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if this user has DBA authority. **blank** otherwise. A user with DBA authority is entitled to see everything in the database, including the catalog tables, and may issue any SQL statement. A user with DBA authority may also acquire and drop PUBLIC DBSPACEs. Changes to the SYSUSERAUTH table may be made only by SQL statements issued by a user with DBA authority (there is at least one DBA at catalog generation time).
PASSWORD	CHAR(8) NOT NULL	This verifies the identity of a user on a CONNECT statement to a DB2 Server for VSE & VM system. It is updated using a GRANT CONNECT or GRANT DBA statement.
SCHEDULEAUTH	CHAR(1) NOT NULL	The possible values are: **Y** if this user is authorized to CONNECT another user without specifying a password. (Used for CICS support by the DB2 Server for VSE database manager.) **blank** otherwise.

SYSVIEWS

The SYSVIEWS table contains the definitions of all views. The views are stored in the form of the original SQL statements that defined the views. The columns in SYSVIEWS are:

Column Name	Data Type	Description and Comments
VIEWNAME	VARCHAR(18) NOT NULL	The name of the view.
VCREATOR	CHAR(8) NOT NULL	The owner who defined the view. VCREATOR.VIEWNAME uniquely identifies the view.
SEQNO	SMALLINT NOT NULL	Because a view definition may consist of more than 254 characters, it may have to be divided among several rows of SYSVIEWS. The row that contains the first portion of a view definition has SEQNO = 1; successive rows have increasing values of SEQNO. You can use SEQNO to order the view definitions properly when you query this table.
VIEWTEXT	VARCHAR(254) NOT NULL	This contains the SQL statement that defined the view.
VIEWMAT	CHAR(1)	Indicates whether this view references another view resulting in view materialization. Possible values are: **Y** if a view materialization is involved. **N** if a view materialization is not involved. **NULL** if the view was created on an SQL/DS database prior to SQL/DS Version 3 Release 1 and subsequently migrated to a later version of the database manager.
VIEWCHECK	CHAR(1)	Indicates whether the view was created with the WITH CHECK OPTION clause. Possible values are: **Y** if a view was created with the clause **N** if a view was created without the clause **NULL** if a view was created prior to SQL/DS Version 3 Release 2.

Appendix D. Sample Tables

The sample tables illustrated in this appendix are used in examples throughout the DB2 Server for VSE & VM library. These tables simulate a database created for use in organization or project management applications. As a group, the tables include information that describes employees, departments, projects, and activities. This appendix contains the following sample tables:

Relationships Among the Tables

Figure 9 on page 410 shows the relationships among many of the tables. These relationships are established by referential constraints, where a foreign key in the dependent table references a primary key in the parent table. In the figure, the referential constraint is symbolized by lines joining the keys; the arrowheads point from the primary key to the foreign key. Only those columns named as foreign or primary keys are listed in the figure. All tables have additional columns. You can easily review the contents of any table by executing an SQL statement, such as `SELECT * FROM SQLDBA.DEPARTMENT`.

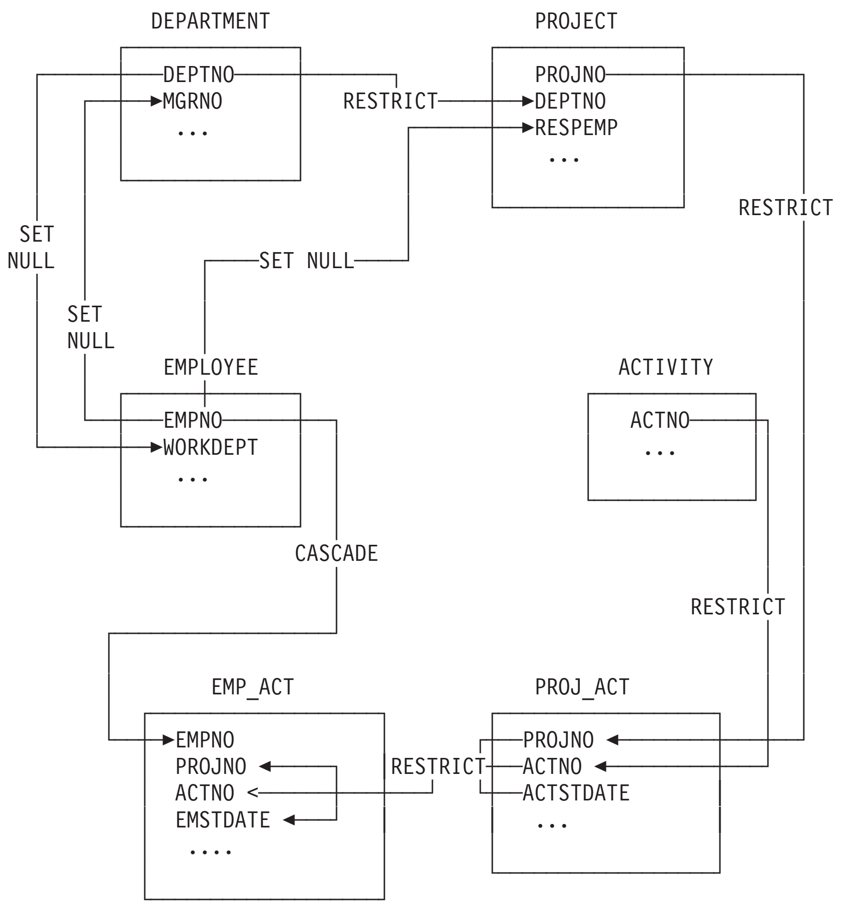


Figure 9. Relationships among Tables in the Sample Application

ACTIVITY Table

The ACTIVITY tables describes the activities that can be performed during a project. The table acts as a master list of possible activities, identifying the activity number, and providing a description of the activity.

Name:	**ACTNO**	**ACTKWD**	**ACTDESC**
Type:	smallint not null	char(6) not null	varchar(20) not null
Desc:	Activity number	Activity keyword	Activity description
Values:	10	MANAGE	Manage/advise
	20	ECOST	Estimate cost
	30	DEFINE	Define specs
	40	LEADPR	Lead program/design
	50	SPECS	Write specs
	60	LOGIC	Describe logic
	70	CODE	Code programs
	80	TEST	Test programs
	90	ADMQS	Adm query system
	100	TEACH	Teach classes
	110	COURSE	Develop courses

Name:	ACTNO	ACTKWD	ACTDESC
	120	STAFF	Pers and staffing
	130	OPERAT	Oper computer sys
	140	MAINT	Maint software sys
	150	ADMSYS	Adm operating sys
	160	ADMDB	Adm databases
	170	ADMDC	Adm data comm
	180	DOC	Document

Relationship of ACTIVITY to Other Tables

ACTIVITY is a parent of the PROJ_ACT table.

CL_SCHED Table

The CL_SCHED table describes a classroom schedule.

Name:	CLASS_CODE	DAY	STARTING	ENDING
Type:	char(7) not null	smallint not null	time not null	time not null
Desc:	Class code (room:teacher)	Day number of 4 day schedule	Class start time	Class end time
Values:	101:KAR	2	14.10.00	16.10.00
	202:LMM	3	14.40.00	16.40.00
	303:RAR	4	09.00.00	09.40.00

DEPARTMENT Table

The DEPARTMENT table describes each department in the business and identifies its manager and the department to which it reports.

Name:	DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
Type:	char(3) not null	varchar(29) not null	char(6)	char(3) not null
Desc:	Department number	Name describing general activities of department	Employee number (EMPNO) of department manager	Department (DEPTNO) that this department reports to
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

Relationship of DEPARTMENT to Other Tables

DEPARTMENT is a parent of the EMPLOYEE and PROJECT tables.

The DEPARTMENT table is a dependent of the EMPLOYEE table; the MGRNO column is the foreign key in the DEPARTMENT table and references EMPNO, the primary key in the EMPLOYEE table.

EMPLOYEE Table

The EMPLOYEE table identifies all employees by an employee number and lists basic personnel information.

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 lists the values in the EMPLOYEE table:

EMPNO	FIRSTNAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	EDUC 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

EMPNO	FIRSTNAME	MID INIT	LASTNAME	WORK DEPT	PHONE NO	HIREDATE	JOB	EDUC LEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
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

Relationship of EMPLOYEE to Other Tables

The EMPLOYEE table is a parent of the DEPARTMENT table, the PROJECT table, and the EMP_ACT table.

The EMPLOYEE table is a dependent of the DEPARTMENT table; the foreign key on the WORKDEPT column in the EMPLOYEE table references the primary key on the DEPTNO column in the DEPARTMENT table.

EMP_ACT Table

The EMP_ACT table identifies the employee performing each activity listed for each project.

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 full time to be spent on one 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

EMP_ACT Table

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	000250	AD3112	180	.50	1982-08-15	1983-01-01
	000260	AD3113	70	.50	1982-06-15	1982-07-01
	000260	AD3113	70	1.00	1982-07-01	1983-02-01
	000260	AD3113	80	1.00	1982-01-01	1982-03-01
	000260	AD3113	80	.50	1982-03-01	1982-04-15
	000260	AD3113	180	.50	1982-03-01	1982-04-15
	000260	AD3113	180	1.00	1982-04-15	1982-06-01
	000260	AD3113	180	.50	1982-06-01	1982-07-01
	000270	AD3113	60	.50	1982-03-01	1982-04-01
	000270	AD3113	60	1.00	1982-04-01	1982-09-01
	000270	AD3113	60	.25	1982-09-01	1982-10-15
	000270	AD3113	70	.75	1982-09-01	1982-10-15
	000270	AD3113	70	1.00	1982-10-15	1983-02-01
	000270	AD3113	80	1.00	1982-01-01	1982-03-01
	000270	AD3113	80	.50	1982-03-01	1982-04-01
	000030	IF1000	10	.50	1982-06-01	1983-01-01
	000130	IF1000	90	1.00	1982-01-01	1982-10-01
	000130	IF1000	100	.50	1982-10-01	1983-01-01
	000140	IF1000	90	.50	1982-10-01	1983-01-01
	000030	IF2000	10	.50	1982-01-01	1983-01-01
	000140	IF2000	100	1.00	1982-01-01	1982-03-01
	000140	IF2000	100	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-03-01	1982-07-01
	000140	IF2000	110	.50	1982-10-01	1983-01-01
	000010	MA2100	10	.50	1982-01-01	1982-11-01
	000110	MA2100	20	1.00	1982-01-01	1982-03-01
	000010	MA2110	10	1.00	1982-01-01	1983-02-01
	000200	MA2111	50	1.00	1982-01-01	1982-06-15
	000200	MA2111	60	1.00	1982-06-15	1983-02-01
	000220	MA2111	40	1.00	1982-01-01	1983-02-01
	000150	MA2112	60	1.00	1982-01-01	1982-07-15
	000150	MA2112	180	1.00	1982-07-15	1983-02-01
	000170	MA2112	60	1.00	1982-01-01	1983-06-01
	000170	MA2112	70	1.00	1982-06-01	1983-02-01
	000190	MA2112	70	1.00	1982-02-01	1982-10-01
	000190	MA2112	80	1.00	1982-10-01	1983-10-01
	000160	MA2113	60	1.00	1982-07-15	1983-02-01
	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

Name:	EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
	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

Relationship of EMP_ACT to Other Tables

The EMP_ACT table is a dependent of:

- The EMPLOYEE table; the foreign key on EMPNO in the EMP_ACT table references the primary key, EMPNO, in the EMPLOYEE table.
- The PROJ_ACT table; the foreign key on the set of PROJNO, ACTNO, EMSTDATE in the EMP_ACT table references the primary key, PROJNO, ACTNO, ACSTDATE, in the PROJ_ACT table.

IN_TRAY Table

The IN_TRAY table contains a person's note log.

Name:	RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
Type:	timestamp not null	char(8) not null	char(64)	varchar(4000)
Desc:	Date and time note was received	User id of person who sent note	Brief description	The text of the note
Values:	1965-01-01-07.00.00	SQLDBA	English	Here is a note from your DBA.

PROJECT Table

The PROJECT table describes each project that the business is currently undertaking. Data contained in each row includes the project number, name, person responsible, and schedule dates as shown below.

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	?

Name:	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
	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

Relationship of PROJECT to Other Tables

PROJECT is a parent of the PROJ_ACT table.

PROJECT is a dependent of:

- The DEPARTMENT table; the foreign key on the DEPTNO column in PROJECT references the primary key in the DEPARTMENT table.
- The EMPLOYEE table; the foreign key on the RESPEMP column in PROJECT references the primary key in the EMPLOYEE table.

PROJ_ACT Table

The PROJ_ACT table lists the activities performed for each project. The table contains information on the start and completion dates of the project activity as well as staffing requirements as shown below.

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
Type:	char(6) not null	smallint not null	decimal(5,2)	date not null	date
Desc:	Project number	Activity number	Estimated mean staffing for activity	Estimated start date for activity	Estimated end date for activity
Values:	AD3100	10	0.50	1982-01-01	1982-07-01
	AD3110	10	1.00	1982-01-01	1983-01-01
	AD3111	60	0.50	1982-03-15	1982-04-15

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	AD3111	60	0.80	1982-01-01	1982-04-15
	AD3111	70	0.50	1982-03-15	1982-10-15
	AD3111	70	1.50	1982-02-15	1982-10-15
	AD3111	80	1.00	1982-09-15	1983-01-01
	AD3111	80	1.25	1982-04-15	1983-01-15
	AD3111	180	1.00	1982-10-15	1983-01-15
	AD3112	60	0.50	1982-02-01	1982-03-15
	AD3112	60	0.75	1982-01-01	1982-05-15
	AD3112	60	0.75	1982-12-01	1983-01-01
	AD3112	60	1.00	1983-01-01	1983-02-01
	AD3112	70	0.25	1982-08-15	1982-10-15
	AD3112	70	0.50	1982-02-01	1982-03-15
	AD3112	70	0.75	1982-01-01	1982-10-15
	AD3112	70	1.00	1982-03-15	1982-08-15
	AD3112	80	0.35	1982-08-15	1982-12-01
	AD3112	80	0.50	1982-10-15	1982-12-01
	AD3112	180	0.50	1982-08-15	1983-01-01
	AD3113	60	0.25	1982-09-01	1982-10-15
	AD3113	60	0.75	1982-03-01	1982-10-15
	AD3113	60	1.00	1982-04-01	1982-09-01
	AD3113	70	0.50	1982-06-15	1982-07-01
	AD3113	70	0.75	1982-09-01	1982-10-15
	AD3113	70	1.00	1982-07-01	1983-02-01
	AD3113	70	1.00	1982-10-15	1983-02-01
	AD3113	70	1.25	1982-06-01	1982-12-15
	AD3113	80	0.50	1982-03-01	1982-04-15
	AD3113	80	1.75	1982-01-01	1982-04-15
	AD3113	180	0.50	1982-06-01	1982-07-01
	AD3113	180	0.75	1982-03-01	1982-07-01
	AD3113	180	1.00	1982-04-15	1982-06-01
	IF1000	10	0.50	1982-01-01	1983-01-01
	IF1000	10	0.50	1982-06-01	1983-01-01
	IF1000	90	0.50	1982-10-01	1983-01-01
	IF1000	90	1.00	1982-01-01	1983-01-01
	IF1000	100	0.50	1982-01-01	1983-01-01
	IF2000	10	0.50	1982-01-01	1983-01-01
	IF2000	100	0.50	1982-03-01	1982-07-01
	IF2000	100	0.75	1982-01-01	1982-07-01
	IF2000	110	0.50	1982-03-01	1982-07-01
	IF2000	110	0.50	1982-10-01	1983-01-01
	MA2100	10	0.50	1982-01-01	1982-11-01

PROJ_ACT Table

Name:	PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
	MA2100	20	1.00	1982-01-01	1982-03-01
	MA2110	10	1.00	1982-01-01	1983-02-01
	MA2111	40	1.00	1982-01-01	1983-02-01
	MA2111	50	1.00	1982-01-01	1092-06-01
	MA2111	60	1.00	1982-06-01	1983-02-01
	MA2111	60	1.00	1982-06-15	1983-02-01
	MA2112	60	2.00	1982-01-01	1982-07-01
	MA2112	70	1.00	1982-02-01	1982-10-01
	MA2112	70	1.00	1982-06-01	1983-02-01
	MA2112	70	1.50	1982-02-15	1983-02-01
	MA2112	80	1.00	1982-10-01	1983-10-01
	MA2112	180	1.00	1982-07-01	1983-02-01
	MA2112	180	1.00	1982-07-15	1983-02-01
	MA2113	60	1.00	1982-02-15	1982-09-01
	MA2113	60	1.00	1982-07-15	1983-02-01
	MA2113	70	2.00	1982-04-01	1983-12-15
	MA2113	80	1.00	1982-01-01	1983-02-01
	MA2113	80	1.50	1982-09-01	1983-02-01
	MA2113	80	0.50	1982-10-01	1983-02-01
	MA2113	180	0.50	1982-10-01	1983-01-01
	OP1000	10	0.25	1982-01-01	1983-02-01
	OP1010	10	1.00	1982-01-01	1983-02-01
	OP1010	130	4.00	1982-01-01	1983-02-01
	OP2000	50	0.75	1982-01-01	1983-02-01
	OP2010	10	1.00	1982-01-01	1983-02-01
	OP2011	140	0.75	1982-01-01	1983-02-01
	OP2011	150	0.25	1982-01-01	1983-02-01
	OP2012	140	0.25	1982-01-01	1983-02-01
	OP2012	160	0.75	1982-01-01	1983-02-01
	OP2013	140	0.50	1982-01-01	1983-02-01
	OP2013	170	0.50	1982-01-01	1983-02-01
	PL2100	30	1.00	1982-01-01	1982-09-15
	PL2100	30	1.00	1982-02-01	1982-09-01

Relationship of PROJ_ACT to Other Tables

PROJ_ACT is a parent of the EMP_ACT table.

It is a dependent of:

- The ACTIVITY table; the foreign key on ACTNO in the PROJ_ACT table references the primary key, ACTNO, in the ACTIVITY table.
- The PROJECT table; the foreign key on PROJNO in the PROJ_ACT table references the primary key, PROJNO, in the PROJECT table.

Appendix E. Data Conversion Chart

Source Data Type	Target Data Type						
	CHAR	DATE	DECIMAL	FLOAT-DOUBLE	FLOAT-SINGLE	GRAPHIC	INTEGER
CHAR	YES[3]	YES[6]	NO	NO	NO	NO	NO
DATE	YES[7]	YES	NO	NO	NO	NO	NO
DECIMAL	NO	NO	YES[1,4]	YES[13]	YES[12,13]	NO	YES[1,2]
FLOAT-DOUBLE	NO	NO	YES[1,4,5]	YES	YES[11]	NO	YES[1,2]
FLOAT-SINGLE	NO	NO	YES[1,4,5]	YES[10]	YES	NO	YES[1,2]
GRAPHIC	NO	NO	NO	NO	NO	YES[3]	NO
INTEGER	NO	NO	YES[1]	YES	YES[12]	NO	YES
LONG VARCHAR	YES[3]	NO	NO	NO	NO	NO	NO
LONG VARGRAPHIC	NO	NO	NO	NO	NO	YES[3]	NO
SMALLINT	NO	NO	YES[1]	YES	YES[12]	NO	YES
TIME	YES[7]	NO	NO	NO	NO	NO	NO
TIMESTAMP	YES[7]	NO	NO	NO	NO	NO	NO
VARCHAR[8]	YES[3]	YES[6]	NO	NO	NO	NO	NO
VARGRAPHIC[9]	NO	NO	NO	NO	NO	YES[3]	NO

Figure 10. Data Conversion Chart (Part 1 of 2)

Source Data Type	Target Data Type						
	LONG VARCHAR	LONG VAR-GRAPHIC	SMALL-INT	TIME	TIME-STAMP	VAR-CHAR[8]	VAR-GRAPHIC[9]
CHAR	YES	NO	NO	YES[6]	YES[6]	YES[3]	NO
DATE	NO	NO	NO	NO	NO	YES	NO
DECIMAL	NO	NO	YES[1,2]	NO	NO	NO	NO
FLOAT-DOUBLE	NO	NO	YES[1,2]	NO	NO	NO	NO
FLOAT-SINGLE	NO	NO	YES[1,2]	NO	NO	NO	NO
GRAPHIC	NO	YES	NO	NO	NO	NO	YES[3]
INTEGER	NO	NO	YES[1]	NO	NO	NO	NO
LONG VARCHAR	YES	NO	NO	NO	NO	YES[3]	NO
LONG VARGRAPHIC	NO	YES	NO	NO	NO	NO	YES[3]
SMALLINT	NO	NO	YES	NO	NO	NO	NO
TIME	NO	NO	NO	YES	NO	YES[7]	NO
TIMESTAMP	NO	NO	NO	NO	YES	YES[7]	NO
VARCHAR[8]	YES	NO	NO	YES[6]	YES[6]	YES[3]	NO
VARGRAPHIC[9]	NO	YES	NO	NO	NO	NO	YES[3]

Figure 10. Data Conversion Chart (Part 2 of 2)

Notes to Figure 10:

1. An overflow error may result.
2. The fractional part of the value is dropped.
3. On output, if the length of the target is smaller than the length of the source, truncation occurs. On input, an error occurs.
4. The database manager automatically aligns the decimal point. Overflow of the integer part may result. The fractional part may be truncated.
5. The database manager attempts to create the best possible result in converting from System/390 floating point to scaled fixed point decimal.
6. The character string must contain a valid representation of a date, time, or timestamp value. However, you cannot transfer data from a CHAR or VARCHAR column into a host variable defined as a date, time, or timestamp type.
7. On output, when the source is a datetime data type and the corresponding target is a character data type, certain truncation occurs for time and timestamp. On input, an error occurs.
8. This applies to VARCHAR fields less than or equal to 254. VARCHAR fields greater than 254 are treated like LONG VARCHAR in data conversion.
9. This applies to VARGRAPHIC fields less than or equal to 127. VARGRAPHIC fields greater than 127 are treated like LONG VARGRAPHIC in data conversion.
10. The single-precision data is padded with eight hex zeros.
11. The double-precision data is converted and rounded up on the seventh hex digit.
12. Conversion is first done in double precision and then rounded to single precision.

13. Some accuracy may be lost when converting DECIMAL data type numbers to single- or double-precision floating point numbers.

Appendix F. Terminology Differences

Some terminology used in this manual may be different from the terminology used in other SQL products. The terminology used in the DB2 Server for VSE & VM manuals may change in their future.

Terminology Cross-Reference

The following tables cross-reference ISO-ANS SQL(89) terms to DB2 Server for VSE & VM terms.

Table 29. ISO-ANS SQL(89) Term to DB2 Server for VSE & VM Term Cross-Reference

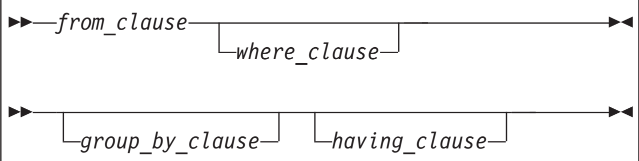
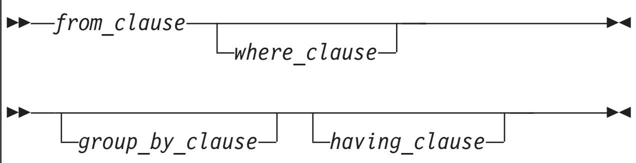
ISO-ANS SQL(89) Term	DB2 Server for VSE & VM Term
comparison predicate	basic predicate
comparison predicate subquery	subquery in a basic predicate
degree of table/cursor	number of items in a select list
grouped table	result table created by a group-by or having clause
grouped view	result view created by a group-by or having clause
grouping column	column in a group-by clause
outer reference	correlated reference
query expression	fullselect
query specification	subselect
query term	subselect or fullselect in parentheses
result specification	result
set function	column function
sort specification	order-by clause specification
table expression	
target specification	host variable followed by an indicator variable
transaction	logical unit of work or unit of work
value expression	arithmetic expression

Table 30. DB2 Server for VSE & VM Term to ISO-ANS SQL(89) Term Cross-Reference

DB2 Server for VSE & VM Term	ISO-ANS SQL(89) Term
arithmetic expression	value expression
basic predicate	comparison predicate
column function	set function
column in a group-by clause	grouping column
correlated reference	outer reference

Table 30. DB2 Server for VSE & VM Term to ISO-ANS SQL(89) Term Cross-Reference (continued)

DB2 Server for VSE & VM Term	**ISO-ANS SQL(89) Term**
 from_clause where_clause group_by_clause having_clause	table expression
fullselect	query expression
host variable followed by an indicator variable	target specification
logical unit of work or unit of work	transaction
number of items in a select list	degree of table/cursor
order-by clause specification	sort specification
result	result specification
result table created by a group-by or having clause	grouped table
result view created by a group-by or having clause	grouped view
subquery in a basic predicate	comparison predicate subquery
subselect	query specification
subselect or fullselect in parentheses	query term

Appendix G. DRDA Considerations

Users who are planning to design applications that:

- run on non-VM platforms and use the Distributed Relation Database Architecture (DRDA) protocol to connect to DB2 Server for VSE & VM servers, or
- run on VM/ESA and use the Distributed Relation Database Architecture (DRDA) protocol to connect to servers other than DB2 Server for VSE & VM

need to be aware that DB2 Server for VSE & VM's support of SQL does not exactly match the IBM SQL standard[4] or the SQL Entry Level standard.[5] This appendix attempts to provide some guidance in discrepancies to these standards.

Omissions from the Standards

For a list of where DB2 Server for VSE & VM does not support the IBM SQL or SQL92 entry level standard, please consult the *DB2 Server for VSE & VM SQL Reference* manual.

Extensions to the Standards

1. Packages that were created in SQLDS protocol by using extended dynamic statements [6] cannot be processed in DRDA protocol, or the other way around.
2. There is no support for modifiable packages created by using extended dynamic statements. If you request such support by specifying the MODIFY option on the CREATE PACKAGE statement, the system will override this option with NOMODIFY.
3. Nonmodifiable packages created by using extended dynamic statements are supported with the following restrictions:
   a. There is no support for the positioned UPDATE and positioned DELETE statements.
   b. If the Basic Extended PREPARE form of the extended PREPARE statement prepares a statement that contains parameter markers, the USING DESCRIPTOR clause must be used to identify an input SQLDA structure.
   c. There is no support for the Single Row Extended PREPARE form of the extended PREPARE statement.
   d. There is no support for the NODESCRIBE option of the CREATE PACKAGE statement. If specified, it will be ignored.
   e. There is no support for "USER" in the ISOLATION option of the CREATE PACKAGE statement. The system will override USER with CS.
   f. There is no support for "LOCAL" in the DATE or TIME option of the CREATE PACKAGE statement. If specified, SQLCODE -168 (SQLSTATE 42615) will be generated, indicating an incorrect parameter.

4. IBM SQL is a superset of the SQL99 Entry Level standard

5. Entry Level of the International Organization for Standardization (ISO) 9075-1992 Database Language SQL specification

6. Since DB2 RXSQL uses extended dynamic statements, any restrictions on the use of extended dynamics apply to DB2 RXSQL as well.

   g. DB2 Server for VSE & VM servers do not support cursors declared with the “WITH HOLD” clause. However, applications may use the “WITH HOLD” clause against other DRDA servers if they support it, except when extended dynamic statements are involved.
4. There is no support for the semantics checking of the Flagger, but the syntax checking of static SQL against the SAA and SQL-89 standards will still be carried out.

DB2 Server for VSE & VM Facility Restrictions

1. There is no support for the USERID option of the SQLPREP EXEC.
2. There is no support for “USER” in the preprocessing parameter ISOLATION. The system will override USER with CS.
3. There is no support for “LOCAL” in the preprocessor parameters DATE and TIME. If specified, SQLCODE -168 (SQLSTATE 42615) will be generated, indicating an incorrect parameter.
4. There is no support for the blocking of PUTs. However, the PUT operation will still be supported one row at a time as unblocked inserts.
5. The following ISQL commands are not supported when using the DRDA protocol, because they request functions specific to DB2 Server for VM:
   - SET ISOLATION
   - COUNTER
   - SHOW
6. The following DBSU commands are not supported when using the DRDA protocol, because they request functions specific to DB2 Server for VM:
   - UNLOAD DBSPACE
   - UNLOAD TABLE
   - UNLOAD PACKAGE
   - RELOAD DBSPACE
   - RELOAD TABLE
   - SET ISOLATION
   - SET UPDATE STATISTICS
   - REBIND PACKAGE
   - REORGANIZE INDEX
7. Fortran packages and any other packages created by using extended dynamic statements that were created in SQLDS protocol cannot be RELOADed by the DBS Utility in DRDA protocol, or the other way around.
8. Portable packages created under SQL/DS Version 2 Release 2 cannot be RELOADed by the DBS Utility in DRDA protocol.
9. If accounting data is sent from a DRDA application requester to a DB2 for VSE & VM server, only the first 16 bytes of user-defined data [7] is captured by the server and put into accounting records.

7. For example, from DDCS for OS/2 user-defined data can be set by the DFT_ACCOUNT_STR configuration parameter.

Appendix H. Incompatibilities Between Releases

This appendix identifies the incompatibilities that exist between each release of the product and the previous release, going back to Version 1 Release 3.5. There is a separate section in the appendix for each release.

Note on Skipping Releases: If your migration plans call for skipping one or more releases (for example, migrating directly from V2R2 to V3R4), you will still be affected by the incompatibilities introduced by the releases that you are skipping.

Within each section, the incompatibility items are grouped into the following categories:
- SQL and Data
- Application Programming
- System Environment

Definition of an Incompatibility

For the purpose of this appendix, an "incompatibility" is defined to be a part of the product that works differently than it did in the previous release, in such a way that if used in an existing application, it will produce a different result, necessitate a change to the application, or reduce performance. In this definition, "application" can apply to a broad range of things (singly or in combination), such as:
- Application program code
- Specifications for preprocessing application programs
- Interactive SQL queries
- ISQL functions
- DBS Utility functions
- Miscellaneous tools in your operating environment.

This appendix does not describe incompatibilities where certain operations in the current release are less likely to generate an error condition than they did in the previous release, as those changes will only have a positive impact on your applications. (For example, the SUM and AVG column functions no longer overflow as easily because they now use a larger accumulator, and a change to the use of the equal (=) compare predicate with a negative indicator variable now evaluates to UNKNOWN rather than generating an error condition.)

Impact on Existing Applications

Read the appropriate section of this appendix carefully to determine what changes you will need to make to your applications when migrating from one release to the next. You may also want to review the chapter in the manual on migration considerations which discusses some of these incompatibilities in more detail, plus other considerations for each release-to-release migration.

This appendix excludes the numerous changes and enhancements for which no impact on existing applications is anticipated. These are listed in the *Summary of Changes* section (included with each manual) of the appropriate release of the library. Review that section to see where you could make changes to your existing applications in order to take advantage of some of these enhancements.

V2R1 and V1R3.5 Incompatibilities

SQL and Data

1. Evaluation of HAVING and SELECT Clauses

   Prior to V2R1, the HAVING clause was evaluated ***after*** the SELECT clause. This caused a statement such as the following to fail on a zero divide and generate SQLCODE -802, if a zero part number was encountered:

   ```
   SELECT 200/PARTNO FROM T1
   GROUP BY PARTNO HAVING PARTNO > 0
   ```

   In V2R1, the HAVING clause is evaluated ***before*** the SELECT clause. This means your applications now have the potential of producing different results. In the above example, if a zero part number is encountered, the query does not fail and SQLCODE -802 is not generated.

2. Null Values as a Grouping Criterion

   Prior to V2R1, if any row had a null value in one of the columns referenced in a GROUP BY clause, each such row was treated as a separate group.

   In V2R1, null values are considered identical for purposes of grouping.

   This means that your existing applications may generate fewer rows in the result table than they did in previous releases, since multiple null-value-groups are now consolidated into one group. Any derived column function values will reflect this consolidation (for example, SUM(BONUS)).

3. Negative Decimal Zero Support

   Prior to V2R1, the system recognized negative decimal zero as a valid value. However, it did not evaluate positive and negative decimal zero values as equivalent.

   In V2R2, any negative decimal zeros found in SQL statements are converted to positive decimal zeros before execution. This means that inserting, updating, or deriving negative decimal zeros, or using them in a comparison, is no longer possible. A utility called SQLZERO is provided which converts all negative decimal zeros in the database to positive decimal zeros.

   For a detailed discussion of this topic, see "Elimination of Negative Decimal Zero" in the chapter which discusses migrating from V1R3.5 in the *System Planning and Administration* manual, V2R1 or later.

4. Insertion of Invalid Decimal Values

   Prior to V2R1, it was possible to insert invalid decimal data into the database during DATALOAD by specifying string values that were invalid for DECIMAL columns. For example, X'0000' has no sign value.

   In V2R1, this is no longer allowed. Doing so will generate SQLCODE -424.

Application Programming

5. Use of ORDER BY Clause with SELECT INTO

   Prior to V2R1, the SELECT INTO statement was allowed to contain an ORDER BY clause.

   In V2R1, this is no longer allowed. Doing so will generate SQLCODE -524.

6. Scope of Prepared Statements

   Prior to V2R1, a prepared statement could sometimes, but not always, be referenced in subsequent logical units of work (LUWs).

In V2R1, this inconsistency is removed. A prepared statement may now only be referenced within the same LUW in which it was prepared.

If your applications contain code that references prepared statements across LUWs, they will have to be restructured accordingly.

7. SQLCODE Returned After a Format 2 INSERT

   Prior to V2R1, when a format 2 INSERT (known as “INSERT via subselect” in V2R2 and later releases) returned an empty answer set for insertion, SQLCODE +0 was generated.

   In V2R1, SQLCODE +100 is generated instead.

8. Preprocessor Errors Converted to Warnings

   Prior to V2R1, a certain set of conditions generated errors during preprocessing.

   In V2R1, these conditions now generate warnings, although the associated SQLCODEs are still negative (starting with V3R1, the codes are presented as positive numbers). These conditions and their corresponding SQLCODEs are shown in the table below.

SQLCODE	DESCRIPTION
-134	IMPROPER USE OF THE LONG FIELD COLUMN column.
-135	THE INPUT FOR A LONG FIELD COLUMN IN AN INSERT OR UPDATE MUST BE FROM A HOST VARIABLE OR THE KEYWORD NULL.
-150	THE VIEW CANNOT BE USED TO MODIFY DATA SINCE IT IS BASED ON MORE THAN ONE TABLE.
-151	A COLUMN OF A VIEW CANNOT BE UPDATED SINCE IT IS DERIVED FROM AN EXPRESSION.
-152	A COLUMN OF A VIEW CANNOT BE USED IN A WHERE-CLAUSE SINCE IT IS DERIVED FROM A COLUMN FUNCTION.
-154	VIEW LIMITATIONS DO NOT ALLOW THE USE OF THE FOLLOWING OPERATION: operation
-155	YOU CANNOT PERFORM A JOIN ON A VIEW CONTAINING A GROUP-BY CLAUSE OR A DISTINCT KEYWORD.
-156	RESTRICTIONS APPLY WHEN SELECTING FROM A VIEW CREATED WITH THE DISTINCT OR GROUP BY KEYWORD.
-202	COLUMN column WAS NOT FOUND IN ANY TABLE REFERENCED BY THE COMMAND.
-205	COLUMN column WAS NOT FOUND IN TABLE creator.table.
-401	INCOMPATIBLE DATA TYPES FOUND IN AN EXPRESSION OR COMPARE OPERATION.
-404	A CHARACTER STRING SPECIFIED IN AN INSERT OR UPDATE IS TOO LARGE FOR THE TARGET COLUMN.
-405	THE NUMERIC VALUE, value, IS NOT WITHIN THE RANGE OF THE DATA TYPE.
-407	AN UPDATE OR INSERT OF A NULL VALUE FOR A COLUMN DEFINED AS NOT NULL IS NOT ALLOWED.
-408	AN UPDATE OR INSERT OF A DATA VALUE IS INCOMPATIBLE WITH THE DATA TYPE OF THE ASSOCIATED TARGET COLUMN.
-414	LIKE WAS USED FOR A NUMERIC OR DATE/TIME COLUMN TYPE. IT MUST ONLY BE USED WITH CHAR OR VARCHAR TYPE COLUMNS.
-415	THE DATA TYPES OF CORRESPONDING ITEMS IN THE SELECT-CLAUSES CONNECTED BY A UNION ARE NOT IDENTICAL.

SQLCODE	DESCRIPTION
-416	YOU CANNOT SPECIFY A LONG FIELD COLUMN IN THE SELECT-CLAUSE OF A UNION.
-419	THE PRECISION OF THE NUMERATOR AND/OR THE SCALE OF THE DENOMINATOR ARE TOO LARGE FOR DECIMAL DIVISION.
-421	A HEXADECIMAL LITERAL WITH AN ODD LENGTH MAY NOT BE USED WITH A DBCS COLUMN IN A PREDICATE.

V2R2 and V2R1 Incompatibilities

SQL and Data

1. Leading and Trailing zeros in Decimal Constants

   Prior to V2R2, leading and trailing zeros of decimal constants were removed by the system when calculating their scale and precision.

   In V2R2, if the precision of a decimal constant is greater than 15, leading zeros are removed to bring the precision down to 15. Trailing zeros are not removed.

   If your current applications provide output from the result table without any intervening formatting, this change has the potential of altering that output. If formatting is involved, you may have to change the formatting logic to obtain the same output.

   Similarly, input to the database by means of INSERT or UPDATE may be affected, if a decimal constant is involved.

2. Use of Host Variables with UNION

   Prior to V2R2, two select-lists could be successfully UNION'ed even when they contained corresponding items that were host variables of different data types and different lengths. The statement below is an example of this, where host variables :hw and :fw are halfword fixed binary (15) and fullword fixed binary (31), respectively.

   ```
   SELECT :hw FROM T1
   UNION
   SELECT :fw FROM T1
   ```

   In V2R2, the above statement is no longer allowed. Issuing it will generate SQLCODE -415.

   Note: In V3R1, some restrictions on the use of data types within a UNION are removed, including the above incompatibility.

   Application Programming

3. Atomic Operations Against the Database

   Prior to V2R2, many types of operational errors (that is, SQL statement errors) against the database caused the system to roll back the entire current logical unit of work (LUW), leaving the application with no control over the status of the LUW.

   In V2R2, all operations against the database are now atomic. That is, within an LUW, each operation can succeed or fail separately, with no effect on other operations, provided they do not depend on it. If an operation fails, the application is free to either continue working on the same LUW, or commit the changes made so far, or roll back the LUW. Some system errors, such as deadlocks, still require the entire LUW to be rolled back by the system. Also, atomic operation is not supported for:

   - Operations on data located in nonrecoverable storage pools

- Operations on data when running without a log (LOGMODE=N).

As a result of this change, you may want to extend the logic of your LUW processing in your applications.

Note: The next incompatibility item contains a special case of atomic operation.

4. Multiple Row Changes Within an Atomic Operation

   Prior to V2R2, if an error occurred during a single operation involving multiple row changes to the database, the database was potentially left in an inconsistent state. (This was one of those operational errors that was not rolled back by the system.) Some of the rows were processed; the rest were not. The only practical way to avoid this inconsistency was to have the application roll back the entire current LUW.

   There was one exception to this: in the case of a data definition statement, such as CREATE TABLE, the system itself rolled back the LUW to avoid a partial definition of a table in the catalog. The application had no control over the status of the LUW.

   In V2R2, with atomic operation in place, the system automatically undoes that portion of the multiple row operation that was processed prior to the error. This eliminates the potential of an inconsistent database resulting from such an operation, and leaves the application free to control the current LUW as it sees fit.

   See "Detailed Notes on V2R2-V2R1 Incompatibilities" on page 432 for an example.

5. Four-Byte Floating-point Data

   Prior to V2R2, all floating-point data had to be eight bytes.

   In V2R2, it can be four bytes.

   This leads to a potential problem in V2R2 for programs that allocate eight bytes when using DESCRIBE on a FLOAT column. When using DESCRIBE, applications should allocate storage based on the SQLLEN of a column (as given in the SQLDA), not the SQLTYPE.

6. Arithmetic and Conversion Errors

   Prior to V2R2, an arithmetic or conversion error terminated processing of the statement and generated SQLCODE -802.

   In V2R2, these types of errors are tolerated when they involve a host variable that has an indicator variable. In such cases, processing of the SQL statement continues; SQLCODE +802 is generated; a -2 is placed in the indicator variable; and the associated database variable remains unchanged.

   If your application is checking for these errors, this could impact its logic. The types of errors that can now be tolerated are:
   - Fixed point overflow
   - Decimal overflow
   - Exponent overflow
   - Exponent underflow
   - Divide exception.

   For more detail, see the *Messages and Codes* manual, V2R2 or later, for SQLCODEs +802 and -802.

7. GRANT Authority for PUBLIC

   Prior to V2R2, "WITH GRANT OPTION" in a GRANT statement passed GRANT authority to the user receiving the privilege in question, even when the user was PUBLIC.

In V2R2, when “PUBLIC” and “WITH GRANT OPTION” are used together, the privilege is granted to PUBLIC, but without GRANT authority. In such cases, a warning is given to that effect.

This can impact your current authorization of views or programs, since these objects, which previously could have been grantable (for example, a value of 'G' recorded for a program in catalog table SYSPROGAUTH), will no longer be so (a value of 'Y' now in SYSPROGAUTH) if they depend on PUBLIC access to an object.

For example, if a program contains a static SELECT statement involving table T1, and the owner of the program is dependent on PUBLIC access to T1, then 'Y' is the highest authorization value attainable for that statement — and therefore for the program. This means that the owner is still able to run the program, but not to grant the RUN privilege on it to others. This, in turn, means that when this program is preprocessed under V2R2, users who previously may have had authority to run it (by virtue of receiving RUN authority from the owner) will no longer have that authority.

System Environment

8. Change to Message Numbers

   Prior to V2R2, the ARI message numbers were three digits long and were followed by an action indicator. This identification formed a header for each line of the message text, as illustrated below:

   ```
   ARI297A  RESPONSE TO ARCHIVE PROMPT
   ARI297A  IS NOT VALID.
   ```

   In V2R2, these message numbers are expanded to four digits to accommodate future expansion of the system. Message numbers existing in the earlier releases now contain a high-order zero. Also, the message header is now only used on the first line of the message. The above example becomes:

   ```
   ARI0297A  RESPONSE TO ARCHIVE PROMPT
             IS NOT VALID.
   ```

   This could impact any automated operating system facility that you may be using (for example, the VM Programmable Operator) to scan the message number and text.

Detailed Notes on V2R2-V2R1 Incompatibilities

1. Multiple Row Changes Within an Atomic Operation

   In the following example, the operations are contained in one LUW. The second operation involves multiple row changes to the database.

   ```
   DELETE FROM SUPPLIER WHERE SUPPNO = 64
   UPDATE INVENTORY SET PARTNO = PARTNO + 1
   INSERT INTO QUOTATIONS VALUES (64, 221, .25, 5, 100)
   ```

   The DELETE statement removes a supplier from the SUPPLIER table. The UPDATE statement changes the first two rows of the INVENTORY table, but fails on the third row because the operation would create a duplicate primary key value.[8]

   Prior to V2R2, the system would have left the new values in the first two rows of INVENTORY, with the rest of the table unchanged. To avoid this undesirable inconsistency, the application would have had to contain logic to recognize this error and roll back the entire LUW, thus undoing the DELETE.

8. In V3R2 this error will not occur, because the enforcement of uniqueness is done after all the rows are updated.

In V2R2, when this error occurs, the system undoes the UPDATE statement by reversing the changes made to the first two rows. Because neither the DELETE nor the INSERT depends on the success of the UPDATE (these operations are atomic), the application has the following options open to it:
- Proceed and perform the INSERT, or
- Commit the successful DELETE, or
- Roll back the LUW to undo the DELETE.

V3R1 and V2R2 Incompatibilities

SQL and Data

1. Table Designation Rules

   Prior to V3R1, the following set of ANS/ISO SQL rules for table designation in FROM clauses were not fully enforced:

   - Duplicate table or view names in a FROM clause must all have a correlation name assigned to them.
   - Correlation names in a FROM clause must be distinct from each other.
   - Correlation names in a FROM clause must be distinct from the table or view names in the same clause.

   When the application contained ambiguities, such as

   ```
   SELECT A.COL1
   FROM A B, B A
   ```

   where COL1 appeared in both table A and table B, the system accepted the statement, employing its own set of rules to resolve the ambiguity. This example represents only one type of ambiguity that could occur.

   In V3R1, the ANS/ISO rules are fully enforced. Any violations generate SQLCODE -211 (SQLSTATE 52012).
2. New Reserved Words

   Prior to V3R1, the following were not reserved words in SQL and could therefore be used as ordinary identifiers:
   - CHAR
   - CHARACTER
   - DOUBLE
   - EXECUTE
   - FIELDPROC
   - GRAPHIC
   - LONG
   - PACKAGE.

   Similarly, the following were not reserved words for the DBS Utility:
   - REORGANIZE
   - SCHEMA.

   In V3R1, these are reserved words, so an existing application that uses any words in the SQL group above as an ordinary identifier will have to be changed before it is preprocessed, or SQLCODE -105 (SQLSTATE 37501) will be generated. Similarly, the words in the DBS Utility group above can no longer be used in DBS Utility commands as ordinary identifiers.

   You can address this incompatibility by changing these ordinary identifiers to use nonreserved words, or you can retain the original names by redefining them as delimited identifiers.
3. Significance of Trailing Blanks

Prior to V3R1, trailing blanks were treated as significant in both object names and VARCHAR and VARGRAPHIC column values.

In V3R1, such trailing blanks are not considered significant.

If your applications must continue to treat trailing blanks as significant, you may have to undertake some redesign. See "Detailed Notes on V3R1-V2R2 Incompatibilities" on page 437 for further discussion and examples.

4. Timestamp at the 24th Hour

   Prior to V3R1, a timestamp value in which the hour portion was 24 and the minute, second, or microsecond portion was not zero, was accepted as valid data for insertion or updating.

   In V3R1, an attempt to insert or update a column with such a value generates SQLCODE -181 (SQLSTATE 22007). When the hour portion is 24, the other time portions must now be zero.

   If you have any of these invalid values in your tables after migrating to V3R1, they will prevent you from doing a DBS Utility unload/reload operation or an INSERT using a subselect. You will have to first correct these values to conform to the rule mentioned above.

Application Programming

5. Invalid Pointers in SQLDA and RDIIN

   Prior to V3R1, the system checked for invalid pointers in the SQLDA and RDIIN structures. This checking was extensive, often resulting in poor performance.

   In V3R1, in the interest of better performance, this checking has been eliminated. It is up to the application programmer to follow the rules on setting pointers in the SQLDA, as outlined in the chapter "Using Dynamic Statements" in the V3R1 *Application Programming* manual. Pointers in the RDIIN must not be changed by the application. If your application does not satisfy these rules, the results will be unpredictable.

6. Continuation Characters in Fortran

   Prior to V3R1, the Fortran preprocessor ignored any continuation character located in front of an EXEC SQL on the same line, provided it was not part of an IF or ELSE statement — even though such coding was incorrect.

   In V3R1, the continuation character is acknowledged and the EXEC SQL is ignored.

7. Missing Comma in COBOL Continuation Lines

   Prior to V3R1, if you left out an intended comma from a list of parameters in an SQL statement embedded in a COBOL program (as illustrated below) and did not code a continuation character in the next line, the system would assume a continuation character and misinterpret the parameter list, giving potentially wrong results.

```
SELECT *
FROM T1
WHERE COL1 IN ('AB'    <--- missing comma
               'CD',   <--- no continuation character
               'EF')
```

   In V3R1, this error is detected and reported at preprocessor time.

8. DROP PROGRAM Statement Containing Host Variables

   Prior to V3R1, the processing of a DROP PROGRAM statement that contained host variables required a specific section in the access module. (In this form of the statement, the name of the owner of the program or the name of the program or both are expressed as host variables.)

Note on New Terminology: As of V3R1, PACKAGE becomes the new reserved word for PROGRAM, the latter remaining as a synonym. Access modules are now referred to as packages. This new terminology is used below.

In V3R1, the host variable form of the DROP PACKAGE statement no longer requires a section in the package. All the information required to execute the statement is sent with the execution-time request. You will be affected if you have this form of the DROP PACKAGE coded in your application programs.

If the programs that use these packages are explicitly repreprocessed, they will have to be recompiled (or reassembled) and relinked in order to execute successfully. Otherwise, errors will result, since there will be fewer sections in the new package and this will cause a mismatch between section numbers in the RDIIN structure and the new package.

9. Data Type of String Constants

   Prior to V3R1, application programs that assumed that string constants have a data type of VARGRAPHIC because they are used in the context of GRAPHIC and VARGRAPHIC data, were accepted.

   In V3R1, such constants are considered to be VARCHAR, and if used in conjunction with GRAPHIC or VARGRAPHIC data will result in an error, such as SQLCODE -171 (SQLSTATE 53015) or SQLCODE -408 (SQLSTATE 53021).

   If the host language is COBOL, PL/I, or C, you should use explicitly coded graphic constants. See the section of the V3R1 *SQL Reference* manual that discusses graphic string constants.

10. New Options in CREATE PROGRAM Statement

    Prior to V3R1, when the following three options:

    ```
    ISOL({RR|CS|USER})
    DATE({ISO|USA|EUR|JIS|LOCAL})
    TIME({ISO|USA|EUR|JIS|LOCAL})
    ```

    were used in conjunction with an extended dynamic access module, the values for these options were determined when statements referencing the extended dynamic access module were executed. The values were set based on the corresponding preprocessing options of the program containing the extended dynamic statements.

    Note on New Terminology: As of V3R1, PACKAGE becomes the new reserved word for PROGRAM, the latter remaining as a synonym of the former. Access modules are now referred to as packages. This new terminology is used below.

    In V3R1, these options are added to the CREATE PACKAGE statement, so that they become preprocessing options. This means that their values are stored with the package itself, and are enforced when the sections of the package are executed. Consequently, your programs may now run at a different isolation level than they did in V2R2.

See "Detailed Notes on V3R1-V2R2 Incompatibilities" on page 437 for examples that illustrate how incompatibilities may arise as a result of this change.

11. Views Created from SELECT *

    Prior to V3R1, views created as SELECT * FROM T1 required no special attention when being migrated from release to release, even when columns had been added to table T1 ***after*** the creation of the view.

    In V3R1, a necessary change to the system now requires special attention in the above situation. The first time the system encounters such a view in an application, it attempts to rebuild the view, and fails with SQLCODE -835 (SQLSTATE 56049).

    To avoid this failure, drop and recreate the view before running the application on V3R1. Depending on how your application logic is coded, you may have to change that logic in order to handle the extra columns that were added to table T1. The best practice is to avoid the use of SELECT * for view creation, and specify the explicit columns that the application requires.

12. Semicolon Delimiter in SYSVIEW Table

    Prior to V3R1, when a view was created through the DBS Utility or by running a preprocessed program, the CREATE VIEW statement was inserted into column VIEWTEXT of catalog table SYSVIEWS with a semicolon delimiter.

    In V3R1, this delimiter is no longer included.

    If your application has a dependency on the existence of this delimiter in the SYSVIEWS table, you will need to change it accordingly.

13. Replacement of Error Message ARI0565E

    Prior to V3R1, error message ARI0565E was issued during preprocessing of Fortran programs whenever the input source contained no SQL statements that required creation of a package.

    In V3R1, this message is replaced by information message ARI0565I. In addition, related message, ARI0598I, dealing with the status of the package, is modified.

    This could impact any automated operating system facility that you may be using (for example, the VM Programmable Operator) to scan the message number and text.

14. Replacement of SQLCODE -150

    Prior to V3R1, an attempt to modify data through a view based on more than one table generated SQLCODE -150.

    In V3R1, this is replaced with SQLCODE +149 at preprocessor time, and SQLCODE -149 (SQLSTATE 53007) at run time.

15. New Positive SQLCODEs

    Prior to V3R1, a number of negative SQLCODEs and associated positive RDSCODEs were returned during preprocessing to indicate a warning situation.

    In V3R1, new positive SQLCODEs are returned instead, which correspond identically to the above negative SQLCODEs in code number and (in most cases) message text and explanation. If the error is not removed, the corresponding negative SQLCODEs will be issued at run time.

    See "Detailed Notes on V3R1-V2R2 Incompatibilities" on page 437 for a list of these new positive SQLCODEs.

System Environment

16. Uppercase and Mixed Case in Message Text

    Prior to V3R1, all message text was in uppercase for all the languages available in the product except German, which was available only in mixed case.

    Note: The uppercase applied to both English language offerings, AMENG and UCENG. It also applied to the English text embedded in the DBCS languages Japanese and Korean (for example, "FORCE", "SQLEND").

    In V3R1, the message text of three more languages is now changed to mixed case only. These languages are AMENG (the default language setting), Italian, and Spanish. If you are using any of these three languages and you have existing case-sensitive applications that scan for specific message text in uppercase only, you will have to modify them to detect lowercase as well. This could impact any automated operating system facility that you may be using for this purpose (for example, the VM Programmable Operator).

    An alternative approach (for English users only) to modifying your applications would be to specify UCENG instead of AMENG, through the SET LANGUAGE command.

17. Authorization for Changing System Catalog Tables

    Prior to V3R1, certain portions of the catalog could be updated, deleted, or inserted into, by any user with DBA authority.

    In V3R1, the number of columns in the catalog tables for which these changes are allowed is reduced.

    This change may affect the authorization of some of your applications. See Appendix E of the V3R1 *SQL Reference* manual for a list of the columns that can now be updated, deleted, or inserted.

18. Modification of Sample Tables and Applications

    Prior to V3R1, the sample tables shipped with the product consisted of five Manufacturing tables and four Organizational-project tables. The sample applications shipped with the product used the Manufacturing tables.

    In V3R1, the Manufacturing tables are not included, but can be installed optionally. The Organization-project tables are enhanced to provide more guidance on referential integrity and also consistency across the IBM relational database products. The enhancements include:

    - Two new tables
    - A new column in an existing table
    - Renaming of a table
    - Modification of a foreign key definition.

    The sample applications are now modified to use the enhanced Organization-project tables. They now issue a ROLLBACK instead of a COMMIT, so that they can be rerun without having to first restore the sample database.

    If you have any applications that use these tables, such as an online tutorial or a test package for new releases, you will need to upgrade them accordingly.

Detailed Notes on V3R1-V2R2 Incompatibilities

1. Significance of Trailing Blanks

   Prior to V3R1, delimited identifiers "TABLE1" and "TABLE1ƀ" would be considered two different tables, and VARCHAR values 'ABC' and 'ABCƀƀ' two different values, where 'ƀ' represents a blank character.

   In V3R1, in the case of the table names, the system would not accept the two tables because they now have identical names. In the case of the VARCHAR values, they are considered equal, except in a LIKE comparison. However, if specified at INSERT or UPDATE time, trailing blanks are included in the varying length string data stored in the database.

If your applications must continue to treat trailing blanks as significant, you may have to undertake some redesign. For example, prior to V3R1, if your table had a VARCHAR column, COLX, containing 'AAAƀƀƀ' and you wanted to select all values from COLX that were not equal to 'AAA', the following search condition would satisfy this requirement, because it would return value 'AAAƀƀƀ' along with any other values not equal to 'AAA':

```
WHERE COLX <> 'AAA'
```

In V3R1, value 'AAAƀƀƀ' does not get returned in the above example. This search condition must be redesigned in order to get the same results as in prior releases. One solution is:

```
WHERE COLX NOT LIKE 'AAA'
```

For more discussion on migration considerations for this item, see "Considerations for VARCHAR and VARGRAPHIC Compare" in the chapter which discusses migrating from V2R2, in the *System Administration* manual, V3R1 or later.

2. New Options in CREATE PROGRAM Statement

   The following examples illustrate the incompatibilities that may arise when you migrate to V3R1.

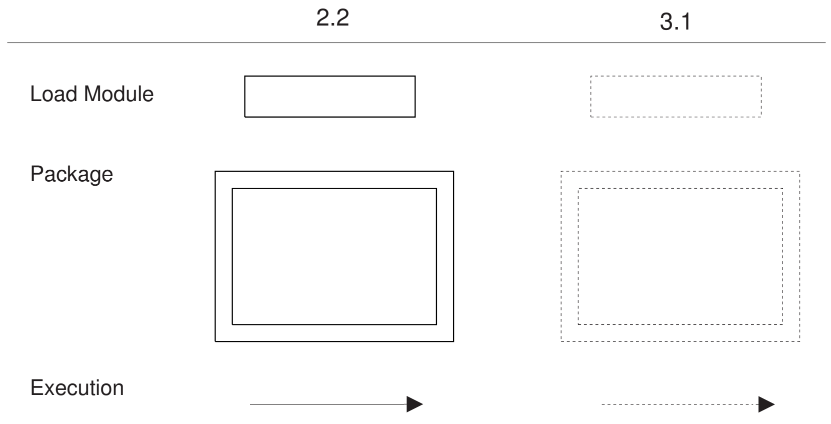


Figure 11. Legend

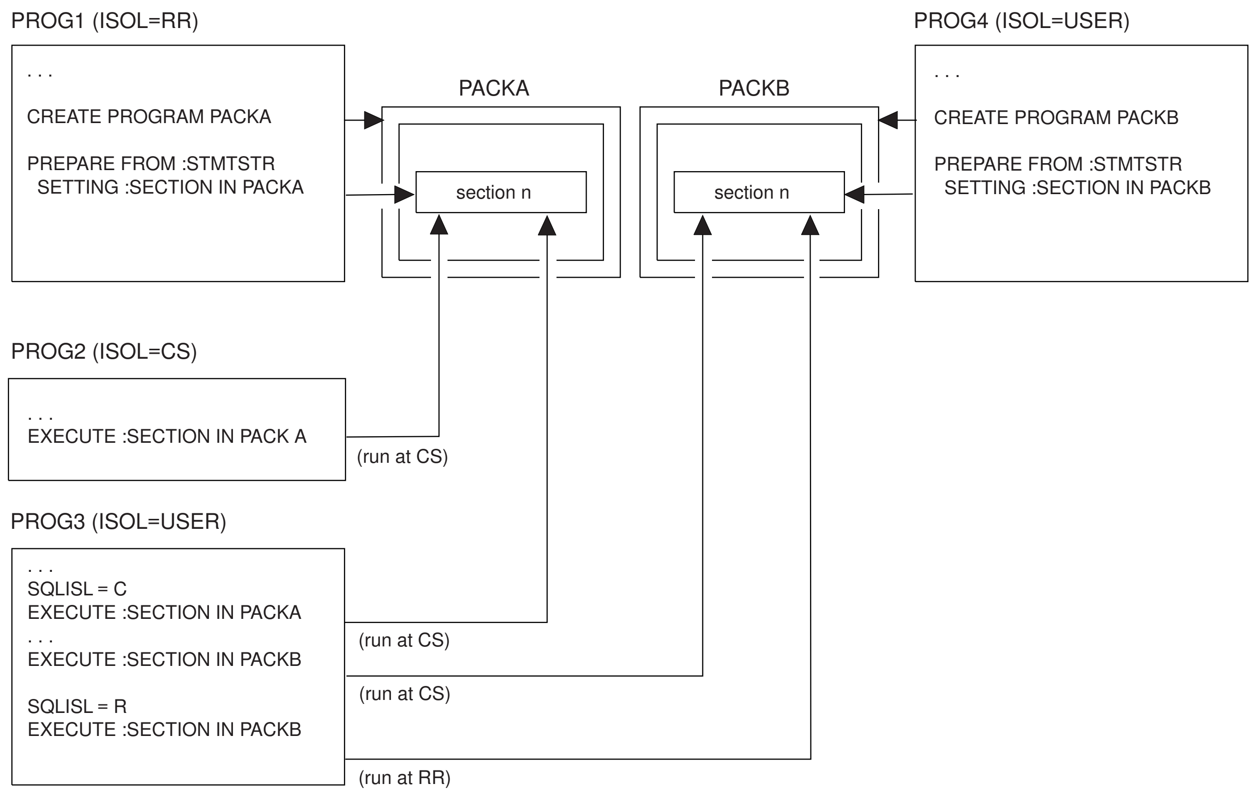


Figure 12. Version 2 Release 2

Figure 12 illustrates how isolation levels are determined for packages created using extended dynamic SQL in V2R2. For example, program PROG1 contains the CREATE PROGRAM statement for package PACKA, and prepares a section in the package. Program PROG2 subsequently executes the section in PACKA. Since program PROG2 was preprocessed with isolation level cursor stability (CS), the section executes using CS.

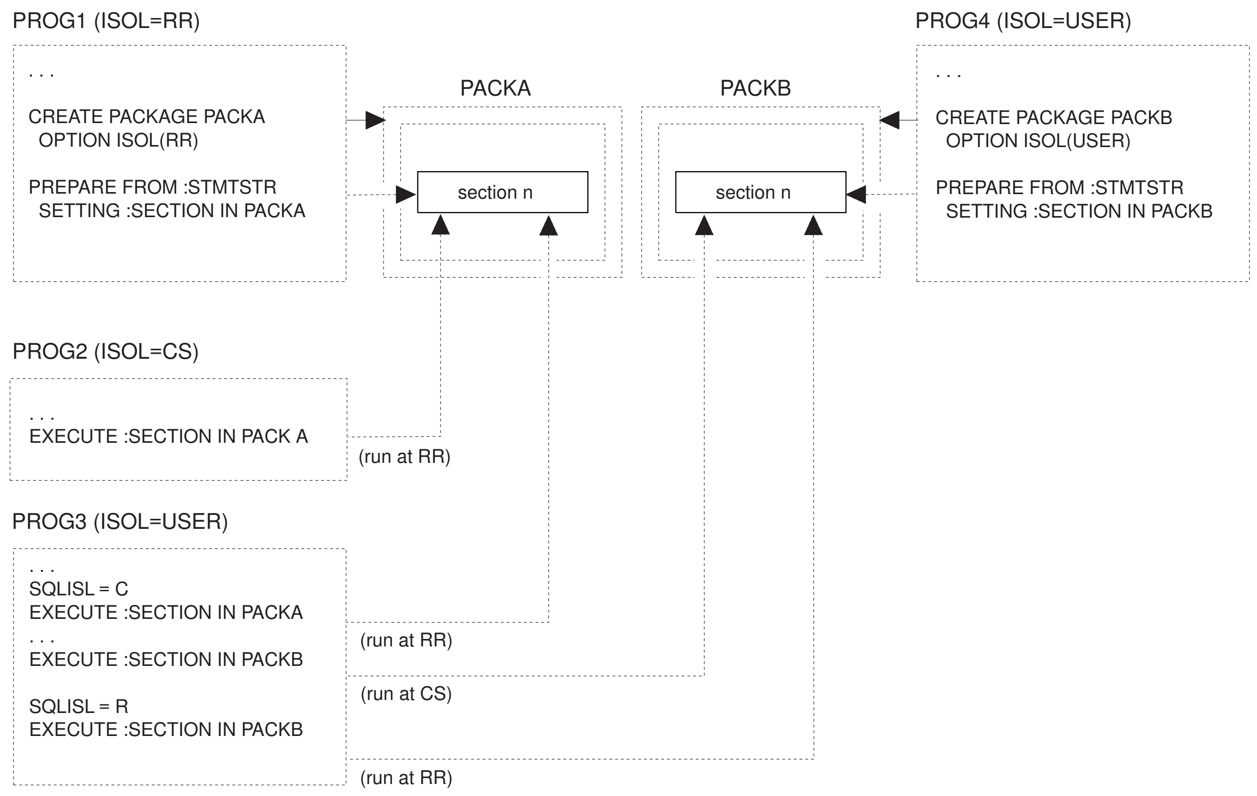


Figure 13. Version 3 Release 1

Figure 13 shows the same scenario in V3R1. In this case, the isolation level RR is specified when the PACKA package is created. When program PROG2 executes a section in PACKA, isolation level RR is used.

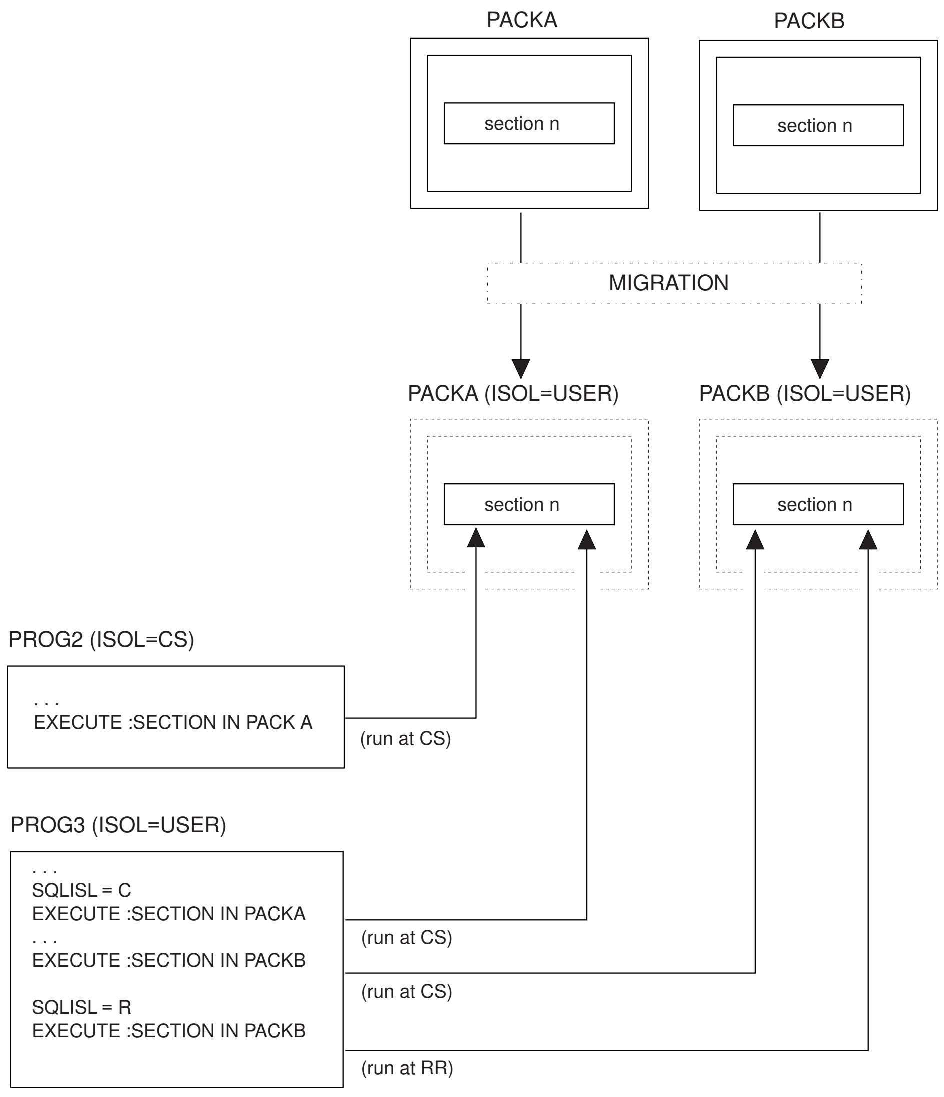


Figure 14. Migration

Figure 14 shows packages being migrated to V3R1. In this case, the isolation level bind option will be automatically set to USER. Applications will notice no change in isolation level handling from previous releases.

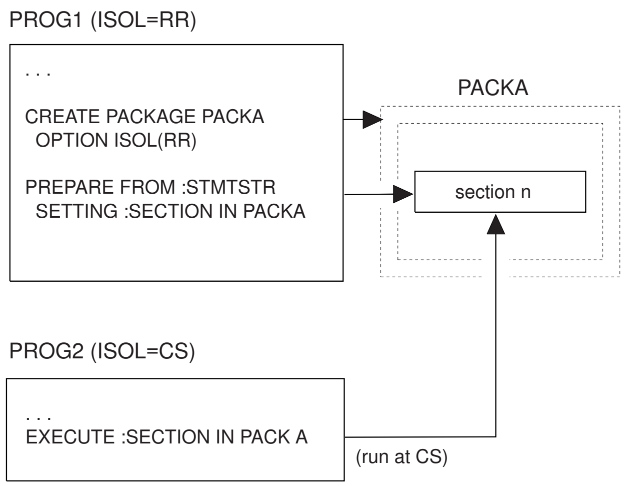


Figure 15. Dropping and Re-creating PACKA Without Repreprocessing PROG2

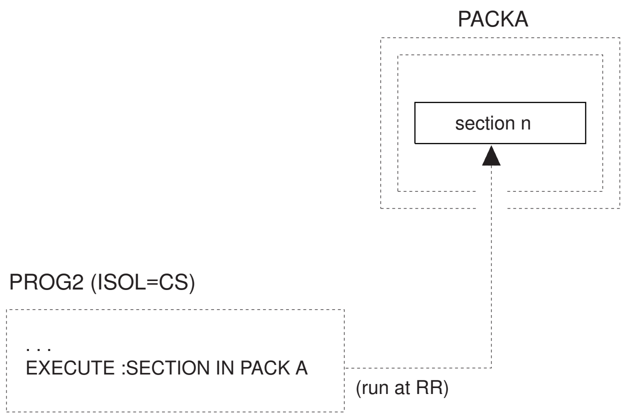


Figure 16. Re-preprocessing PROG2

Figure 15 and Figure 16 show that once an extended dynamic package has been dropped and recreated in V3R1 with an isolation level other than USER, the isolation level bind option will be enforced whenever the executing application has also been preprocessed, assembled, and re-linked under V3R1. If the PACKA package has been dropped and recreated in V3R1, with an isolation level of RR, then:

- If program PROG2 is still pre-V3R1, when the section in PACKA is executed, isolation level CS will be used.
- Otherwise, isolation level RR will be enforced whenever sections in PACKA are executed.

3. New Positive SQLCODEs

   These codes are shown in the table below.

SQLCODE	SQLSTATE	DESCRIPTION
+117	01525	The number of data values to be inserted does not equal the number of columns specified or implied.
+134		Improper use of long string.

SQLCODE	SQLSTATE	DESCRIPTION
+135		The input for a long string column in an INSERT statement or UPDATE statement must be from a host variable or be the keyword NULL.
+149		The view cannot be used to modify data because it is based on more than one table.
+151		A column of a view cannot be updated since it is derived from an expression.
+154		View limitations do not allow you to use the following operation: xxxxxx
+202	01533	Column xxxxxx was not found in any table referenced by the statement.
+204	01532	xxxxxx was not found in the system catalog.
+205	01533	Column xxxxxx was not found in table yyyyyy.
+206	01533	The xxxxxx on yyyyyy was not found.
+401		Incompatible data types found in an expression or compare operation.
+404		A character string specified in an INSERT or UPDATE statement is too large for the target column.
+405		The numeric value, xxxxxx, is not within the range of the data type.
+407		Either an UPDATE statement or an INSERT statement with a null value for a column defined as NOT NULL is not allowed, or a null host variable value is not allowed in a SELECT list.
+408		An UPDATE or INSERT of a data value is incompatible with the data type of the associated target column.
+414		The LIKE clause was used for a numeric or date/time column type. LIKE must only be used with character or graphic compatible columns.
+415		The corresponding columns, n, of the operand of a UNION or a UNION ALL do not have comparable column descriptions.
+416		You cannot specify a long string column in the SELECT clause of a UNION.
+419		The precision of the numerator and/or the scale of the denominator are too large for decimal division.
+421		A hexadecimal literal associated with a graphic compatible column in a predicate cannot have an odd length.
+551	01548	User xxxxxx does not have the yyyyyy privilege.
+552	01542	xxxxxx is not authorized to perform this statement.
+668		Table xxxxxx is inactive and you cannot access it.

V3R3 and V3R2 Incompatibilities (VM Only)

Note: This section does not include the restrictions on the use of DRDA protocol, as that topic is covered in the appendix describing DRDA considerations.

SQL and Data

1. New Reserved Word, CONCAT

   Prior to V3R3, CONCAT was not a reserved word in SQL and could therefore be used as an ordinary identifier.

   In V3R3, CONCAT is a reserved word, and can be used as an alternative to the concatenation operator (||). Any existing applications that use it as an ordinary identifier will have to be changed before they are preprocessed under V3R3; otherwise SQLCODE -105 (SQLSTATE 37501) will be generated.

   You can address this incompatibility by changing this ordinary identifier to use a nonreserved word, or you can retain the original name by redefining it as a delimited identifier.

2. REVOKE UPDATE

   Prior to V3R3, the REVOKE statement for the UPDATE privilege ignored any column names that might be present as parameters of the UPDATE option — even though such coding was invalid. (This statement is only done on a table basis, never a column basis.)

   In V3R3, such parameters are not allowed. If they are used, SQLCODE -105 (SQLSTATE 37501) will be generated.

3. Numeric Data in Character Strings

   Prior to V3R3, columns with a data type of CHAR or VARCHAR accepted numeric data, including FLOAT, on insert or update. For example, the following statements did not create an error:

   ```
   CREATE TABLE T1 (COL CHAR(8))
   CREATE TABLE T2 (COL VARCHAR(8))

   INSERT INTO T1 (123)
   INSERT INTO T2 (123)
   INSERT INTO T1 (1E1)
   INSERT INTO T2 (1E1)

   UPDATE T1 SET COL = 123
   UPDATE T2 SET COL = 123
   UPDATE T1 SET COL = 1E1
   UPDATE T2 SET COL = 1E1
   ```

   In V3R3, these inserts and updates now generate SQLCODE -408 (SQLSTATE 53021).

   If you want to use the value 123, you must now use it as a character literal ('123'). Float literals are no longer allowed for character columns.

4. Invalid String Representation of Datetime

   Prior to V3R3, when a predicate was being evaluated that contained an operand that was one of the special registers CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, and one of the other operands was a character column of the correct length but containing a value that was not a valid string representation of a datetime, the application ran successfully. Any row containing such an invalid value was returned if it met the search condition. For example, all invalid date values in column, ORDERDATE, were returned for the following condition:

   ```
   WHERE CURRENT DATE <> ORDERDATE
   ```

In V3R3, SQLCODE -180 (SQLSTATE 22007) is generated under the above condition.

5. Internally Generated Table Names

   Prior to V3R3, the system internally built a composite table name that included the name of the relational database, based on a certain maximum length.

   In V3R3, this length is slightly increased, and the internal process is now common to the SQL/DS and DRDA protocols. As a result, there is a very small probability that some of your SQL statements could exceed an internal limitation of the system and generate an SQLCODE -101 (SQLSTATE 54001).

   The more table names you have in a statement, the greater the probability of this occurring. If you experience this error, one possible solution would be to break the statement down into two separate statements.

Application Programming

6. Setting of SQLN Field

   Prior to V3R3, if field SQLD in the SQLDA area held a greater value than the SQLN field after a DESCRIBE, the system set SQLN to zero.

   In V3R3, the value of SQLN is not changed.

   If your application tests SQLN for zero to verify successful completion of the DESCRIBE, the logic will have to be revised to test for SQLD > SQLN.

7. C NUL-Terminated Strings - Variable Length

   Prior to V3R3, a C input string with a length greater than 1 was treated as a fixed length character host variable. It was not mandatory to have a NUL present in it except when the input host variable length was 255, in which case SQLCODE -426 (SQLSTATE 22523) was generated.

   In V3R3, a C input string is no longer treated as fixed length. A NUL must be present on all C NUL-terminated input strings except those with a length of 1; otherwise SQLCODE -302 (SQLSTATE 22001) is generated. SQLCODE -426 (SQLSTATE 22523) is no longer generated.

8. C NUL-Terminated Strings - NUL Byte

   Prior to V3R3, the NUL byte in a C NUL-terminated string was treated as a blank.

   In V3R3, it is treated as a string terminator.

9. C NUL-Terminated Strings - Trailing Blanks

   Prior to V3R3, any trailing blanks in a C NUL-terminated string were removed when using the string to update or insert a VARCHAR column or to compare to a VARCHAR column.

   In V3R3, these blanks will no longer be removed.

10. C NUL-Terminated Strings - Length

    Prior to V3R3, the scalar function, LENGTH, with a C NUL-terminated string as its argument, returned the defined length.

    In V3R3, this function now returns the length according to the position of the NUL terminator. (This length excludes the terminator itself.)

11. SQL Statement String

    Prior to V3R3, an SQL statement string could end with a statement terminator, when used in conjunction with EXECUTE IMMEDIATE, PREPARE, or Extended PREPARE. An example of such a statement is

    ```
    DROP TABLE T1;
    ```

which has a trailing semicolon. This was allowed in application programs, even though such coding was invalid. It was also allowed in ISQL and QMF*, since those facilities also use the above three statements to process interactively issued statements.

In V3R3, this statement terminator is not allowed. If it is used, SQLCODE -104 (SQLSTATE 37501) will be generated.

If you have been using such a terminator for the CREATE VIEW statement, your use of catalog table SYSVIEWS could be affected, as described in item "SYSVIEWS" on page 406 under V3R1 and V2R2 Incompatibilities.

12. Preprocessing of Extended Dynamic Statements

    Prior to V3R3, a cursor-variable with a defined length greater than 18 was accepted by the preprocessor, even though such variables should only be defined with a length of 18.

    In V3R3, the preprocessor traps this condition and generates SQLCODE -324 (SQLSTATE spaces). You will have to change any applications that use these invalid cursor-variable lengths in your extended dynamic statements.

13. Data Type of Hexadecimal Constants

    Prior to V3R3, application programs that assumed that hexadecimal constants have a data type of VARGRAPHIC, because they are used in the context of GRAPHIC and VARGRAPHIC data, were accepted.

    In V3R3, such constants are considered to be VARCHAR. If used in conjunction with GRAPHIC or VARGRAPHIC data, they will cause a number of specific SQLCODEs and corresponding SQLSTATEs, dependent on individual cases.

    This also means that SQLCODE -421 (SQLSTATE 53055), dealing with hexadecimal literals of odd length, is no longer generated.

14. Non-updatable View

    Prior to V3R3, a user with DBA authority who tried to update a view that was not updatable got an appropriate error, such as SQLCODE -154 (SQLSTATE 56009). A user without DBA authority, however, got an authorization error, SQLCODE -551 (SQLSTATE 59001).

    In V3R3, the latter user receives the same error message as the DBA user, instead of the authorization message.

15. SYSTEM Table Missing from the System Catalog

    Prior to V3R3, if you tried to INSERT, DELETE, or UPDATE a table or view created by 'SYSTEM', but which was not in the system catalog, SQLCODE -823 (SQLSTATE 53032) was generated, indicating that you lacked proper authorization.

    In V3R3, SQLCODE -204 (SQLCODE 52004) is generated instead, indicating that the object could not be found in the system catalog.

16. Folding of Lowercase in PREP and DBSU

    Prior to V3R3, folding of lowercase into uppercase in PREP and the DBS Utility was done by adding X'40' to the hexadecimal representation of the lowercase character. Sometimes this resulted in characters being folded incorrectly (for example, in the Katakana character set).

    In V3R3, this is done using the 370 built-in Assembler instruction TRANSLATE and the user-specified character translation table, in order to be consistent with how the application server handles this operation. One exception to this is when the DBS Utility processes SCHEMA input files. Folding is no longer done on these files; this makes it consistent with the DBS Utility control file, which only allows uppercase input.

If your applications have built-in dependencies on the previous folding scheme, you could get different results. For example, a Katakana user may have a character in his or her coding scheme that has a hexadecimal value that appears to the database manager as one of the 26 lowercase English letters. Instead of being folded to uppercase English, the Katakana character will now be folded according to the Katakana character translation table.

If you have lowercase in your DBS Utility SCHEMA input file, you will have to change it to uppercase.

17. Loading Audit Trace

    Prior to V3R3, the *Database Administration* manual contained sample table definition and DATALOAD parameters for creating a security audit table and loading trace records into it.

    In V3R3, the position of the columns within the table are changed and a new column, EXTLUWID, added. If you have been loading audit trace data using this table definition and a DATALOAD job, you will need to change the DATALOAD job, as documented in the V3R3 *Database Administration* manual. If you also want to make use of the new EXTLUWID column, you will need to recreate the table as well.

18. Switching Databases without Connect Authority

    Prior to V3R3, if you attempted to switch databases and did not have connect authority for the new database, SQLCODE -561 (SQLSTATE 42505) was generated as a warning situation. It was possible to continue processing on the original database with a non-CONNECT statement.

    In V3R3, this situation is treated as a severe error, SQLWARN0 and SQLWARN6 are set to 'S', and any subsequent non-CONNECT statement results in termination of the application. Only a CONNECT statement is accepted.

19. SQLCODE Generated by Operator FORCE Command

    Prior to V3R3, either SQLCODE -933 (SQLSTATE 57027) or SQLCODE -948 (SQLSTATE 57027) was returned to the application, when the operator issued a FORCE command to roll back the current logical unit of work.

    In V3R3, only SQLCODE -933 (SQLSTATE 57027) is returned.

20. SQLSTATE Changes

    Prior to V3R3, certain SQLCODEs had associated SQLSTATEs that did not conform to the SAA standards.

    In V3R3, these SQLSTATEs are replaced with ones that do conform. See “Detailed Notes on V3R3-V3R2 Incompatibilities” on page 449 for a list of these codes, along with their old and new SQLSTATEs.

System Environment

21. The Use of DBCS Characters with the CHARNAME Setting

    Prior to V3R3, you could use graphic or mixed constants, the VARGRAPHIC scalar function, or you could define columns as GRAPHIC or FOR MIXED DATA, independent of the CHARNAME setting on the application server. Furthermore, you could use graphic or mixed constants, independent of the CHARNAME setting on the application requester.

    In V3R3, the above usages result in error conditions such as SQLCODE -640 (SQLSTATE 56031) and SQLCODE -332 (SQLSTATE 57017), if the corresponding CHARNAME does not define a character set with mixed CCSID (that is, if CCSIDMIXED = 0).

22. Setting of CHARNAME

Prior to V3R3, if no CHARNAME was specified, SQLSTART defaulted to CHARNAME = ENGLISH.

In V3R3, it defaults to the CHARNAME used on the previous invocation. If the CHARNAME setting does not define a character set with mixed CCSID (that is, if CCSIDMIXED = 0), then the default character subtype (CHARSUB) will be forced to a value of SBCS.

See the V3R3 *System Administration* manual for the initial default CHARNAME value after installation or migration.

23. Addressing Mode 31-Bit

    Prior to V3R3, application programs running in single user mode in a VM environment of XA, ESA 1.0 ESA, or ESA 1.1 ESA, as well as any user exits (accounting, datetime, or field procedures) executed in these environments on the database machine, whether single or multiple user mode, only ran in 24-bit addressing mode.

    In V3R3, if the database manager is running in 31-bit addressing mode (AMODE 31) on the database machine, the above application programs and user exits will also run in this mode.

    If you have application programs or user exits that fit into this category, you must do one of the following:

    - Ensure that they can accommodate 31-bit addressing mode
    - Operate the database machine in 370 mode
    - Set the AMODE SQLSTART parameter to 24 to force the database manager to run in 24-bit addressing mode.

    For information on converting your applications to accommodate 31-bit addressing mode, see the *VM/XA* Application Conversion Guide* For more information on single user mode and user exits, see the *System Administration* manual.

24. Section Size in a Package

    Prior to V3R3, during the preprocessing of a program, the system allocated a section size for each statement in the package.

    In V3R3, due to other design changes, it is necessary to increase the size of these sections for SELECT statements. As a result, when an existing package is subjected to a dynamic repreparation, it may cause the dbspace to become full, generating SQLCODE -946 (SQLSTATE 57025).

    If this occurs in your installation, you will have to explicitly prepare the program with the SQLPREP EXEC, making sure that you have a dbspace that can accommodate the revised package.

    Also, the larger sections increase the amount of virtual storage required to run the package. For example, if you have many dynamic SELECT statements in a logical unit of work, they will use up more storage than in the previous release.

25. Three-Part Object Names

    Prior to V3R3, an object that was created on a database named (for example) DBX could be successfully referenced later by an application, even though the name for that database had been changed (to, say, DBY). All you had to do was use the revised name, DBY, when you established the database for the application by means of the SQLINIT EXEC.

    In V3R3, the system maintains the name of the database that was used at the time of the object's creation (DBX in this example), as the first part of the object name, thereby making it a three-part name. If you now establish the database for the application under a different name (for example, DBY) the system uses that name as the new qualifier when you try to reference the object. This results in a mismatch of object names and causes SQLCODE -114 (SQLSTATE 56061) to be generated.

This problem can be avoided by simply not changing the names of your databases.

26. Special Characters for CONCAT Operation and Not Equal Condition

    Prior to V3R3, the class of the hexadecimal values in the table below was 0.

CHARNAME	Hexadecimal Values
ENGLISH	X'5A', X'B0'
FRENCH	X'BA', X'BB'
GERMAN	X'BA', X'BB'
ITALIAN	X'BA', X'BB'
KATAKANA	X'5A', X'B0'
SPANISH	X'BA', X'BB'

In V3R3, the class of these hexadecimal characters is changed to 6. This is reflected in the CHARCLASS column values of the SYSTEM.SYSCHARSETS catalog table. This change provides additional special characters that can be used to depict the CONCAT operation and the not equal condition in SQL syntax. This, in turn, provides greater flexibility in the use of these two SQL facilities between application requesters and servers that are assigned different CHARNAMES.

This could affect your applications, if they are dependent on previous reclassifications of any of the above characters from class 0 to class 3, for use in ordinary identifiers. For example, if you had reclassified the explanation mark (!) so that DANGER! could be used as an ordinary identifier, this will no longer work because the explanation mark is one of the characters that is now assigned to class 6.

See the *DB2 Server for VM System Administration* manual for details on these classifications.

Detailed Notes on V3R3-V3R2 Incompatibilities

1. SQLSTATE Changes

   These changes are shown in the following table.

SQLCODE	Old SQLSTATE	New SQLSTATE	DESCRIPTION
-131	53004	22019	Either the LIKE predicate has an invalid escape character, or the string pattern contains an invalid occurrence of the escape character.
-551	59001	42501	User wwwwww does not have the xxxxxx privilege to perform yyyyyy on zzzzzz.
-552	59002	42502	xxxxxx is not authorized to yyyyyy.
-554	59002	42502	You cannot grant a privilege to yourself.
-555	59002	42502	You cannot revoke an authority or a privilege from yourself.
-556	59002	42502	An attempt to revoke a privilege from xxxxxx was denied. Either xxxxxx does not have this privilege, or yyyyyy does not have this authority to revoke this privilege.

SQLCODE	Old SQLSTATE	New SQLSTATE	DESCRIPTION
-556	59004	42504	An attempt to revoke a privilege from xxxxxx was denied. Either xxxxxx does not have this privilege, or yyyyyy does not have this authority to revoke this privilege.
-558	59004	42504	You cannot revoke an authority from xxxxxx because xxxxxx has DBA authority.
-560	59005	42505	A CONNECT statement contains an incorrect password for xxxxxx.
-561	59005	42505	User xxxxxx does not have CONNECT authority.
-566	59001	42501	User ID xxxxxx does not have authorization to modify package yyyyyy.
-606	59002	42502	The COMMENT ON or LABEL on statement failed because the specified table or column is not owned by xxxxxx.
-610	59002	42502	The statement failed because a user without DBA authority attempted to create a table in a DBSPACE owner by another user or by the system.
-708	59002	42502	You cannot ALTER, LOCK, or DROP a PUBLIC DBSPACE because you do not have DBA authority.
-713	37515	53015	Incorrect isolation level value xxxxxx specified. Only values C or R may be used.
-801	22004	22003	Exception error xxxxxx occurred during yyyyyy operation on zzzzzz data.
-802	22004	22003	Exception error xxxxxx occurred during yyyyyy operation on zzzzzz data, position nnnnnn. psw1 psw2.
-815	59005	42502	CONNECT denied by accounting user exit routine.
-30053	59006	42506	Owner xxxxxx authorization failed.

V3R4 and V3R3 Incompatibilities (VM Only)

Note: This section does not include the restrictions on the use of DRDA protocol, as that topic is covered in the appendix describing DRDA considerations.

SQL and Data

1. Enhanced EXPLAIN Tables

   Prior to V3R4, the tables used by the EXPLAIN statement had some major differences from the corresponding tables in the DB2* product.

   In V3R4, these differences are minimized to enhance the EXPLAIN functions and make them more compatible with those in the DB2 product. As a result, there are significant changes to the design of these tables, and the EXPLAIN statement no longer works on the old tables. These changes include new columns dispersed among old ones, the loss of one column, a column data type change, and a column length change.

   See the *DB2 Server for VSE & VM SQL Reference* manual for the new design of these tables.

   If you have used the EXPLAIN tables in prior releases, you will have to recreate the revised tables before using the EXPLAIN statement in V3R4. To assist you in this task, a DBSU job file containing the necessary create statements is now included as a MACRO file (called ARISEXP) with the product.

Similarly, if you have applications which depend upon the design of the old EXPLAIN tables, you will need to modify these applications to reflect the new design.

Application Programming

2. Reason Codes for Incorrect Host Variable Declarations

   Prior to V3R4, a large number of SQLERRD1 codes were associated with SQLCODE -314 (SQLSTATE spaces) at preprocessor time for invalid host variables.

   In V3R4, with the introduction of host structures and the associated parsing of declaration statements by the preprocessor, the values of some of these SQLERRD1 codes have changed.

   If your application has dependencies on specific SQLERRD1 values, you should look for these changes in the *DB2 Server for VM Messages and Codes* manual and modify your application accordingly.
3. Structured Declarations in COBOL and C

   Prior to V3R4, there were a number of error situations for structure declarations in the SQL DECLARE SECTION that were not checked by the COBOL and C preprocessors.

   In V3R4, these situations are subjected to validation checks, resulting in the following potential errors, which must be corrected before compilation:

SQLCODE	SQLSTATE	Condition
-107	54003	Host variable name too long
-307	spaces	Duplicate host variable names
-314	spaces	Syntax and semantic errors in a host variable

4. Qualified Field Names in RPG

   Prior to V3R4, it was not necessary to qualify the name of a field or subfield in an SQL statement, when that field or subfield name had been duplicated in more than one data structure.

   In V3R4, you must qualify these names as follows:
   - file-name.field-name
   - DS-name.subfield-name

   The preprocessor needs this information in order to interpret the reference. If the qualifier is missing, a preprocessor ARI5370E message is generated.
5. Use of Structures in RPG as Host Variables

   Prior to V3R4, when the database manager referenced an RPG structure as a host variable, one of two things happened:
   - If the structure contained one or more subfields, the database manager accepted the reference. The structure was interpreted as a single character field with a length equal to the length of the total structure.
   - If the structure contained no subfields, the database manager rejected the reference, generating an error message.

   In V3R4:
   - If the structure contains one or more subfields, the reference to it is now interpreted as a reference to each subfield in the structure, giving unpredictable results and potential errors at execution time.

- If the structure contains no subfields, the reference to it is now interpreted as a reference to a fixed length character string with a length equal to the length of the data structure.

Note: Individual subfields within a structure can still be directly referenced as valid host variables. There is no change to this.

If your application references RPG structures as host variables, you will have to change either the declaration section or the SQL statements affected.

6. Application Programs in an Unconnected State

   Prior to V3R4, if an application program was connectable but in an unconnected state as a result of a severe error (SQLWARN6 = S) and issued a non-connect SQL statement, the database manager initiated an abend of the application.

   In V3R4, SQLCODE -900 (SQLSTATE 51018) is generated and the abend does not occur. If your application is dependent on the abend scenario in this situation, you will have to change it. Otherwise, it may enter an infinite loop.

7. Use of Host Variables in CONNECT Statement

   Prior to V3R4, if you used a host variable for the userid or password in a CONNECT statement and the data type of that variable did not satisfy one of the conditions listed below, an error was generated at run time:
   - C programs: C-NUL string of length 9
   - Assembler, COBOL, or PL/I programs: fixed length character string of length 8.

   In V3R4, these conditions are checked by the preprocessor. If they fail the check, SQLCODE -324 (SQLSTATE spaces) is generated.

8. Data Types of Parameter Markers in Predicates

   Prior to V3R4, the resolution of data types for a parameter marker was dependent on the highest order of the data types of all the operands to the left of the parameter marker. Highest order, in the case of numeric operands, implies FLOAT > DECIMAL > INTEGER > SMALLINT.

   In V3R4, this resolution process is changed to become more consistent with the DB2 product. If there is an operand expressed as a column name in a BETWEEN predicate, the data type of any parameter marker is resolved as that of the leftmost such operand. Otherwise, the data type of the parameter marker is resolved as that of the leftmost operand that is not a parameter marker — whether in a BETWEEN predicate or an IN predicate.

   This could cause a different result from previous releases for predicates that can have more than two operands (namely BETWEEN and IN), but only if your application assigns parameter marker values that are inappropriate for your data.

   See “Detailed Notes on V3R4-V3R3 Incompatibilities” on page 454 for some examples and further discussion.

9. Bad Input Records in DATALOAD

   Prior to V3R4, a bad input record would terminate DATALOAD command processing on multiple tables when the DBS Utility was running in multiple user mode — whether or not it was preprocessed with the NOBLOCK option. An insert error would be indicated with one of the following codes, followed by message ARI0862E:

SQLCODE	**SQLSTATE**
-405	53020
-424	22502
-530	23503

-802	22003, 22012, or 22502
-803	23505

In V3R4, such command processing is no longer terminated, if the DBS Utility is preprocessed with the NOBLOCK option. The error indications are still generated, but the processing skips over the bad record and continues.

If you have a dependency in your application on this termination approach prior to V3R4, you may want to address this change in the case of the NOBLOCK option.

10. Index Dependency of a Package

    Prior to V3R4, when a SELECT DISTINCT was applied to a single column that had a unique index, the system assumed uniqueness within the column, rather than applying a sort. However, this kind of index dependency was not recorded in the package.

    In V3R4, this technique now records the index dependency in the package (for system integrity), even though the index is not actually used to access the table. In addition, the technique is extended to column functions that use DISTINCT — for example, SELECT COUNT(DISTINCT(COL4)), where COL4 has a unique index.

    If the index is dropped, the package will now be marked as invalid, causing a dynamic reprep. After the reprep, the application will take longer to execute, because a sort will be needed to process DISTINCT correctly.

System Environment

11. Invocation of TRACE for Storage

    Prior to V3R4, if you specified level 2 trace for the STAT or PA component of the TRACDBSS or TRACRDS parameter, respectively, when starting the database manager, you received the Working Storage Manager tracing.

    In V3R4, you can use the same specifications, but the Working Storage manager tracing is no longer part of the output.

    In order to get this part, you must now use the TRACSTG parameter, or select the STG component when using the TRACE operator command. The format from this trace is different.

12. DBCS Data Conversion Errors

    Prior to V3R4, if there was a loading error in a DBCS data conversion routine, SQLCODE -332 (SQLSTATE 57017) was generated with reason code 9. If there was a dropping error in a DBCS data conversion routine, SQLCODE -901 (SQLSTATE 58004) or SQLCODE -30020 (SQLSTATE 58009) was generated.

    In V3R4, the above codes are replaced with SQLCODE -674 (SQLSTATE 57011) with a separate reason code for each specific error.

13. Saved Segments in Installation Process

    Prior to V3R4, you could install into saved segments during the installation process (with the I5688103 EXEC), or at post installation time.

    In V3R4, this step is no longer in the I5688013 EXEC. Installing into saved segments must be done afterwards.

    If you have automated the running of this EXEC by providing an input file containing the answers to the prompts (rather than submitting them from the console), the EXEC will fail when trying to process your input to the removed saved segment step. You will have to modify your answer file accordingly.

14. Enhancement to COLDLOG

Prior to V3R4, the COLDLOG reconfiguration function erased the log contents before starting the database manager. No warning was given if there were any logical units of work in the log that were needed for recovery processing.

In V3R4, the log content is not erased until after startup, and the user is warned beforehand if the log content is needed for recovery.

If you have automated the COLDLOG function in some way by providing a predetermined set of answers to the prompts (rather than submitting them from the console), the SQLLOG EXEC will fail. You will have to modify your automated process to accommodate the change. See the *DB2 Server for VM System Administration* manual for more information on this function.

Detailed Notes on V3R4-V3R3 Incompatibilities

1. Data Types of Parameter Markers in Predicates

   In this first example, prior releases would resolve the data type of the parameter marker as DEC(4,2), whereas V3R4 would resolve it as INTEGER (assuming INTEGERCOL is the name of a column with a data type of INTEGER).

   ```
   23.55 BETWEEN ? AND INTEGERCOL
   ```

   The next two examples illustrate how these data type differences can produce quite different end results when the SQL statement is executed. In this next example, the predicate would generate SQLCODE -302 (SQLSTATE 22003) in prior releases, when the leftmost parameter marker is assigned a value of 345 and the rightmost parameter marker is assigned a value of 206.7. This error will not occur in V3R4.

   ```
   EDLEVEL IN (16, ?, 17.3, ?)
   ```

   This is because the prior releases assign a data type of DEC(3,1) to the rightmost parameter marker, to which the value 206.7 cannot be assigned. V3R4 assigns a data type of SMALLINT to the rightmost parameter marker (based on the column EDLEVEL) and then truncates 206.7 to accommodate this data type.

   In the next example, the predicate would generate SQLCODE -302 (SQLSTATE 22001) in V3R4, but not in prior releases, when the parameter marker is assigned a value of 'GHIJKL'.

   ```
   DEPTNO IN ('ABCDEF', ?, 'ABC')
   ```

   This is because V3R4 assigns a data type of CHAR(3) to the parameter marker (based on column DEPTNO), to which the value 'GHIJKL' cannot be assigned. Prior releases assign a data type of CHAR(6) to the parameter marker.

V3R4 and V3R2 Incompatibilities (VSE Only)

SQL and Data

1. New Reserved Word, CONCAT

   Prior to V3R4, CONCAT was not a reserved word in SQL and could therefore be used as an ordinary identifier.

   In V3R4, CONCAT is a reserved word, and can be used as an alternative to the concatenation operator (||). Any existing applications that use it as an ordinary identifier will have to be changed before they are preprocessed under V3R4; otherwise SQLCODE -105 (SQLSTATE 37501) will be generated.

You can address this incompatibility by changing this ordinary identifier to use a nonreserved word, or you can retain the original name by redefining it as a delimited identifier.

2. REVOKE UPDATE

   Prior to V3R4, the REVOKE statement for the UPDATE privilege ignored any column names that might be present as parameters of the UPDATE option — even though such coding was invalid. (This statement is only done on a table basis, never a column basis.)

   In V3R4, such parameters are not allowed. If they are used, SQLCODE -105 (SQLSTATE 37501) will be generated.

3. Numeric Data in Character Strings

   Prior to V3R4, columns with a data type of CHAR or VARCHAR accepted numeric data, including FLOAT, on insert or update. For example, the following statements did not create an error:

   ```
   CREATE TABLE T1 (COL CHAR(8))
   CREATE TABLE T2 (COL VARCHAR(8))

   INSERT INTO T1 (123)
   INSERT INTO T2 (123)
   INSERT INTO T1 (1E1)
   INSERT INTO T2 (1E1)

   UPDATE T1 SET COL = 123
   UPDATE T2 SET COL = 123
   UPDATE T1 SET COL = 1E1
   UPDATE T2 SET COL = 1E1
   ```

   In V3R4, these inserts and updates now generate SQLCODE -408 (SQLSTATE 53021).

   If you want to use the value 123, you must now use it as a character literal ('123'). Float literals are no longer allowed for character columns.

4. Invalid String Representation of Datetime

   Prior to V3R4, when a predicate was being evaluated that contained an operand that was one of the special registers CURRENT DATE, CURRENT TIME, or CURRENT TIMESTAMP, and one of the other operands was a character column of the correct length but containing a value that was not a valid string representation of a datetime, the application ran successfully. Any row containing such an invalid value was returned if it met the search condition. For example, all invalid date values in column, ORDERDATE, were returned for the following condition:

   ```
   WHERE CURRENT DATE <> ORDERDATE
   ```

   In V3R4, SQLCODE -180 (SQLSTATE 22007) is generated under the above condition.

5. Internally Generated Table Names

   Prior to V3R4, the system internally built a composite table name that included the name of the relational database, based on a certain maximum length.

   In V3R4, this length is slightly increased, and the internal process is the same, whether DRDA server support is involved or not. As a result, there is a very small probability that some of your SQL statements could exceed an internal limitation of the system and generate an SQLCODE -101 (SQLSTATE 54001).

   The more table names you have in a statement, the greater the probability of this occurring. If you experience this error, one possible solution would be to break the statement down into two separate statements.

6. Enhanced EXPLAIN Tables

Prior to V3R4, the tables used by the EXPLAIN statement had some major differences from the corresponding tables in the DB2* product.

In V3R4, these differences are minimized to enhance the EXPLAIN functions and make them more compatible with those in the DB2 product. As a result, there are significant changes to the design of these tables and the EXPLAIN statement no longer works on the old tables. These changes include new columns dispersed among old ones, the loss of one column, a column data type change, and a column length change.

See the *DB2 Server for VSE & VM SQL Reference* manual for the new design of these tables.

If you have used the EXPLAIN tables in prior releases, you will have to recreate the revised tables before using the EXPLAIN statement in V3R4. To assist you in this task, a DBSU job file containing the necessary create statements is now included as an A-type member (called ARIXEXP) with the product.

Similarly, if you have applications which depend upon the design of the old EXPLAIN tables, you will need to modify these applications to reflect the new design.

Application Programming

7. Setting of SQLN Field

   Prior to V3R4, if field SQLD in the SQLDA area held a greater value than the SQLN field after a DESCRIBE, the system set SQLN to zero.

   In V3R4, the value of SQLN is not changed.

   If your application tests SQLN for zero to verify successful completion of the DESCRIBE, the logic will have to be revised to test for SQLD > SQLN.

8. C NUL-Terminated Strings - Variable Length

   Prior to V3R4, a C input string with a length greater than 1 was treated as a fixed length character host variable. It was not mandatory to have a NUL present in it except when the input host variable length was 255, in which case SQLCODE -426 (SQLSTATE 22523) was generated.

   In V3R4, a C input string is no longer treated as fixed length. A NUL must be present on all C NUL-terminated input strings except those with a length of 1; otherwise SQLCODE -302 (SQLSTATE 22001) is generated. SQLCODE -426 (SQLSTATE 22523) is no longer generated.

9. C NUL-Terminated Strings - NUL Byte

   Prior to V3R4, the NUL byte in a C NUL-terminated string was treated as a blank.

   In V3R4, it is treated as a string terminator.

10. C NUL-Terminated Strings - Trailing Blanks

    Prior to V3R4, any trailing blanks in a C NUL-terminated string were removed when using the string to update or insert a VARCHAR column or to compare to a VARCHAR column.

    In V3R4, these blanks will no longer be removed.

11. C NUL-Terminated Strings - Length

    Prior to V3R4, the SQL/DS scalar function, LENGTH, with a C NUL-terminated string as its argument, returned the defined length.

    In V3R4, this function now returns the length according to the position of the NUL terminator. (This length excludes the terminator itself.)

12. SQL Statement String

Prior to V3R4, an SQL statement string could end with a statement terminator, when used in conjunction with EXECUTE IMMEDIATE, PREPARE, or Extended PREPARE. An example of such a statement is

```
DROP TABLE T1;
```

which has a trailing semicolon. This was allowed in application programs, even though such coding was invalid. It was also allowed in ISQL and QMF*, since those facilities also use the above three statements to process interactively issued statements.

In V3R4, this statement terminator is not allowed. If it is used, SQLCODE -104 (SQLSTATE 37501) will be generated.

If you have been using such a terminator for the CREATE VIEW statement, your use of catalog table SYSVIEWS could be affected, as described in item "SYSVIEWS" on page 406 under V3R1 and V2R2 Incompatibilities.

13. SQL/DS Preprocessing of Extended Dynamic Statements

    Prior to V3R4, a cursor-variable with a defined length greater than 18 was accepted by the preprocessor, even though such variables should only be defined with a length of 18.

    In V3R4, the preprocessor traps this condition and generates SQLCODE -324 (SQLSTATE spaces). You will have to change any applications that use these invalid cursor-variable lengths in your extended dynamic statements.

14. Reason Codes for Incorrect Host Variable Declarations

    Prior to V3R4, a large number of SQLERRD1 codes were associated with SQLCODE -314 (SQLSTATE spaces) at preprocessor time for invalid host variables.

    In V3R4, with the introduction of host structures and the associated parsing of declaration statements by the preprocessor, the values of some of these SQLERRD1 codes have changed.

    If your application has dependencies on specific SQLERRD1 values, you should look for these changes in the *DB2 Server for VM Messages and Codes* or *DB2 Server for VSE Messages and Codes* manual and modify your application accordingly.

15. Structured Declarations in COBOL and C

    Prior to V3R4, there were a number of error situations for structure declarations in the SQL DECLARE SECTION that were not checked by the COBOL and C preprocessors.

    In V3R4, these situations are subjected to validation checks, resulting in the following potential errors, which must be corrected before compilation:

SQLCODE	**SQLSTATE**	**Condition**
-107	54003	Host variable name too long
-307	spaces	Duplicate host variable names
-314	spaces	Syntax and semantic errors in a host variable

16. Data Type of Hexadecimal Constants

    Prior to V3R4, application programs that assumed that hexadecimal constants have a data type of VARGRAPHIC, because they are used in the context of GRAPHIC and VARGRAPHIC data, were accepted.

In V3R4, such constants are considered to be VARCHAR. If used in conjunction with GRAPHIC or VARGRAPHIC data, they will cause a number of specific SQLCODEs and corresponding SQLSTATEs, dependent on individual cases.

This also means that SQLCODE -421 (SQLSTATE 53055), dealing with hexadecimal literals of odd length, is no longer generated.

17. Non-updatable View

    Prior to V3R4, a user with DBA authority who tried to update a view that was not updatable got an appropriate error, such as SQLCODE -154 (SQLSTATE 56009). A user without DBA authority, however, got an authorization error, SQLCODE -551 (SQLSTATE 59001).

    In V3R4, the latter user receives the same error message as the DBA user, instead of the authorization message.

18. SYSTEM Table Missing from the System Catalog

    Prior to V3R4, if you tried to INSERT, DELETE, or UPDATE a table or view created by 'SYSTEM', but which was not in the system catalog, SQLCODE -823 (SQLSTATE 53032) was generated, indicating that you lacked proper authorization.

    In V3R4, SQLCODE -204 (SQLCODE 52004) is generated instead, indicating that the object could not be found in the system catalog.

19. Folding of Lowercase in PREP and DBSU

    Prior to V3R4, folding of lowercase into uppercase in PREP and the DBS Utility was done by adding X'40' to the hexadecimal representation of the lowercase character. Sometimes this resulted in characters being folded incorrectly (for example, in the Katakana character set).

    In V3R4, this is done using the 370 built-in Assembler instruction TRANSLATE and the user-specified character translation table, in order to be consistent with how the application server handles this operation. One exception to this is when the DBS Utility processes SCHEMA input files. Folding is no longer done on these files; this makes it consistent with the DBS Utility control file, which only allows uppercase input.

    If your applications have built-in dependencies on the previous folding scheme, you could get different results. For example, a Katakana user may have a character in his or her coding scheme that has a hexadecimal value that appears to the SQL/DS system as one of the 26 lowercase English letters. Instead of being folded to uppercase English, the Katakana character will now be folded according to the Katakana character translation table.

    If you have lowercase in your DBS Utility SCHEMA input file, you will have to change it to uppercase.

20. Loading Audit Trace

    Prior to V3R4, the *Database Administration* manual contained sample table definition and DATALOAD parameters for creating a security audit table and loading trace records into it.

    In V3R4, the position of the columns within the table are changed and a new column, EXTLUWID, added. If you have been loading audit trace data using this table definition and a DATALOAD job, you will need to change the DATALOAD job, as documented in the V3R4 *Database Administration* manual. If you also want to make use of the new EXTLUWID column, you will need to recreate the table as well.

21. Use of Host Variables in CONNECT Statement

Prior to V3R4, if you used a host variable for the userid or password in a CONNECT statement and the data type of that variable did not satisfy one of the conditions listed below, an error was generated at run time:

- C programs: C-NUL string of length 9
- Assembler, COBOL, or PL/I programs: fixed length character string of length 8.

In V3R4, these conditions are checked by the preprocessor. If they fail the check, SQLCODE -324 (SQLSTATE spaces) is generated.

22. Data Types of Parameter Markers in Predicates

    Prior to V3R4, the resolution of data types for a parameter marker was dependent on the highest order of the data types of all the operands to the left of the parameter marker. Highest order, in the case of numeric operands, implies FLOAT > DECIMAL > INTEGER > SMALLINT.

    In V3R4, this resolution process is changed to become more consistent with the DB2 product. If there is an operand expressed as a column name in a BETWEEN predicate, the data type of any parameter marker is resolved as that of the leftmost such operand. Otherwise, the data type of the parameter marker is resolved as that of the leftmost operand that is not a parameter marker — whether in a BETWEEN predicate or an IN predicate.

    This could cause a different result from previous releases for predicates that can have more than two operands (namely BETWEEN and IN), but only if your application assigns parameter marker values that are inappropriate for your data.

    See "Detailed Notes on V3R4-V3R2 Incompatibilities" on page 462 for some examples and further discussion.

23. Bad Input Records in DATALOAD

    Prior to V3R4, a bad input record would terminate DATALOAD command processing on multiple tables when the DBS Utility was running in multiple user mode — whether or not it was preprocessed with the NOBLOCK option. An insert error would be indicated with one of the following codes, followed by message ARI0862E:

SQLCODE	**SQLSTATE**
-405	53020
-424	22502
-530	23503
-802	22003, 22012, or 22502
-803	23505

    In V3R4, such command processing is no longer terminated, if the DBS Utility is preprocessed with the NOBLOCK option. The error indications are still generated, but the processing skips over the bad record and continues.

    If you have a dependency in your application on this termination approach prior to V3R4, you may want to address this change in the case of the NOBLOCK option.

24. Index Dependency of a Package

    Prior to V3R4, when a SELECT DISTINCT was applied to a single column that had a unique index, the system assumed uniqueness within the column, rather than applying a sort. However, this kind of index dependency was not recorded in the package.

    In V3R4, this technique now records the index dependency in the package (for system integrity), even though the index is not actually used to access the table. In addition, the technique is extended to column functions that use DISTINCT — for example, SELECT COUNT(DISTINCT(COL4)), where COL4 has a unique index.

If the index is dropped, the package will now be marked as invalid, causing a dynamic reprep. After the reprep, the application will take longer to execute, because a sort will be needed to process DISTINCT correctly.

25. SQLSTATE Changes

    Prior to V3R4, certain SQLCODEs had associated SQLSTATEs that did not conform to the SAA standards.

    In V3R4, these SQLSTATEs are replaced with ones that do conform. See "Detailed Notes on V3R4-V3R2 Incompatibilities" on page 462 for a list of these codes, along with their old and new SQLSTATEs.

System Environment

26. The Use of DBCS Characters with the CHARNAME Setting

    Prior to V3R4, you could use graphic or mixed constants, the VARGRAPHIC scalar function, or you could define columns as GRAPHIC or FOR MIXED DATA, independent of the CHARNAME setting on the application server. Furthermore, you could use graphic or mixed constants, independent of the CHARNAME setting on the application requester.

    In V3R4, the above usages result in error conditions such as SQLCODE -640 (SQLSTATE 56031) and SQLCODE -332 (SQLSTATE 57017), if the corresponding CHARNAME does not define a character set with mixed CCSID (that is, if CCSIDMIXED = 0).

27. Setting of CHARNAME

    Prior to V3R4, if no CHARNAME is specified, SQLSTART defaulted to CHARNAME = ENGLISH.

    In V3R4, it defaults to the CHARNAME used on the previous invocation. If the CHARNAME setting does not define a character set with mixed CCSID (that is, if CCSIDMIXED = 0), then the default character subtype (CHARSUB) will be forced to a value of SBCS.

    See the V3R4 *System Administration* manual for the initial default CHARNAME value after installation or migration.

28. Addressing Mode 31-Bit

    Prior to V3R4, user exits and field procedures , executed in a VSE environment, only ran in 24-bit addressing mode.

    In V3R4, with VSE/ESA* 1.3 or later releases, they can be executed in 31-bit addressing mode. If the SQL/DS system is running in 31-bit addressing mode (that is, ESA or VMESA supervisor mode) on the application server, then user exits (except accounting) will be executed in 31-bit addressing mode.

    If you have user exits (except accounting) that fit into this category, you must do one of the following to avoid any potential problems:

    - Ensure that they can accommodate 31-bit addressing mode
    - Operate the application server in 370 or VM supervisor mode.

    For more information on user exits, see the *DB2 Server for VSE System Administration* manual.

29. Section Size in a Package

    Prior to V3R4, during the preprocessing of a program, the system allocated a section size for each statement in the package.

In V3R4, due to other design changes, it is necessary to increase the size of these sections for SELECT statements. As a result, when an existing package is subjected to a dynamic repreparation, it may cause the dbspace to become full, generating SQLCODE -946 (SQLSTATE 57025).

If this occurs in your installation, you will have to explicitly prepare the program with the SQLPREP EXEC, making sure that you have a dbspace that can accommodate the revised package.

Also, the larger sections increase the amount of virtual storage required to run the package. For example, if you have many dynamic SELECT statements in a logical unit of work, they will use up more storage than in the previous release.

30. Three-Part Object Names

    Prior to V3R4, an object that was created on a database named (for example) DBX could be successfully referenced later by an application, even though the name for that database had been changed (to, say, DBY). All you had to do was use the revised name, DBY, when you established the database for the application.

    In V3R4, the system maintains the name of the database that was used at the time of the object's creation (DBX in this example), as the first part of the object name, thereby making it a three-part name. If you now establish the database for the application under a different name (for example, DBY), the system uses that name as the new qualifier when you try to reference the object. This results in a mismatch of object names, and causes SQLCODE -114 (SQLSTATE 56061) to be generated.

    This problem can be avoided by simply not changing the names of your databases.

31. Special Characters for CONCAT Operation and Not Equal Condition

    Prior to V3R4, the class of the hexadecimal values in the table below was 0.

CHARNAME	Hexadecimal Values
ENGLISH	X'5A', X'B0'
FRENCH	X'BA', X'BB'
GERMAN	X'BA', X'BB'
ITALIAN	X'BA', X'BB'
KATAKANA	X'5A', X'B0'
SPANISH	X'BA', X'BB'

In V3R4, the class of these hexadecimal characters is changed to 6. This is reflected in the CHARCLASS column values of the SYSTEM.SYSCHARSETS catalog table. This change provides additional special characters that can be used to depict the CONCAT operation and the not equal condition in SQL syntax. This, in turn, provides greater flexibility in the use of these two SQL facilities between application requesters and servers that are assigned different CHARNAMES.

This could affect your applications, if they are dependent on previous reclassifications of any of the above characters from class 0 to class 3, for use in ordinary identifiers. For example, if you had reclassified the explanation mark (!) so that DANGER! could be used as an ordinary identifier, this will no longer work because the explanation mark is one of the characters that is now assigned to class 6.

See the *DB2 Server for VSE System Administration* manual for details on these classifications.

32. Invocation of TRACE for Storage

    Prior to V3R4, if you specified level 2 trace for the STAT or PA component of the TRACDBSS or TRACRDS parameter, respectively, when starting the SQL/DS system, you received the Working Storage Manager tracing.

    In V3R4, you can use the same specifications, but the Working Storage manager tracing is no longer part of the output.

    In order to get this part, you must now use the TRACSTG parameter, or select the STG component when using the TRACE operator command. The format from this trace is different.

33. Change to Headers in Multiline Operator Console Messages

    Prior to V3R4, for ease of reading, only the first line of a multiline message contained the message header identification, as illustrated below:

    ```
    ARI0418A SQL/DS is not ready. Retry the enable
             transaction CIRB after SQL/DS starts.
    ```

    However, operator console messages which were multiline could not be handled by the VSE Programmed Operator tool, because the system sent such messages one line at a line. The tool could not identify the extra lines.

    In V3R4, these operating console messages are sent as one multiline record, so that the VSE Programmed Operator tool can handle them. (For the console operator, there is no change to the appearance of these messages.)

    If you have your own application equivalent to the above tool, it could be affected by this change.

Detailed Notes on V3R4-V3R2 Incompatibilities

1. Data Types of Parameter Markers in Predicates

   In this first example, prior releases would resolve the data type of the parameter marker as DEC(4,2), whereas V3R4 would resolve it as INTEGER (assuming INTEGERCOL is the name of a column with a data type of INTEGER).

   ```
   23.55 BETWEEN ? AND INTEGERCOL
   ```

   The next two examples illustrate how these data type differences can produce quite different end results when the SQL statement is executed. In this next example, the predicate would generate SQLCODE -302 (SQLSTATE 22003) in prior releases, when the leftmost parameter marker is assigned a value of 345 and the rightmost parameter marker is assigned a value of 206.7. This error will not occur in V3R4.

   ```
   EDLEVEL IN (16, ?, 17.3, ?)
   ```

   This is because the prior releases assign a data type of DEC(3,1) to the rightmost parameter marker, to which the value 206.7 cannot be assigned. V3R4 assigns a data type of SMALLINT to the rightmost parameter marker (based on the column EDLEVEL) and then truncates 206.7 to accommodate this data type.

   In the next example, the predicate would generate SQLCODE -302 (SQLSTATE 22001) in V3R4, but not in prior releases, when the parameter marker is assigned a value of 'GHIJKL'.

   ```
   DEPTNO IN ('ABCDEF', ?, 'ABC')
   ```

   This is because V3R4 assigns a data type of CHAR(3) to the parameter marker (based on column DEPTNO), to which the value 'GHIJKL' cannot be assigned. Prior releases assign a data type of CHAR(6) to the parameter marker.

2. SQLSTATE Changes

   These changes are shown in the following table.

SQLCODE	Old SQLSTATE	New SQLSTATE	DESCRIPTION
-131	53004	22019	Either the LIKE predicate has an invalid escape character, or the string pattern contains an invalid occurrence of the escape character.
-551	59001	42501	User wwwwww does not have the xxxxxx privilege to perform yyyyyy on zzzzzz.
-552	59002	42502	xxxxxx is not authorized to yyyyyy.
-554	59002	42502	You cannot grant a privilege to yourself.
-555	59002	42502	You cannot revoke an authority or a privilege from yourself.
-556	59002	42502	An attempt to revoke a privilege from xxxxxx was denied. Either xxxxxx does not have this privilege, or yyyyyy does not have this authority to revoke this privilege.
-556	59004	42504	An attempt to revoke a privilege from xxxxxx was denied. Either xxxxxx does not have this privilege, or yyyyyy does not have this authority to revoke this privilege.
-558	59004	42504	You cannot revoke an authority from xxxxxx because xxxxxx has DBA authority.
-560	59005	42505	A CONNECT statement contains an incorrect password for xxxxxx.
-561	59005	42505	User xxxxxx does not have CONNECT authority.
-566	59001	42501	User ID xxxxxx does not have authorization to modify package yyyyyy.
-606	59002	42502	The COMMENT ON or LABEL on statement failed because the specified table or column is not owned by xxxxxx.
-610	59002	42502	The statement failed because a user without DBA authority attempted to create a table in a DBSPACE owner by another user or by the system.
-708	59002	42502	You cannot ALTER, LOCK, or DROP a PUBLIC DBSPACE because you do not have DBA authority.
-713	37515	53015	Incorrect isolation level value xxxxxx specified. Only values C or R may be used.
-801	22004	22003	Exception error xxxxxx occurred during yyyyyy operation on zzzzzz data.
-802	22004	22003	Exception error xxxxxx occurred during yyyyyy operation on zzzzzz data, position nnnnnn. psw1 psw2.
-815	59005	42502	CONNECT denied by accounting user exit routine.
-30053	59006	42506	Owner xxxxxx authorization failed.

V3R5 and V3R4 Incompatibilities

1. SQL/DS Database Archive Incompatibilities

   Archives that were created on prior releases of SQL/DS cannot be restored by the SQL/DS V3R5 database manager. If this is attempted, the database manager will issue message ARI2038E and terminate. See the *DB2 Server for VM Messages and Codes* or *DB2 Server for VSE Messages and Codes* manual for more details on this message.

2. SQL/DS VSAM Shareoptions Changes under VSE

   In prior releases of SQL/DS (VSE), the VSAM SQL/DS directory, data and log data sets were defined with SHAREOPTIONS(1). In SQL/DS V3R5, these VSAM files must now be defined with SHAREOPTIONS(2).

3. SQLSTATE Values Changes

   Many SQLSTATE values have changed in SQL/DS V3R5. The new SQLSTATE values and their former values can be found in the *DB2 Server for VM Messages and Codes* or *DB2 Server for VSE Messages and Codes* manuals. Changing SQLSTATEs is an incompatible change since many SQLSTATE values that are returned from diagnostic situations will be different from previous releases of SQL/DS. Application programmers should review any programs that use SQLSTATE in the SQLCA each time an SQL statement is executed.

4. Messages and Codes Changes

   Some SQL/DS messages and codes have changed, and some new ones have been added in SQL/DS V3R5. See the *DB2 Server for VM Messages and Codes* and *DB2 Server for VSE Messages and Codes* manuals for details.

5. Display CICS Information on SHOW CONNECT

   If the package that the connected user is running was created in SQL/DS Version 2 Release 2 or earlier, the CICS information will not be displayed by the SHOW CONNECT command because the RDIIN for V2R2 or earlier does not contain the RDIIN extension area. The package must be reprepped with SQL/DS V3R5 and recompiled to make the CICS information available.

V5R1 and V3R5 Incompatibilities

1. Messages and Codes Changes

   Many messages and codes have changed, and some new ones have been added in DB2 Server for VSE & VM Version 5 Release 1. See the *DB2 Server for VM Messages and Codes* and *DB2 Server for VSE Messages and Codes* manuals.

2. DB2 Database Archive Incompatibilities

   Archives that were created on prior releases cannot be restored by the DB2 Server for VSE & VM Version 5 Release 1 database manager. If this is attempted the database manager will issue message ARI2038E and terminate. See the *DB2 Server for VM Messages and Codes* and *DB2 Server for VSE Messages and Codes* manuals for more details on this message.

3. DBSU

   If you use R350 DBSU to unload and reload a table in a R510 database, the value of the DATACAPTURE column will be lost.

4. Date/Time Exits and Field Procedures

   VM Users with Date/Time or Field Procedure Exits that are dependant on running in a 370 Mode virtual machine must convert to execute in a ESA mode virtual machine. Note that exits requiring AMODE=24 are not affected, as we still support running the Server code in AMODE=24. The above also applies to Single User Mode application programs. The above also applies to Vendor programs that run on the Server, such as database monitoring or tape mount handling programs.

V6R1 and V5R1 Incompatibilities

1. Running the Database Server in 24-bit Addressing Mode (VM)

With Version 7 Release 5 the RDS component is linkedited with the AMODE ANY option, instead of AMODE 24. This allows RDS to be loaded and executed above the 16 MB line. This will free up valuable storage below the 16 MB line. However, if you use the AMODE(24) parameter, then RDS cannot be executed above the line. If this is attempted, a program check will occur at start up time.

To avoid this, you must use a maximum virtual storage size of 16MB which will force RDS to be loaded below the line. If you need to run with AMODE(24) all of the time, you should create an RDS saved segment that resides below the 16MB line. If you only use AMODE(24) some of the time, such as with some single user mode applications, you can create an alternate bootstrap package which specifies an alternate RDS saved segment which resides below the 16MB line, or specifies that RDS is run from free storage.

The AMODE parameter value is saved in the ″resid SQLDBN Q″ file. See the *DB2 Server for VM System Administration* or *DB2 Server for VSE System Administration* manual for details on the AMODE parameter and saved segments.

2. Exploiting RDS above the 16 Megabyte Line

   With Version 7 Release 5, the RDS component is linkedited with the ″RMODE ANY″ option. This allows RDS to be loaded and executed above the 16MB line. This will free up valuable storage below the 16 MB line. As the RDS code will be loaded above the 16MB line before other storage is allocated, extremely storage constrained systems may need to increase their partition size to maximize their below the 16MB line free storage.

3. DBNAME Directory format change

   The format of the DBNAME directory source member, ARISDIRD, has been changed to support DRDA Online Requester support.

V7R1 and V6R1 Incompatibilities

There are no incompatibilities between DB2 Server for VM V6R1 and DB2 Server for VM V7R1.

DB2 Server for VSE only:

1. DBNAME Directory format change

   ARISDIRD has been restructured to improve readability and flexibility. Each DBNAME entry is now defined explicitly by its type (Local, Remote or Host VM (Guest Sharing)). CICS AXE Transaction TPNs (Transaction Program Names) are still included in the directory as a type of ’LOCALAXE’. The DBNAME Directory Builder program, ARICBDID has been rewritten as a REXX/VSE procedure with extensive error and dependency checking. Support for TCP/IP information is added and ’alias’ DBNAMEs are supported. **ALL** DBNAMEs **must** be specified in the new DBNAME Directory, including the Product Default DBNAME ″SQLDS″. A migration REXX/VSE procedure, ARICCDID, is provided to assist in migrating to the new format.

V7R2 and V7R1 Incompatibilities

There are no incompatibilities between DB2 Server for VM V7R2 and DB2 Server for VM V7R1.

There are no incompatibilities between DB2 Server for VSE V7R2 and DB2 Server for VSE V7R1.

Notices

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain services of DB2 Server for VSE & VM.

Trademarks

Bibliography

This bibliography lists publications that are referenced in this manual or that may be helpful.

DB2 Server for VM Publications

- *DB2 Server for VSE & VM Application Programming*, SC09-2889
- *DB2 Server for VSE & VM Database Administration*, SC09-2888
- *DB2 Server for VSE & VM Database Services Utility*, SC09-2983
- *DB2 Server for VSE & VM Diagnosis Guide and Reference*, LC09-2907
- *DB2 Server for VSE & VM Overivew*, GC09-2995
- *DB2 Server for VSE & VM Interactive SQL Guide and Reference*, SC09-2990
- *DB2 Server for VSE & VM Master Index and Glossary*, SC09-2890
- *DB2 Server for VM Messages and Codes*, GC09-2984
- *DB2 Server for VSE & VM Operation*, SC09-2986
- *DB2 Server for VSE & VM Quick Reference*, SC09-2988
- *DB2 Server for VM System Administration*, SC09-2980
- *DB2 Server for VSE & VM Performance Tuning Handbook*, GC09-2987
- *DB2 Server for VSE & VM SQL Reference*, SC09-2989

DB2 Server for VSE Publications

- *DB2 Server for VSE & VM Application Programming*, SC09-2889
- *DB2 Server for VSE & VM Database Administration*, SC09-2888
- *DB2 Server for VSE & VM Database Services Utility*, SC09-2983
- *DB2 Server for VSE & VM Diagnosis Guide and Reference*, LC09-2907
- *DB2 Server for VSE & VM Overivew*, GC09-2995
- *DB2 Server for VSE & VM Interactive SQL Guide and Reference*, SC09-2990
- *DB2 Server for VSE & VM Master Index and Glossary*, SC09-2890
- *DB2 Server for VSE Messages and Codes*, GC09-2985
- *DB2 Server for VSE & VM Operation*, SC09-2986
- *DB2 Server for VSE System Administration*, SC09-2981
- *DB2 Server for VSE & VM Performance Tuning Handbook*, GC09-2987
- *DB2 Server for VSE & VM SQL Reference*, SC09-2989

Related Publications

- *DB2 Server for VSE & VM Data Restore*, SC09-2991
- *DRDA: Every Manager's Guide*, GC26-3195
- *IBM SQL Reference, Version 2, Volume 1*, SC26-8416
- *IBM SQL Reference*, SC26-8415

VM/ESA Publications

- *VM/ESA: General Information*, GC24-5745
- *VM/ESA: VMSES/E Introduction and Reference*, GC24-5837
- *VM/ESA: Installation Guide*, GC24-5836
- *VM/ESA: Service Guide*, GC24-5838
- *VM/ESA: Planning and Administration*, SC24-5750
- *VM/ESA: CMS File Pool Planning, Administration, and Operation*, SC24-5751
- *VM/ESA: REXX/EXEC Migration Tool for VM/ESA*, GC24-5752
- *VM/ESA: Conversion Guide and Notebook*, GC24-5839
- *VM/ESA: Running Guest Operating Systems*, SC24-5755
- *VM/ESA: Connectivity Planning, Administration, and Operation*, SC24-5756
- *VM/ESA: Group Control System*, SC24-5757
- *VM/ESA: System Operation*, SC24-5758
- *VM/ESA: Virtual Machine Operation*, SC24-5759
- *VM/ESA: CP Programming Services*, SC24-5760
- *VM/ESA: CMS Application Development Guide*, SC24-5761
- *VM/ESA: CMS Application Development Reference*, SC24-5762
- *VM/ESA: CMS Application Development Guide for Assembler*, SC24-5763
- *VM/ESA: CMS Application Development Reference for Assembler*, SC24-5764

- *VM/ESA: CMS Application Multitasking*, SC24-5766
- *VM/ESA: CP Command and Utility Reference*, SC24-5773
- *VM/ESA: CMS Primer*, SC24-5458
- *VM/ESA: CMS User's Guide*, SC24-5775
- *VM/ESA: CMS Command Reference*, SC24-5776
- *VM/ESA: CMS Pipelines User's Guide*, SC24-5777
- *VM/ESA: CMS Pipelines Reference*, SC24-5778
- *VM/ESA: XEDIT User's Guide*, SC24-5779
- *VM/ESA: XEDIT Command and Macro Reference*, SC24-5780
- *VM/ESA: Quick Reference*, SX24-5290
- *VM/ESA: Performance*, SC24-5782
- *VM/ESA: Dump Viewing Facility*, GC24-5853
- *VM/ESA: System Messages and Codes*, GC24-5841
- *VM/ESA: Diagnosis Guide*, GC24-5854
- *VM/ESA: CP Diagnosis Reference*, SC24-5855
- *VM/ESA: CP Diagnosis Reference Summary*, SX24-5292
- *VM/ESA: CMS Diagnosis Reference*, SC24-5857
- CP and CMS control block information is not provided in book form. This information is available on the IBM VM/ESA operating system home page (http://www.ibm.com/s390/vm).
- *IBM VM/ESA: CP Exit Customization*, SC24-5672
- *VM/ESA REXX/VM User's Guide*, SC24-5465
- *VM/ESA REXX/VM Reference*, SC24-5770

C for VM/ESA Publications

- *IBM C for VM/ESA Diagnosis Guide*, SC09-2149
- *IBM C for VM/ESA Language Reference*, SC09-2153
- *IBM C for VM/ESA Compiler and Run-Time Migration Guide*, SC09-2147
- *IBM C for VM/ESA Programming Guide*, SC09-2151
- *IBM C for VM/ESA User's Guide*, SC09-2152

Virtual Storage Extended/Enterprise Systems Architecture (VSE/ESA) Publications

- *IBM VSE/ESA Administration*, SC33-6505
- *IBM VSE/ESA Diagnosis Tools*, SC33-6514
- *IBM VSE/ESA General Information*, GC33-6501
- *IBM VSE/ESA Guide for Solving Problems*, SC33-6510
- *IBM VSE/ESA Guide to System Functions*, SC33-6511
- *IBM VSE/ESA Installation*, SC33-6504
- *IBM VSE/ESA Messages & Codes*, SC33-6507
- *IBM VSE/ESA Networking Support*, SC33-6508
- *IBM VSE/ESA Operation*, SC33-6506
- *IBM VSE/ESA Planning*, SC33-6503
- *IBM VSE/ESA System Control Statements*, SC33-6513
- *IBM VSE/ESA System Macros User's Guide*, SC33-6515
- *IBM VSE/ESA System Macros Reference*, SC33-6516
- *IBM VSE/ESA System Utilities*, SC33-6517
- *IBM VSE/ESA Unattended Node Support*, SC33-6512
- *IBM VSE/ESA Using IBM Workstations*, SC33-6509

CICS/VSE Publications

- *CICS/VSE Application Programming Reference*, SC33-0713
- *CICS/VSE Application Programming Guide*, SC33-0712
- *CICS Application Programming Primer (VS COBOL II)*, SC33-0674
- *CICS/VSE CICS-Supplied Transactions*, SC33-0710
- *CICS/VSE Customization Guide*, SC33-0707
- *CICS/VSE Facilities and Planning Guide*, SC33-0718
- *CICS/VSE Intercommunication Guide*, SC33-0701
- *CICS/VSE Performance Guide*, SC33-0703
- *CICS/VSE Problem Determination Guide*, SC33-0716
- *CICS/VSE Recovery and Restart Guide*, SC33-0702
- *CICS/VSE Release Guide*, GC33-1645
- *CICS/VSE Report Controller User's Guide*, SC33-0705
- *CICS Transaction Server for VSE/ESA V1R1.0 Resource Definition Guide*, SC33-0709
- *CICS/VSE Resource Definition (Online)*, SC33-0708
- *CICS/VSE System Definition and Operations Guide*, SC33-0706
- *CICS/VSE System Programming Reference*, SC33-0711
- *CICS/VSE User's Handbook*, SX33-6079
- *CICS/VSE XRF Guide*, SC33-0704

CICS/ESA Publications

- *CICS/ESA General Information*, GC33-0803

VSE/Virtual Storage Access Method (VSE/VSAM) Publications

- *VSE/VSAM Commands and Macros*, SC33-6532
- *VSE/VSAM Introduction*, GC33-6531
- *VSE/VSAM Messages and Codes*, SC24-5146
- *VSE/VSAM Programmer's Reference*, SC33-6535

VSE/Interactive Computing and Control Facility (VSE/ICCF) Publications

- *VSE/ICCF Administration and Operation*, SC33-6562
- *VSE/ICCF Primer*, SC33-6561
- *VSE/ICCF User's Guide*, SC33-6563

VSE/POWER Publications

- *VSE/POWER Administration and Operation*, SC33-6571
- *VSE/POWER Application Programming*, SC33-6574
- *VSE/POWER Networking*, SC33-6573
- *VSE/POWER Remote Job Entry*, SC33-6572

Distributed Relational Database Architecture (DRDA) Library

- *Application Programming Guide*, SC26-4773
- *Architecture Reference*, SC26-4651
- *Connectivity Guide*, SC26-4783
- *DRDA: Every Manager's Guide*, GC26-3195
- *Planning for Distributed Relational Database*, SC26-4650
- *Problem Determination Guide*, SC26-4782

C/370 for VSE Publications

- *IBM C/370 General Information*, GC09-1386
- *IBM C/370 Programming Guide for VSE*, SC09-1399
- *IBM C/370 Installation and Customization Guide for VSE*, GC09-1417
- *IBM C/370 Reference Summary for VSE*, SX09-1246
- *IBM C/370 Diagnosis Guide and Reference for VSE*, LY09-1805

VSE/REXX Publication

- *VSE/REXX Reference*, SC33-6642

Other Distributed Data Publications

- *IBM Distributed Data Management (DDM) Architecture, Architecture Reference, Level 4*, SC21-9526
- *IBM Distributed Data Management (DDM) Architecture, Implementation Programmer's Guide*, SC21-9529
- *VM/Directory Maintenance Licensed Program Specification*, GC20-1836
- *IBM Distributed Relational Database Architecture Reference*, SC26-4651
- *IBM Systems Network Architecture, Format and Protocol Reference*, SC30-3112
- *SNA LU 6.2 Reference: Peer Protocols*, SC31-6808
- *Reference Manual: Architecture Logic for LU Type 6.2*, SC30-3269
- *IBM Systems Network Architecture, Logical Unit 6.2 Reference: Peer Protocols*, SC31-6808
- *Distributed Data Management (DDM) General Information*, GC21-9527

CCSID Publications

- *Character Data Representation Architecture, Executive Overview*, GC09-2207
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

DB2 Server RXSQL Publications

- *DB2 REXX SQL for VM/ESA Installation and Reference*, SC09-2891

C/370 Publications

- *IBM C/370 Installation and Customization Guide*, GC09-1387
- *IBM C/370 Programming Guide*, SC09-1384

Communication Server for OS/2 Publications

- *Up and Running!*, GC31-8189
- *Network Administration and Subsystem Management Guide*, SC31-8181
- *Command Reference*, SC31-8183
- *Message Reference*, SC31-8185
- *Problem Determination Guide*, SC31-8186

Distributed Database Connection Services (DDCS) Publications

- *DDCS User's Guide for Common Servers*, S20H-4793
- *DDCS for OS/2 Installation and Configuration Guide*, S20H-4795

VTAM Publications

- *VTAM Messages and Codes*, SC31-6493
- *VTAM Network Implementation Guide*, SC31-6494
- *VTAM Operation*, SC31-6495
- *VTAM Programming*, SC31-6496
- *VTAM Programming for LU 6.2*, SC31-6497
- *VTAM Resource Definition Reference*, SC31-6498
- *VTAM Resource Definition Samples*, SC31-6499

CSP/AD and CSP/AE Publications

- *Developing Applications*, SH20-6435
- *CSP/AD and CSP/AE Installation Planning Guide*, GH20-6764
- *Administering CSP/AD and CSP/AE on VM*, SH20-6766
- *Administering CSP/AD and CSP/AE on VSE*, SH20-6767
- *CSP/AD and CSP/AE Planning*, SH20-6770
- *Cross System Product General Information*, GH23-0500

Query Management Facility (QMF) Publications

- *Introducing QMF*, GC27-0714
- *Installing and Managing QMF for VSE*, GC27-0721
- *QMF Reference*, SC27-0715
- *Installing and Managing QMF for VM*, GC27-0720
- *Developing QMF Applications*, SC27-0718
- *QMF Messages and Codes*, GC27-0717
- *Using QMF*, SC27-0716

Query Management Facility (QMF) for Windows Publications

- *Getting Started with QMF for Windows*, SC27-0723
- *Installing and Managing QMF for Windows*, GC27-0722

DL/I DOS/VS Publications

- *DL/I DOS/VS Application Programming*, SH24-5009

COBOL Publications

- *VS COBOL II Migration Guide for VSE*, GC26-3150
- *VS COBOL II Migration Guide for MVS and CMS*, GC26-3151
- *VS COBOL II General Information*, GC26-4042
- *VS COBOL II Language Reference*, GC26-4047
- *VS COBOL II Application Programming Guide*, SC26-4045
- *VS COBOL II Application Programming Debugging*, SC26-4049
- *VS COBOL II Installation and Customization for CMS*, SC26-4213
- *VS COBOL II Installation and Customization for VSE*, SC26-4696
- *VS COBOL II Application Programming Guide for VSE*, SC26-4697

Data Facility Storage Management Subsystem/VM (DFSMS/VM) Publications

- *DFSMS/VM RMS User's Guide and Reference*, SC35-0141

Systems Network Architecture (SNA) Publications

- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *SNA Format and Protocol Reference: Architecture Logic for LU Type 6.2*, SC30-3269
- *SNA LU 6.2 Reference: Peer Protocols*, SC31-6808
- *SNA Synch Point Services Architecture Reference*, SC31-8134

Miscellaneous Publications

- *IBM 3990 Storage Control Planning, Installation, and Storage Administration Guide*, GA32-0100
- *Dictionary of Computing*, ZC20-1699
- *APL2 Programming: Using Structured Query Language*, SH21-1056
- *ESA/390 Principles of Operation*, SA22-7201

Related Feature Publications

- *DB2 for VM Control Center Operations Guide*, GC09-2993
- *DB2 for VSE Control Center Operations Guide*, GC09-2992
- *DB2 Replication Guide and Reference*, SC26-9920

Index

Contacting IBM

Before you contact DB2 customer support, check the product manuals for help with your specific technical problem.

For information or to order any of the DB2 Server for VSE & VM products, contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

If you live in the U.S.A., then you can call one of the following numbers:
- 1-800-237-5511 for customer support
- 1-888-426-4343 to learn about available service options

Product information

DB2 Server for VSE & VM product information is available by telephone or by the World Wide Web at http://www.ibm.com/software/data/db2/vse-vm

This site contains the latest information on the technical library, product manuals, newsgroups, APARs, 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 http://www.ibm.com/planetwide

In some countries, IBM-authorized dealers should contact their dealer support structure for information.

File Number: S370/4300-50
Program Number: 5697-F42

Printed in USA

SC09-2989-02

Spine information:

IBM DB2 Server for VSE & VM SQL Reference Version 7 Release 5