IBM

Application System/400™

SC21-9608-1

Programming:
Structured Query Language/400 Reference

Application System/400™

SC21-9608-1

Programming:
Structured Query Language/400 Reference

| **Second Edition (September 1989)**

| This major revision makes obsolete SC21-9608-0.

| This edition applies to Release 2 Modification Level 0 of the IBM Operating System/400 Licensed
| Program (Program 5728-ST1), and to all subsequent releases and modifications until otherwise
| indicated in new editions or technical newsletters.

See About This Manual for a summary of major changes to this edition. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions or technical newsletters.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to your IBM-approved remarketer.

This publication could contain technical inaccuracies or typographical errors.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department 245, Rochester, Minnesota, U.S.A. 55901. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About this Manual

This manual contains reference information for the tasks of system administration, database administration, application programming, and operation. It presents detailed information on Structured Query Language/400 (SQL/400), including syntax, usage notes, keywords, and examples for each of the SQL statements implemented on the AS/400 system.

This manual may refer to products announced but not yet available.

Who Should Use This Manual

This book is intended for programmers who want to write applications that will use SQL to access an AS/400 system database.

What You Should Know

It is assumed that you possess an understanding of system administration, database administration, or application programming in the AS/400 system environment, as provided by the *SQL/400 Programmer's Guide*, and that you have some knowledge of the following:

- A programming language (RPGIII, COBOL/400, C, and/or PL/I)
- Structured Query Language (SQL)

This book is a reference rather than a tutorial. It assumes you are already familiar with SQL programming. This book also assumes that you will be writing applications solely for the AS/400 system environment and therefore presents the full functions of the AS/400 system. Should you be planning applications which will be ported to other Systems Applications Architecture (SAA) environments, it will be necessary for you to reference the appropriate SAA books in addition to this one.

How This Manual Is Organized

This book has the following sections:

- Chapter 1 discusses the basic concepts of relational databases and SQL.
- Chapter 2 describes the basic syntax of SQL and the language elements that are common to many SQL statements.
- Chapter 3 contains syntax diagrams, semantic descriptions, rules, and usage examples of SQL column and scalar functions.
- Chapter 4 describes the various forms of a query, which is a component of various SQL statements.
- Chapter 5 contains syntax diagrams, semantic descriptions, rules, and examples of all SQL statements.
- The appendixes contain information about SQL limits, a description of the SQLCA and SQLDA control blocks, and a list of reserved words.

This manual also contains a glossary of terms and abbreviations, and an index.

Related Online Information

The following online information is available on the AS/400 system. You can press the Help key a second time to see an explanation of how the online information works, including the index search function.

Help for Displays

You can press the Help key on any display to see information about the display. There are two types of help available:

General
Specific

General help explains the purpose of the display. General help appears if you press the Help key when the cursor is outside the areas for which specific help is available.

Specific help explains the field on which the cursor is positioned when you press the Help key. For example, it describes the choices available for a prompt. If a system message appears at the bottom of the display, position the cursor on the message and press the Help key to see information about the cause of the message and the appropriate action to take.

To exit the online information, press F3 (Exit). You return to the display on which you pressed the Help key.

Index Search

Index search allows you to specify the words or phrases you want to see information about. To use index search, press the Help key, then press F11 (Search index).

Help for Control Language Commands

To see prompts for parameters for a control language command, type the command, then press the Help key or F4 (Prompt) instead of the Enter key.

Online Education

AS/400 system online education provides tutorials on a wide variety of topics. To use the online education, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use online education.

Question-and-Answer Function

The question-and-answer (Q & A) function provides answers to questions you may have about using an AS/400 system. To use the Q & A function, press F13 (User support) on any system menu to show the User Support menu. Then select the option to use the question-and-answer function.

Related Printed Information

If you need more information about using SQL statements, statement syntax and parameters, see *Programming: Structured Query Language/400 Programmer's Guide, SC21-9609*.

If you need more information about the interactive data definition utility, see *Utilities: Interactive Data Definition Utility User's Guide, SC21-9657*.

For more information about AS/400 system security, see *Programming: Security Concepts and Planning, SC21-8083*.

For more information about entering source and syntax checking of host language and SQL statements, see *Application Development Tools: Source Entry Utility User's Guide and Reference, SC09-1172*.

For more information about AS/400 system control language commands and AS/400 system programming, see the following:

- *Languages: COBOL/400 User's Guide, SC09-1158*
- *Languages: COBOL/400 Reference, SC09-1240*
- *Languages: COBOL/400 Reference Summary, SX09-1049*
- *Languages: PL/I Reference Summary, SX09-1051*
- *Languages: PL/I User's Guide and Reference, SC09-1156*
- *Languages: C User's Guide*
- *Languages: RPG/400 Reference, SC09-1161*
- *Programming: Command Reference Summary, SC21-8076*
- *Programming: Control Language Programmer's Guide, SC21-8077*
- *Programming: Control Language Reference, SBOF-0481*

For more information about databases, see the following:

- *Programming: Backup and Recovery Guide, SC21-8079*
- *Programming: Database Guide, SC21-9659*
- *Programming: Data Description Specifications Reference, SC21-9620*

How This Manual Has Changed

The following is a list of the major changes or additions that have been made to this manual:

- CREATE DATABASE has been replaced by CREATE COLLECTION.
- DROP DATABASE has been replaced by DROP COLLECTION.
- Chapter 1, "Concepts" on page 1, has been expanded to include a section on collections.
- The section "How SQL Statements Are Invoked" on page 50 has been expanded to include the following sections:
  - Embedding a Statement in an Application Program
  - Dynamic Preparation and Execution
  - Static Invocation of a select-statement
  - Dynamic Invocation of a select-statement
  - Interactive Invocation
- Information on support for the C programming language has been added where appropriate throughout the book. This includes changes to the following sections:

- INCLUDE SQLCA, on pages 91 and 131.
- INCLUDE SQLDA, on page 137.

- A list of reserved words, included in Appendix C, "Reserved Words" on page 139, has been added.

Changes since the previous edition of this manual are indicated by a vertical line to the left of the change.

Contents

Chapter 1. Concepts

Structured Query Language (SQL) is the language used to access data in a relational database. SQL is unlike many programming and data languages because you do not have to code a sequence of instructions explaining how to access the data. SQL allows you to select data by using a single statement directed to the database manager. It is the function of the database manager to access and to maintain the data.

SQL provides full data definition and data manipulation capabilities. You can use it to define objects such as indexes, tables, and views. You can also retrieve, insert, update, and delete data, and control access authorization to data.

The SQL statements can be:

- Embedded inside application programs written in other languages, such as RPG, COBOL, C, and PL/I.

  This is called *static* SQL. The SQL statements are present in the program at the time it is precompiled.

- Typed in from a terminal or built by a program.

  This is termed *interactive* or *dynamic* SQL. The SQL statements are not provided to the database manager until the program runs.

Static SQL

SQL programmers can write source programs containing static SQL statements. Before a COBOL, RPG, C, or PL/I program containing static SQL statements is compiled, the SQL precompiler flags the SQL statements as comments and includes the code necessary to invoke the database manager. Then the compiler can process the program. The precompiler also checks the syntax of the SQL statements.

Dynamic SQL

A capability to enter SQL statements from a terminal is part of the architecture of SQL. You can write programs that read SQL statements from terminals. Programs that you write use dynamic SQL to process SQL statements and present the results to users. Dynamic SQL allows you to create your own query programs, tailored to your users and designed for your specific needs.

Collections

The objects in a relational database are organized into sets called collections. A collection provides a logical classification of objects in the database.

When a table, view, or index is created, it is assigned to exactly one collection. The collection to which an object is assigned is determined by the name of the object. For example, CREATE TABLE C.X creates table X in collection C.

Tables

A relational database is a set of *tables*. Tables are logical structures maintained by the database manager. Tables are made up of columns and rows. There is no inherent order of the rows within a table. At the intersection of every column and row is a specific data item called a *value*. A *column* is a set of values of the same type. A *row* is a sequence of values such that the *n*th value is a value of the *n*th column of the table.

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

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. (A key is a column, or an ordered collection of columns, on which the index is created.)

Views

Views provide an alternative way of looking at the data in one or more tables.

Like tables, views have rows and columns with no inherent order of rows. You specify view names in the FROM clause of the SELECT statement just as you specify table names. You can create views and authorize usage for table-like operations. Certain operations are not valid on views; otherwise, users never need know they are working with a view and not with a table.

A table has a storage representation, but a view does not. When a view is created, its definition is stored in the catalog. No data is stored and, therefore, no index can 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.

Catalog

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

Tables and views in the catalog are like any other database tables and views. If you have authorization, you can use SQL statements to look at data in the catalog views in the same way you retrieve data from any other table in the

AS/400 system. The database manager ensures that the catalog contains accurate descriptions of the databases at all times.

Application Processes, Concurrency, and Recovery

All SQL programs execute 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.

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 terminates, 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 a failure on the part of an application process. An application process itself, however, can explicitly request that its database changes be backed out. This operation is called *rollback*.

A *unit of recovery* (also known as a *logical unit of work*) is a recoverable sequence of operations within an application process. An application process represents a single unit of recovery, but may be broken down into many shorter units of recovery by means of commit or rollback operations. Thus, a unit of recovery is effectively begun by the initiation of an application process, or by the termination of a previous unit of recovery. A unit of recovery 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 recovery 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 by a rollback.

Locks acquired by the database manager on behalf of an application process are held until the termination of a unit of recovery. A lock explicitly acquired by a LOCK TABLE statement may be held past the termination of a unit of recovery if COMMIT HOLD or ROLLBACK HOLD is used to terminate the unit of recovery. A cursor may implicitly lock the row at which it is positioned. This lock will prevent another cursor in the same application process (or a DELETE or UPDATE statement not associated with that cursor) from acquiring a lock on the same row.

The initiation and termination of a unit of recovery 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 recovery, thereby making the changes available to other application processes. If a failure occurs before the unit of recovery 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 recovery was initiated.

Chapter 2. Language Elements

This chapter defines the basic syntax of SQL and language elements that are common to many SQL statements. Although examples are shown and most terms are defined before they are used, this chapter is not a tutorial. It is intended for those who require a definition of the following language elements:

Characters

The basic symbols of the language are characters from the EBCDIC collating sequence and code points. Characters are classified as letters, digits, or special characters. A *letter* is any one of the uppercase or lowercase characters A through Z plus the three characters reserved as alphabetic extenders for national languages (#, @, and $ in the United States). A *digit* is any one of the characters 0 through 9. A *special character* is any character other than a letter or a digit.

Tokens

The basic syntactical units of the language are called *tokens*. A token consists of one or more characters, excluding blanks and characters within a string constant or delimited identifier. (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.
- 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 109.

Spaces

A *space* is a sequence of one or more blank characters. Tokens, other than string constants, must not include a space. Any token may 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. The following examples illustrate the rule stated in this paragraph.

Here are some examples of ordinary tokens:

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

Here are some examples of combinations of the above ordinary tokens that, in effect, change the tokens:

```
1.1       .1+2       SELECTE       .1E       E3       SELECT1
```

This demonstrates why ordinary tokens must be followed by a delimiter token or a space.

Here are some examples of delimiter tokens:

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

Here are some examples of combinations of the above ordinary tokens and the above delimiter tokens that, in effect, change the tokens:

```
1.       .3
```

The dot (.) is a delimiter token when it is used as a separator in the qualification of names. Here the dot is used in combination with an ordinary token of a numeric constant. Thus, the syntax does not allow an ordinary token to be followed by a delimiter token. Instead, the ordinary token must be followed by a space.

If the system value QDECFMT is set to the value J, as described in "Decimal Point" on page 17, the comma is interpreted as a decimal point. Here are some examples of these numeric constants:

```
1,2       ,1       1,       1,e1
```

If the '1,1' or '1,e1' were meant to be two items, both the ordinary token (1) and the delimiter token (,) must be followed by a space, to prevent the comma from being interpreted as a decimal point. Although the comma is usually a delimiter token, the comma is part of the number when it is interpreted as a decimal point. Therefore, the syntax does not allow an ordinary token (1) to be followed by a delimiter token (,). Instead, an ordinary token must be followed by a space.

Uppercase and Lowercase

Ordinary tokens are folded to uppercase. Delimiter tokens are never folded to uppercase. Thus, the statement:

```
select * from CORPDATA.TEMPL where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM CORPDATA.TEMPL WHERE LASTNAME = 'Smith';
```

Identifiers

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

SQL Identifiers

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

- An *ordinary identifier* is a letter followed by zero or more characters, each of which is a letter, a digit, or the underscore character. Note that ordinary identifiers are converted to uppercase. An ordinary identifier must not be identical to a reserved word. (See Appendix C, "Reserved Words" on page 139 for a list of reserved words.)
- A *delimited identifier* is a sequence of one or more characters of the standard character set enclosed within SQL escape characters. Note that delimited identifiers are not converted to uppercase. The escape character is the quotation mark (") except for:
  - Dynamic SQL when the SQL string delimiter is set to the quotation mark. Here the SQL escape character is the apostrophe (').
  - COBOL application programs. A COBOL precompiler option specifies whether the escape character is the quotation mark (") or the apostrophe (').

  The following characters are not allowed within delimited identifiers:

  - A blank (X'40')
  - An asterisk (X'5C')
  - An apostrophe (X'7D')
  - A question mark (X'6F')
  - A quotation mark (X'7F')
  - X'00' through X'3F' and X'FF'

Identifiers are also classified according to their maximum length. A short identifier has a maximum length of 10 bytes. A long identifier has a maximum length of 18 bytes. Long identifiers are valid for *cursor-name* and *statement-name*.

For delimited identifiers, the bytes required for the escape characters are included in the length of the identifier, unless the characters within the delimiters would form an ordinary identifier.

For example, "PRIVILEGES" is in uppercase and the characters within the delimiters form an ordinary identifier; therefore, it has a length of 10 bytes and is a valid short identifier. On the other hand, "privileges" is in lowercase, has a length of 12 bytes, and is not a valid short identifier, because the bytes required for the delimiters must be included in the length of the identifier.

Examples: `WEEKLYSAL` `WEEKLY_SAL` `"WEEKLY.SAL"` `$500`

Host Identifiers

A *host-identifier* is a name declared in the host program. The rules for forming a *host-identifier* are the rules of the host language. For example, the rules for forming a *host-identifier* in a COBOL program are the same as the rules for forming a user-defined word in COBOL, except that *host-identifiers* must begin with a letter. Double-byte character set (DBCS) identifiers are not supported.

Naming Conventions

The rules for forming a name depend on the type of the object designated by the name. The syntax diagrams use different terms for different types of names. The following list defines these terms.

authorization-name A short identifier that designates a user. An *authorization-name* is a user profile name on the AS/400 system. An *authorization-name* containing a period (.) cannot be used as a qualifier unless it is enclosed in delimiters. SQL will use ten characters of the name, but only eight are allowed for the special register USER. If more than eight characters are found for USER, a negative value is returned in the SQLCODE field of the SQLCA.

collection-name A short identifier that designates a collection.

column-name A qualified or unqualified name that designates a column of a table or a view. The unqualified form of a column name is a short identifier. The qualified form is a qualifier followed by a period and a short identifier. The qualifier is a table name, a view name, or a correlation name.

Column names cannot be qualified with system names in the form *collection-name/table-name.column-name*, except in the COMMENT ON and LABEL ON statements. If column names need to be qualified, and correlation names are allowed in the statement, a correlation must be used to qualify the column. Column names can be SQL delimited identifiers, but the characters within the delimiters must not include special characters.

correlation-name A short identifier that designates a table, a view, or individual rows of a table or view.

cursor-name An long identifier that designates an SQL cursor.

descriptor-name A *host-identifier* that designates an SQL descriptor area (SQLDA). See "Host Variables" on page 21 for a description of a host identifier. A host variable that designates an SQL descriptor area must be of the form *:host-variable*. The form *:host-variable:indicator-variable* is not allowed.

host-label A token that designates a label in a host program.

host-variable A sequence of tokens that designates a host variable. A *host-variable* includes at least one *host-identifier*, as explained in "Host Variables" on page 21.

index-name

A qualified or unqualified name that designates an index. The unqualified form of an *index-name* is a short identifier. The qualified form of an *index-name* depends on whether the naming option (*SQL or *SYS) was specified on the STRSQL or CRTSQL*xxx* command (where *xxx* is RPG, CBL, C, or PLI).

For SQL names, the unqualified *index-name* in an SQL statement is implicitly qualified by the authorization ID of the statement. The qualified form is the *collection-name* followed by a period (.) and a short identifier.

For system names, the unqualified *index-name* in an SQL statement is implicitly qualified by *LIBL (user library list). The qualified form is a *collection-name* followed by a slash (/) and a short identifier.

statement-name

A long identifier that designates a prepared SQL statement.

table-name

A qualified or unqualified name that designates a table. The unqualified form of a *table-name* is a short identifier. The qualified form of a *table-name* depends on whether the naming option (*SQL or *SYS) was specified on the STRSQL or CRTSQL*xxx* command (where *xxx* is RPG, CBL, C, or PLI).

For SQL names, the unqualified *table-name* in an SQL statement is implicitly qualified by the authorization ID of the statement. The qualified form is the *collection-name* followed by a period (.) and a short identifier.

For system names, the unqualified *table-name* in an SQL statement is implicitly qualified by *LIBL (user library list). The qualified form is a *collection-name* followed by a slash (/) and a short identifier.

view-name

A qualified or unqualified name that designates a view. The unqualified form of a *view-name* is a short identifier. The qualified form of a *view-name* depends on whether the naming option (*SQL or *SYS) was specified on the STRSQL or CRTSQL*xxx* command (where *xxx* is RPG, CBL, C, or PLI).

For SQL names, the unqualified *view-name* in an SQL statement is implicitly qualified by the authorization ID of the statement. The qualified form is the *collection-name* followed by a period (.) and a short identifier.

For system names, the unqualified *view-name* in an SQL statement is implicitly qualified by *LIBL (user library list). The qualified form is a *collection-name* followed by a slash (/) and a short identifier.

SQL Names and System Names: Special Considerations

An override CL command (OVRDBF) may be specified that overrides an SQL or system name to another object name for data manipulation SQL statements. Overrides are ignored for data definition SQL statements. See *Programming: Data Management Guide* for more information about the override function.

You can access tables or views using either SQL names or system names. If you choose to use SQL names:

- If a qualified name is specified, SQL/400 attempts to find the object in the specified collection.
- If an object is unqualified, it is implicitly qualified by the authorization ID of the statement. Because the authorization ID can change based on user, most SQL syntax names should be qualified.

If you choose to use system names, the following rules apply:

- If a qualified name is specified, SQL/400 attempts to find the object in the specified library.
- If an unqualified object name is specified, SQL/400 searches the library list (*LIBL).

Authorization IDs

An authorization ID is a user profile. It is a character string of not more than 10 characters that designates a set of privileges.

The database manager uses authorization IDs to provide:

1. Authorization checking of SQL statements, and
2. Implicit qualifiers for the names of tables, views, and indexes.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID of the owner of the program. The authorization ID that applies to a dynamic SQL statement is the authorization ID of the user running the program. 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. For example, assume that SMITH is your user profile and you execute the following statement interactively:

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

SMITH is the run-time authorization ID, and the database manager therefore checks to ensure that SMITH is authorized to issue the statement. KEENE is the *authorization-name* specified in the statement. A group user profile may also be used when checking authority for an SQL statement. For information on group user profiles, see *Programming: Security Concepts and Planning*.

Examples: `NAME1` `SMITH.NAME1`

If you choose to use SQL names and SMITH is the authorization ID of the statement that contains NAME1, then NAME1 identifies the same object as SMITH.NAME1. Otherwise NAME1 and SMITH.NAME1 identify different objects.

Data Types

For information about specifying the data types of columns, see "CREATE TABLE" on page 65.

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, and special registers.

The basic data types are character string, integer, floating-point, numeric, and decimal. Floating-point values are further classified as single precision and double precision, while integers are further classified as small integer and large integer. Integers may be specified in some host variables as having precision and scale.

All data types include the null value. The null value is a special value that is distinct from all nonnull values and thereby denotes the absence of a (nonnull) value. In SQL/400, a column of a table cannot contain a null value.

Character Strings

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

Fixed-Length 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 32766 inclusive.

String Variables

Fixed-length string variables can be defined in all host languages. Varying-length string variables can be defined in all host languages except RPG. Null-terminated string variables can be defined in host language C.

Mixed Data in Character Strings

Character strings may contain sequences of double-byte characters, each sequence preceded by a "shift-out" character and followed by a "shift-in" character. A string containing one or more such sequences is called "mixed." The principal use of mixed data is to represent national language texts.

SQL does not recognize subclasses of double-byte characters, and does not assign any specific meaning to particular double-byte codes. However, if you choose to use mixed data, then two single-byte EBCDIC codes are given special meanings:

- X'0E', the "shift-out" character, is used to mark the beginning of a sequence of double-byte codes.
- X'0F', the "shift-in" character, is used to mark the end of a sequence of double-byte codes.

In order for SQL/400 to recognize double-byte characters in a mixed string, the following condition must be met:

- 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 invalid. 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 may be more than one set of paired shift-out and shift-in characters in the string.

The length of a mixed 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.

When the system value QIGC indicates that DBCS is allowed, CREATE TABLE will create character columns as OPEN fields, unless FOR BIT DATA or FOR SBCS is specified. The SQL user will see these as character fields, but the system database support will see them as DBCS-Open fields. For a definition of the DBCS-Open field, see *Programming: Data Description Specifications Reference*.

Numbers

You can define small integer and large integer variables in all languages. Decimal and numeric variables can be defined in PL/I, COBOL, and RPG. Floating-point variables can be defined in PL/I and C.

All numbers have a sign and a precision. The precision is the total number of binary or decimal digits excluding the sign. The sign is positive if the value is zero.

Small Integer

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

For small integers, precision and scale are supported by AS/400 system host variables in languages and by AS/400 system physical and logical files. For information concerning the precision and scale of binary integers, see *Programming: Data Description Specifications Reference*.

Large Integer

A *large integer* is a binary integer with a precision of 31 bits. The range of large integers is -2147483648 to +2147483647.

For large integers, precision and scale are supported by AS/400 system host variables in languages and by AS/400 system physical and logical files. For information concerning the precision and scale of binary integers, see *Programming: Data Description Specifications Reference*.

Single Precision Floating-Point

A *single precision floating-point* number is an IEEE short (32 bits) floating-point number. The range of magnitude is approximately $1.17549436 \times 10^{-38}$ to $3.40282356 \times 10^{38}$.

Double Precision Floating-Point

A *double precision floating-point* number is 64 bits long. The range of magnitude is approximately 2.2250738585072014 $\times 10^{-308}$ to 1.7976931348623158 $\times 10^{308}$.

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, cannot be negative or 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 $-10^{31}+1$ to $10^{31}-1$.

Numeric

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

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

Basic Operations

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, and SELECT INTO 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 numbers and strings are not compatible. Thus, numbers and strings cannot be compared, numbers cannot be assigned to string columns or variables, and strings cannot be assigned to numeric columns or variables.

For assignment operations, a null value cannot be assigned to a column, nor to a host variable that does not have an associated indicator variable. (See "Host Variables" on page 21 for a discussion of indicator variables.)

Numeric Assignments

The basic rule for numeric assignments is that the whole part of a number is never truncated. If the whole part *is* truncated, a negative value is returned in the SQLCODE field of the SQLCA. If the target of the assignment cannot contain the entire fractional part of a number, the fractional part of the number is truncated.

Decimal, Numeric, or Integer to Floating-Point

Floating-point numbers are approximations of real numbers. Hence, when a decimal, numeric, 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 will be more accurate if the receiving column or variable is defined as double precision rather than single precision.

Decimal, Numeric, or Floating-Point to Integer

When a decimal, numeric, or floating-point number is assigned to a binary integer column or variable, the number is converted, if necessary, to the precision and the scale of the target. If the scale of the target is zero, the fractional part of the number is lost. 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.

Decimal or Numeric to Decimal or Numeric

When a decimal or numeric number is assigned to a decimal or numeric 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 or Numeric

When an integer is assigned to a decimal or numeric 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. If the scale of the integer is zero, the precision of the temporary decimal number is 5,0 for a small integer, or 11,0 for a large integer.

Floating-Point to Decimal or Numeric

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

To COBOL Integers

Assignment to integer host variables takes into account any scale specified for the host variable. However, assignment to integer host variables uses the full size of the integer. Thus, the value placed in the data item may be larger than the maximum precision specified for the host variable.

For example, if column 1 contains a value of 12345, the COBOL statements:

```
01  A  PIC  S9999  COMP-4.
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 basic 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.)

When a string is assigned to a fixed-length string column or 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 EBCDIC or double-byte blanks.

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.

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. When a string is assigned to a column and the string is longer than the length attribute of that column, an error occurs. For a description of the SQLCA, see "SQL Communication Area (SQLCA)" on page 127.

If the string contains mixed data, the assignment rules may require truncation within a sequence of double-byte codes. To prevent the loss of the shift-in character that ends the double-byte sequence, additional characters may be cut from the end of the string; then a shift-in character is appended before the assignment is made. In the truncated result, there is always an even number of bytes between each shift-out character and its matching shift-in character.

Numeric Comparisons

Numbers are compared algebraically; that is, with regard to sign. −2, for example, is less than +1. Conversion for the comparison is handled internally, and packed decimal is used if the numbers are any combination of decimal and numeric numbers.

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

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended

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

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

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

String Comparisons

The comparison of two strings is determined by the comparison of the corresponding bytes of each string. The strings must not be longer than 32,766 bytes. 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.

Two strings are equal if they are both empty or if all corresponding bytes are equal. Varying-length strings that differ only in the number of trailing blanks are considered equal. 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.

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the attribute NOT NULL. A negative 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 may include a sign and a decimal point; the second number may 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 24. 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 3. 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) rounded to the next highest odd integer; the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples: 25.5 1000. -15. +37589.3333333333

Character String Constants

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 32765. Two consecutive string delimiters are used to represent one string delimiter within the character string.
- An X followed by a sequence of characters that starts and ends with a string delimiter. The characters between the string delimiters must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 32764. A hexadecimal digit is a digit or any of the letters A through F (upper or lower case). Under the conventions of hexadecimal notation, each pair of hexadecimal digits represents a character. This form of string constant allows you to specify characters that do not have a keyboard representation.

Examples:

```
'12/14/1985'
'32'
'DON''T CHANGE'
''
X'FFFF'
```

Alternative Syntax

Decimal Point

You have the option of specifying whether the decimal point in a numeric constant is represented by a period or a comma. The default value for Interactive SQL is indicated by the system value QDECFMT. This value can be set through the CL command CHGSYSVAL. For information on this command, see *Programming: Control Language Reference*.

*PERIOD, *COMMA, and *SYSVAL are mutually exclusive COBOL and RPG precompiler options that specify the character that represents the decimal point in SQL statements embedded in the program. If *PERIOD is specified, the decimal point is the period; if *COMMA is specified, the decimal point is the

comma; if *SYSVAL is specified, the decimal point is determined by the system value QDECFMT.

In PL/I and in C, the usage is fixed. The decimal point is always the period.

Delimiters

*APOST and *QUOTE are mutually exclusive COBOL precompiler options that name the string delimiter within COBOL statements. *APOST names the apostrophe (') as the string delimiter; *QUOTE names the quotation mark ("). *APOSTSQL and *QUOTESQL are mutually exclusive COBOL precompiler options that play a similar role for SQL statements embedded in COBOL programs. *APOSTSQL names the apostrophe (') as the SQL string delimiter; with this option, the quotation mark (") is the SQL escape character. *QUOTESQL names the quotation mark as the SQL string delimiter; with this option, the apostrophe is the SQL escape character. The values of *APOSTSQL and *QUOTESQL are respectively the same as the values of *APOST and *QUOTE.

In host languages other than COBOL, the usages are fixed. The string delimiter for the host language and for static SQL statements is the apostrophe ('); the SQL escape character is the quotation mark (").

Special Registers

A *special register* is a storage area whose primary use is to store information produced with the use of specific features of the database manager.

USER

The USER special register is 8 characters in length; it specifies the run-time authorization ID. Thus, if you execute SQL statements interactively, USER specifies your user profile name. USER 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. The value in USER cannot be changed.

Example:

```
SELECT * FROM SYSIBM.SYSTABLES
  WHERE CREATOR = USER
```

Column Names

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

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In a *column function*, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. (Groups and intermediate result tables are explained under "SELECT INTO" on page 117.) 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

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

- In some forms of the COMMENT ON and LABEL ON statements, a column name *must* be qualified. This is shown in the syntax diagrams.
- Where the column name specifies values of the column, it *may* be qualified at the user's option.
- In all other contexts, a column name *must not* be qualified.

Where a qualifier is optional, it can serve as a correlation, as described under "Column Name Qualifiers to Avoid Ambiguity" on page 20.

Correlation Names

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

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

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.

If a correlation name is not specified, a name that is the same as the table or view name is implicitly assigned. If SQL naming is specified, the implicit correlation name is the qualified table name after any implicit qualification. If system naming is specified and the table name is qualified, the implicit correlation name is the table name portion of the qualified name. No two correlation names, whether implicitly or explicitly assigned, may be the same. Thus, while a correlation name may be the same as the name of another table, the other table cannot be referenced in the same statement unless a different correlation name has been assigned to it.

A correlation name can be used as a qualifier to avoid ambiguity or to establish a correlated reference. It can also be used 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.

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 partial statement:

```
SELECT Z.CODE, MYTABLE.CODE
  FROM MYTABLE Z, MYTABLE
  WHERE ...
```

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

1. A name that follows a table or view name is both a correlation name and a table designator. Thus, Z is a table designator, and qualifies the first column name after SELECT.
2. A table name or view name that is *not* followed by a correlation name is a table designator. Thus, MYTABLE is a designator and qualifies the second column name after SELECT.

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.

Two or more object tables can be instances of the same table. In this case, distinct correlation names must be used to unambiguously designate the particular instances of the table.

In the following FROM clause, for example, X and Y are defined to refer, respectively, to the first and second instances of the table TEMPL.

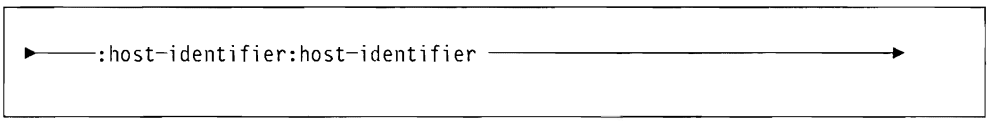
```
FROM CORPDATA.EMP X, CORPDATA.EMP Y
```

Host Variables

A *host variable* is a PL/I, C, or RPG variable, or a COBOL group data item, that is referenced in an SQL statement. Host variables can only be referenced in static SQL statements. Host variables should not begin with the characters 'SQL', 'sql', or 'RDI'. Host variables are defined by statements of the host language.

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

The general form of a *host-variable* reference is:



The variable designated by the second *host-identifier* must have a data type of *small integer* with zero scale. One purpose of the indicator variable is to specify the null value. A negative value in the indicator variable specifies the null value.

For example, if :V1:V2 is specified in a FETCH or SELECT INTO statement, and if the value returned is null, V1 is not changed, and V2 is set to a negative value, either to -1 if the value selected was the null value, or to -2 if the null value was returned because of numeric conversion errors or arithmetic expression errors met in the SELECT list of an outer SELECT statement. If the value returned is not null, that value is assigned to V1, and V2 is set to zero (unless the assignment to V1 requires string truncation in which case V2 is set to the original length of the string). If an indicator variable used in other than a FETCH statement or an INTO clause contains a negative value, a negative value is returned in the SQLCODE field of the SQLCA.

Another form of *host-variable* reference is:

```
:host-identifier
```

If this form is used, 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 null values cannot be assigned to the variable. Thus this form should not be used in an INTO clause unless the corresponding column cannot contain null values.

If a null value is returned, and you have not provided an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. If your data is truncated and there is no indicator variable, no error condition results.

A host variable must always be preceded by a colon when it is used in an SQL statement.

In PL/I and C, 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, this rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

Host Structures in PL/I, C, and COBOL

A host structure is a PL/I structure, C structure, or COBOL group that is referenced in an SQL statement. Host structures are defined by statements of the host language.

Host Structures in COBOL, PL/I, C, and RPG

A host structure is a COBOL group, PL/I, or C structure, or RPG data structure that is referenced in an SQL statement. Host structures are defined by statements of the host language, as explained in the *Programming: Structured Query Language/400 Programmer's Guide*. 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:S2 is a host structure reference if S1 designates a host structure. If S2 designates a host structure, it must be defined as a vector of small integer variables. S1 is the main structure and S2 is its indicator structure.

A host structure may be referenced in any context where a list of host variables may be referenced. A host structure reference is equivalent to a reference to each of the host variables contained within the structure in the order which they are defined in the host language structure declaration. The *n*th variable of the indicator structure is the indicator variable for the *n*th variable of the main structure.

In COBOL, for example, if V1, V2, and V3 are declared as variables within the structure S1, the statement:

```
EXEC SQL FETCH CURSOR1 INTO :S1 END-EXEC.
```

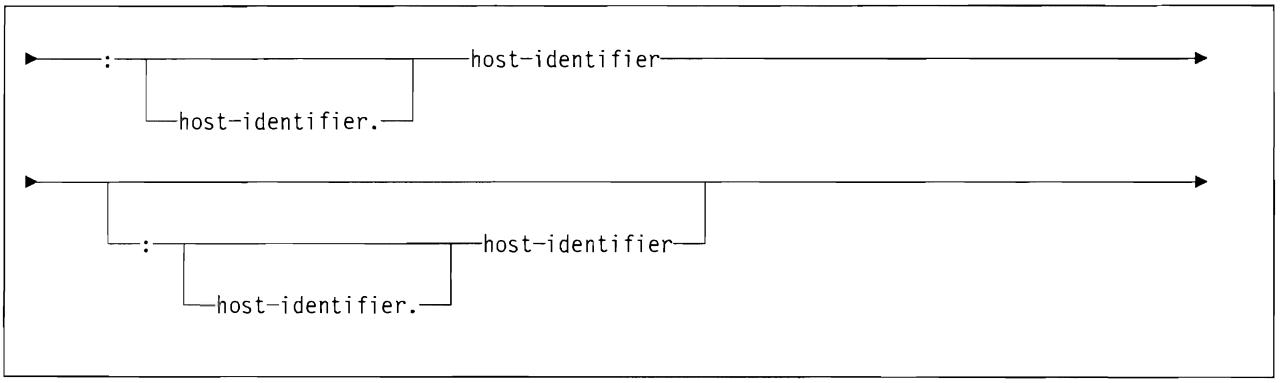
is equivalent to:

```
EXEC SQL FETCH CURSOR1 INTO :V1, :V2, :V3 END-EXEC.
```

If the main structure has *m* more variables than the indicator structure, the last *m* variables of the main structure do not have indicator variables. If the main structure has *m* less variables than the indicator structure, the last *m* variables of the indicator structure are ignored. These rules also apply if a reference to a host structure includes an indicator variable or if a reference to a host variable includes an indicator structure. If an indicator structure or variable is not specified, no variable of the main structure has an indicator variable.

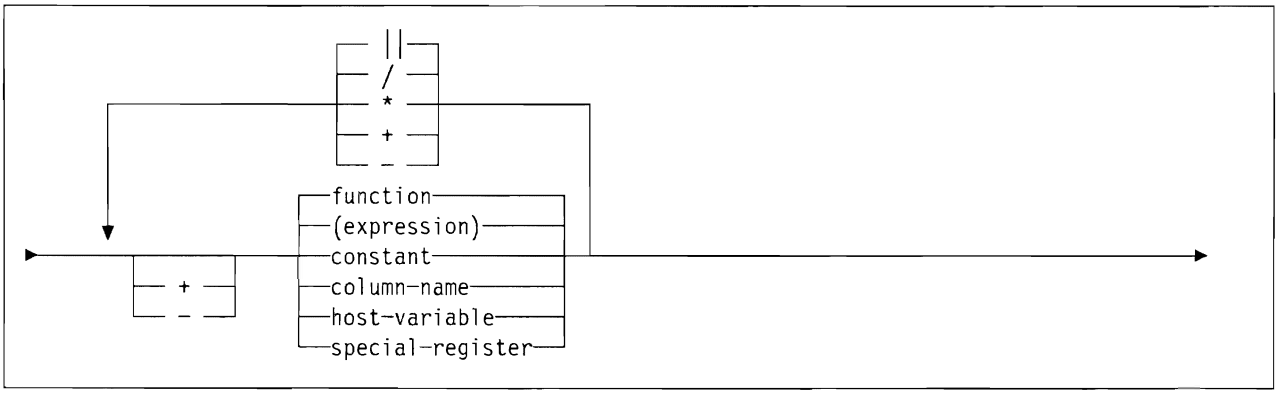
In addition to structure references, individual host variables or indicator variables in PL/I, C, and COBOL may be referenced by qualified names. The qualified form is a host identifier followed by a period and another host identifier. The first host identifier must designate a structure, and the second host identifier must designate a host variable within that structure.

The following diagram specifies the syntax of references to host variables and host structures:



Expressions

An expression specifies a value. The form of an expression is as follows:



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

If the concatenation operator (||) is used, the result of the expression is a string. The operands of concatenation must both be the result of an expression, and both must be character strings. The sum of their lengths must not exceed 32,766.

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 (X'0F'), while the second operand is a character string beginning with a "shift-out" character (X'0E'), these two bytes are eliminated from the result.

Example: FIRSTNME||' '||LASTNAME

With Arithmetic Operators

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 invalid.

The prefix operator + (*unary plus*) does not change its operand. The prefix operator - (*unary minus*) reverses the sign of a nonzero operand; and if the data type of A is *small integer*, then the data type of -A is *large integer*. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, -, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero.

Two Integer Operands

If both operands of an arithmetic operator are integers with zero scale, 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. If either integer operand has nonzero scale, it is converted to a decimal operand with the same precision and scale.

Integer and Decimal or Numeric Operands

If one operand is an integer with zero scale and the other is decimal or numeric, the operation is performed in decimal using a temporary copy of the integer which 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 (including leading zeros)	same as the number of digits in the constant

If one operand is an integer with nonzero scale, it is first converted to a decimal operand with the same precision and scale.

Two Decimal or Numeric Operands

If both operands are decimal or numeric, 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.

The result of a decimal operation must not have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from a temporary result which may have a precision greater than 31. If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result. If the precision of the temporary result is greater than 31, the final result is derived from the temporary result by the elimination of leading zeros so the final result has a precision of 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.

Floating-Point Operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point, the operands having first been converted to double precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double precision floating-point. An operation involving a floating-point number and a decimal or numeric number is performed with a temporary copy of the decimal or numeric number which has been converted to double precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

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: `1.10 * (SALARY + BONUS)`

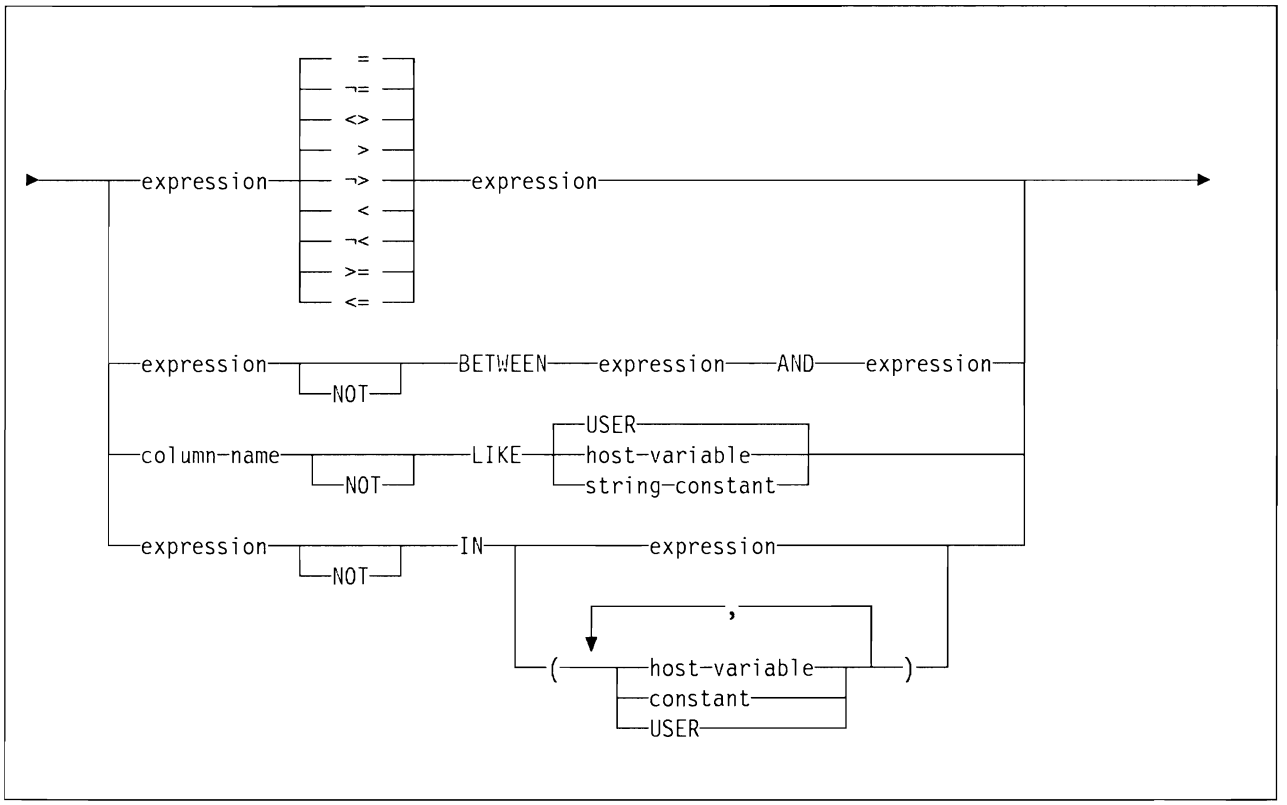
Host Variables

A host variable in an expression must identify a host variable (not a structure) described in the program according to the rules for declaring host variables. For further information about declaring host variables, see *Programming: Structured Query Language/400 Programmer's Guide*.

Predicates

A *predicate* specifies a condition that is "true," "false," or "unknown" about a given row or group.

The general form of a predicate is as follows:



All values specified in a predicate must be compatible. The value of a host variable must not be a string longer than 32766 bytes. The value of a host variable must not be null (that is, the variable may not have a negative indicator variable).

A view column referenced in a predicate must not be derived from a column function.

Basic Predicate

A *basic predicate* compares two values. The format of a basic predicate is an expression followed by a comparison operator and another expression.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

For values *x* and *y*:

Predicate	Is True If and Only If...
x = *y*	*x* is equal to *y*
x¬= *y*	*x* is not equal to *y*
x<> *y*	*x* is 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*
x¬< *y*	*x* is not less than *y*
x¬> *y*	*x* is not greater than *y*

Examples:

```
EMPNO = '528671'
SALARY < 20000
PRSTAFF<>:VAR1
```

BETWEEN Predicate

The BETWEEN predicate compares a value with a range of values. The format of a BETWEEN predicate is as follows:

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

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

The BETWEEN predicate:

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

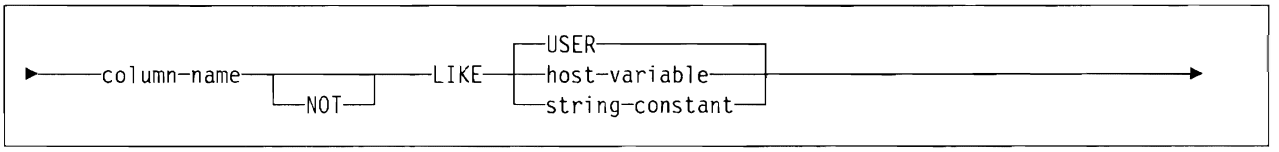
is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,
value1 < value2 or value1 > value3.
```

Example: `SALARY BETWEEN 20000 AND 40000`

LIKE Predicate

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



The *column-name* must identify a string column. If a host variable is specified, it must identify a character variable (not a structure) that is described in the program under the rules for declaring string host variables; it cannot have an indicator variable. The terms "character", "percent sign", and "underscore" in the following discussion refer to EBCDIC characters. The following description is intended for those who require a rigorous definition. The description uses *x* to denote a value of the column and *y* to 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 nonempty sequence of characters other than an underscore or a percent sign.

The result of the predicate is either true or false. The result is true 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.

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

With Mixed Data: If the column identified by *column-name* allows mixed data, the column may contain double-byte characters, as may the host variable or string constant. In that case the special characters in *y* are interpreted as follows:

- A EBCDIC underscore refers to one EBCDIC character; a double-byte underscore refers to one double-byte character.
- A percent sign, either EBCDIC or double-byte, refers to any number of characters of any type, either EBCDIC or double-byte.

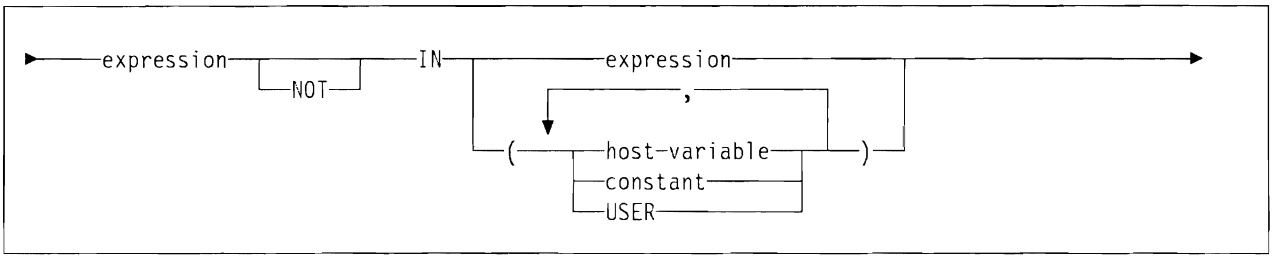
Examples:

```
NAME LIKE '%SMITH%'
STATUS LIKE 'N_'
```

The first example is true if 'SMITH' appears anywhere within NAME. The second example is true if the value of STATUS has a length of two and the first character is 'N'.

IN Predicate

The IN predicate compares a value with a collection of values. The format of the IN predicate is as follows:



Each host variable specified must identify a structure or variable that is described in the program under the rules for declaring host structures and variables.

An IN predicate of the form:

```
expression IN expression
```

is equivalent to a basic predicate of the form:

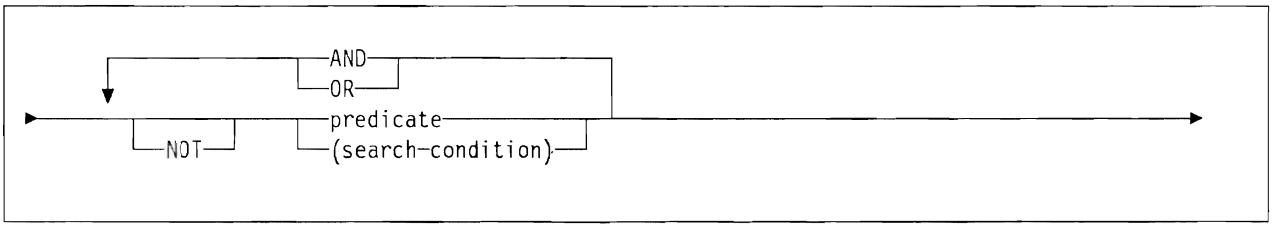
```
expression = expression
```

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

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

Search Conditions

A *search condition* specifies a condition that is "true," "false," or "unknown" about a given row or group. The form of a search condition is as follows:



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.

Examples: `SALARY > 2000` `NAME LIKE :VAR4` `AVG(SALARY) < 30000`

AND and OR are defined in Table 1 on page 30, in which P and Q are any predicates:

Table 1. Truth Tables for AND and OR			
P	**Q**	**P AND Q**	**P OR Q**
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

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

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

Example: `MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03')`

Chapter 3. Functions

A *function* is an operation denoted by a function name followed by a pair of parentheses enclosing the specification of one or more operands. 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 the application of the function to the result of the expression.

Functions are classified as *scalar functions* or *column functions*. The argument of a column function is a collection of values. An argument of a scalar function is a single value.

In the syntax of SQL, the only use of the term "function" is 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 below and in Chapter 4, "Queries" on page 39.

Column Functions

The argument of COUNT(*) is a group or an intermediate result table as explained in Chapter 4, "Queries" on page 39. The following applies to all column functions other than COUNT(*).

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 4, "Queries" on page 39. For example, the result of the following SELECT statement is the number of employees in department D01:

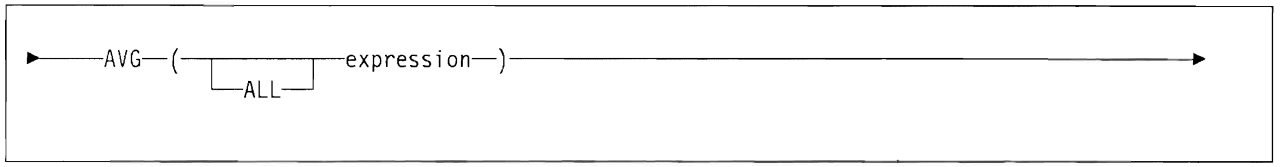
```
SELECT COUNT(*)
  FROM CORPDATA.EMP
  WHERE WORKDEPT = 'D01'
```

The values of the argument are specified by an expression. This expression must not include a column function, and must include at least one *column-name*.

Following, in alphabetical order, is a definition of each of the column functions.

AVG

The AVG function returns the average of a set of numbers. The form of the function is:



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

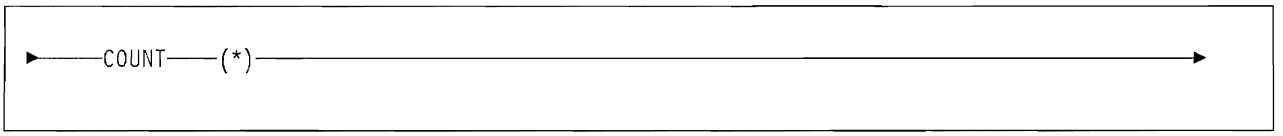
The data type of the result is the same as the data type of the argument values, except that the result is double precision floating-point if the argument values are single precision floating-point, and the result is decimal if the argument values are nonzero scale binary. If the data type of the argument values is decimal, numeric, or nonzero scale binary with precision *p* and scale *s*, the precision of the result is 31 and the scale is 31-*p*+*s*. The result can be null.

The function is applied to the set of values derived from the argument values. If this set is empty, the result of the function is the null value. Otherwise, the result is the average value of the set.

Example: `AVG(SALARY)`

COUNT

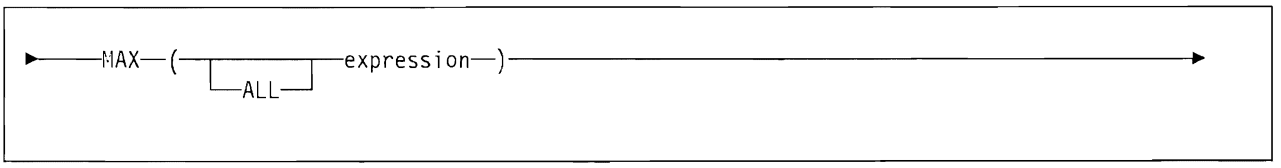
The COUNT function returns the number of rows or values in a set of rows or values. The form of the function is:



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

MAX

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



The argument values can be any values other than character strings whose maximum lengths are greater than 256.

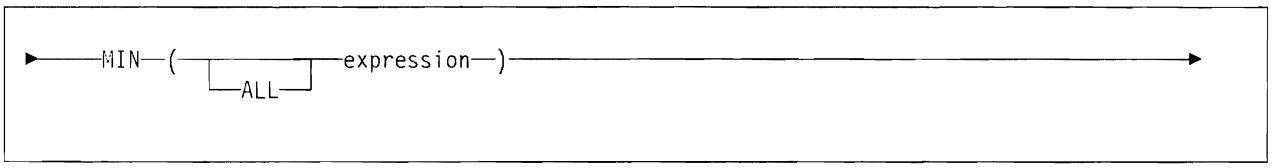
The data type and length attribute of the result are the same as the data type and length attribute of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values. If this set is empty, the result of the function is the null value. Otherwise, the result is the maximum value in the set.

Example: MAX(SALARY)

MIN

The MIN function returns the minimum value in a set of values. The form of the function is:



The argument values can be any values other than character strings whose maximum lengths are greater than 256.

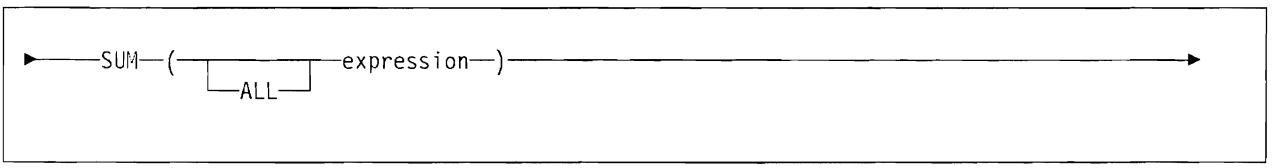
The data type and length attribute of the result are the same as the data type and length attribute of the argument values. The result can be null.

The function is applied to the set of values derived from the argument values. If this set is empty, the result of the function is the null value. Otherwise, the result is the minimum value in the set.

Example: MIN(SALARY)

SUM

The SUM function returns the sum of a set of numbers. The form of the function is:



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 double precision floating-point if the argument values are single precision floating-point, large integers if the argument values are small integers, and decimal if the argument values are nonzero scale binary. If the data type of the argument values is numeric, decimal, or nonzero scale binary, 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. If this set is empty, the result of the function is the null value. Otherwise, the result is the sum of the values in the set.

Example: SUM(SALARY)

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. 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.

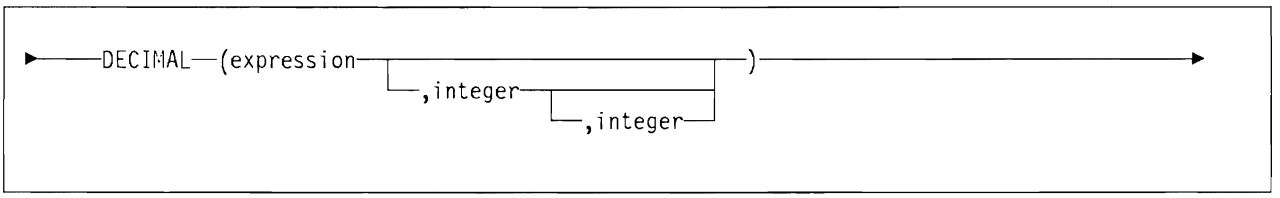
The restrictions on the use of column functions do not apply to scalar functions because a scalar function is applied to a single value rather than a collection of values. For example, the result of the following SELECT statement has as many rows as there are employees in department D01:

```
SELECT EMPNO, LASTNAME, YEAR(CURRENT DATE - BRTHDATE)
  FROM CORPDATA.TEMPL
  WHERE WORKDEPT = 'D01'
```

Following in alphabetical order is the definition of each of the scalar functions.

DECIMAL

The DECIMAL function returns a packed decimal representation of a numeric value. The form of the function is:



The first argument must be a number. The second argument, if specified, must be in the range of 1 to 31. The third argument, if specified, must be in the range of 0 to *p*, where p is the second argument. Omission of the third argument is an implicit specification of zero.

The default for the second argument depends on the data type of the first argument:

- 15 for floating-point, decimal, numeric, or nonzero scale binary
- 11 for large integer
- 5 for small integer

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.

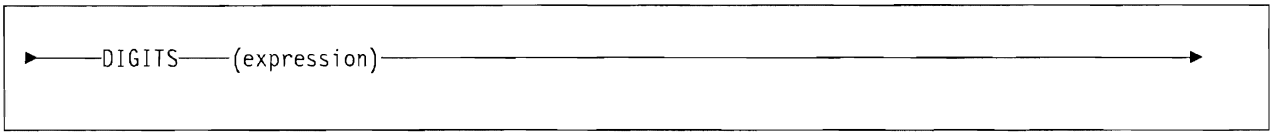
The result is the same number that would occur if the first argument were assigned to a decimal column or variable with a precision of *p* and a scale of *s*.

An error occurs if the number of significant decimal digits required to represent the whole part of the number is greater than *p-s*.

Example: `DECIMAL(AVG(SALARY),8,2)`

DIGITS

The DIGITS function returns a character string representation of a number. The form of the function is:



The argument must be an integer, decimal, or numeric value.

The result of the function is a fixed-length character string.

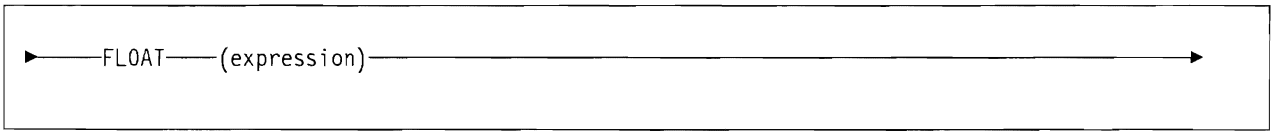
The result is a string of digits that represents the absolute value of the argument without regard to its scale. Thus, the result does not include a sign or a decimal point. The result includes any necessary leading zeros so that the length of the string is:

- 5 if the argument is a small zero scale integer
- 10 if the argument is a large zero scale integer
- *p* if the argument is a decimal, numeric, or nonzero scale integer with a precision of *p*.

Example: `DIGITS(JOBCODE) = '6'`

FLOAT

The FLOAT function returns a floating-point representation of a number. The form of the function is:



The argument must be a number.

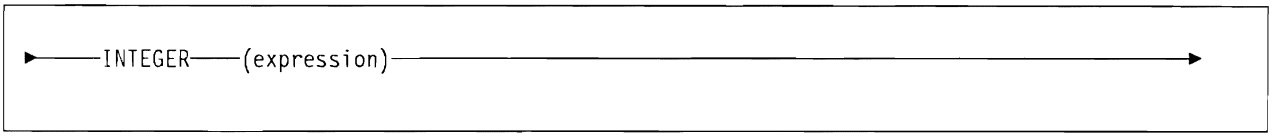
The result of the function is a double precision floating-point number.

The result is the same number that would occur if the argument were assigned to a double precision floating-point column or variable.

Example: `FLOAT(ACSTAFF)/2`

INTEGER

The INTEGER function returns an integer representation of a number. The form of the function is:



The argument must be a number.

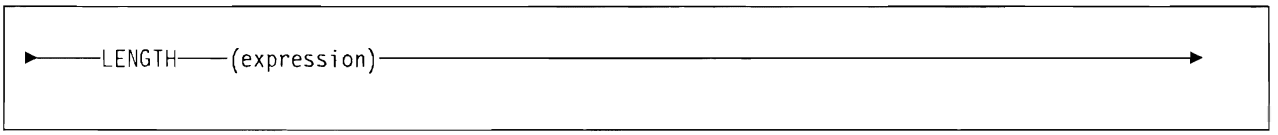
The result of the function is a large integer with zero scale.

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, a negative value is returned in the SQLCODE field of the SQLCA.

Example: `INTEGER(SUM(EMPTIME)+.5)`

LENGTH

The LENGTH function returns the length of a value. The form of the function is:



The argument can be any value.

The result of the function is a large integer with zero scale.

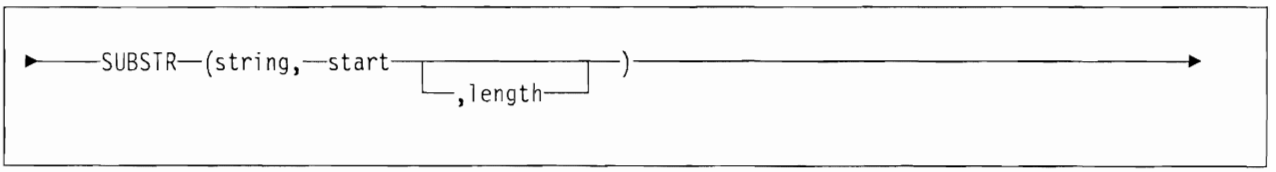
The result is the length of the argument. The length is the number of bytes used to represent the value:

- the length of the string for character strings
- 2 for small integer
- 4 for large integer
- 4 for single precision floating-point
- 8 for double precision floating-point
- INTEGER(*p*/2) + 1 for decimal numbers with precision *p*.
- *p* for numeric numbers with precision *p*

Example: `LENGTH(STRING)-:N`

SUBSTR

The SUBSTR function returns a substring of a string. The form of the function is:



string
The first argument must be a string expression.

A substring of *string* is one or more contiguous characters of *string*. The SUBSTR function does not recognize mixed data, so if *string* contains mixed data, the result may not be a well-formed mixed data string.

start
start must be an integer between 1 and the length of *string*. *start* specifies the first character of the result.

length
length specifies the length of the result. If specified, *length* must be an integer in the range 1 to *n*, where *n* is the length attribute of *string* - *start* + 1. Omission of *length* is an implicit specification of LENGTH(*string*) - *start* + 1. The default for *length* is the number of characters from the character specified by *start* to the last character of *string*.

Example: `SUBSTR(FIRSTNAME,1,1)`

Chapter 4. Queries

A *query* specifies a result table or intermediate result table.

In a program, a query is a component of other SQL statements. The three forms of a query described in this chapter are:

- The subselect,
- The fullselect, and
- The *select-statement*

Note that there is another form of select, described under "SELECT INTO" on page 117.

Note: Where the syntax outlined in these descriptions is specifically limited to a *column-name*, (rather than to an *expression*), the column you identify must not be a column of a view derived from an expression, function, or constant.

Authorization

For any form of a query, the privileges held by the authorization ID of the statement must include the SELECT privilege on every table and view identified in the statement.

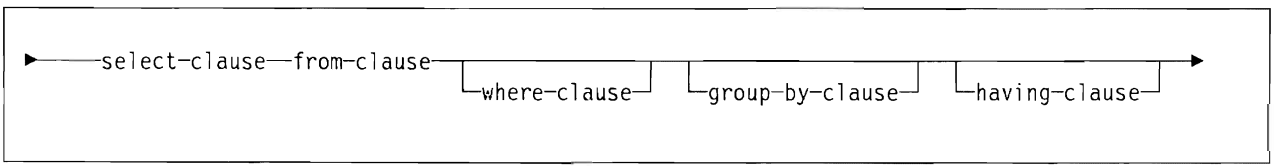
You have the SELECT privilege on a table if any of the following apply:

- You are the owner of the table.
- You have been granted the SELECT privilege on the table.
- You have been granted the system authorities *OBJOPR and *READ on the table.

You have the SELECT privilege on a view if any of the following apply:

- You have been granted the SELECT privilege on the view.
- You created the view, you had the SELECT privilege on its base table when the view was created, and you still have that SELECT privilege.
- You have been granted the system authority *OBJOPR on the view and the system authority *READ on the base table.

subselect



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.

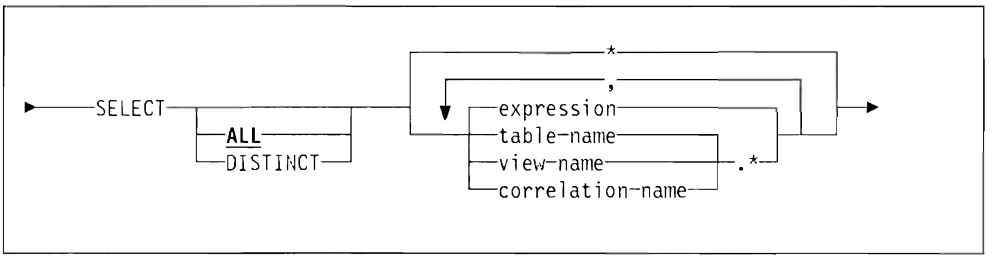
A subselect specifies a result table derived from the tables or views identified in the FROM clause. The derivation can be described as a sequence of operations in which the result of each operation is input for the next. (This is

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



Produces a final result table by selecting only the columns indicated by the *select list* from R, where R is the result of the previous operation.

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.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value of the second.

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 is established when the program is prepared and does not represent any columns that have been added to the table later.

expression
May be any expression of the type described in Chapter 2, but commonly the expressions used include column names. Each column name used in the select list must unambiguously identify a column of R.

*name.**
Represents a list of names that identify the columns of *name*. *name* may be a table name, view name, or correlation name, and must designate a table or view named in the FROM clause, but must not be of the form *data-base/table-name*. The first name in the list identifies the first column of *name*, the second name identifies the second column, and so forth. The list is established when the program is prepared and does not represent any columns that have been added to the table later.

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 at prepare time), and may not exceed 8000.

Other Limitations: The select list must not include column functions if R is derived from a view whose subselect includes DISTINCT, GROUP BY, or HAVING. Furthermore, if R is derived from a view whose subselect includes DISTINCT, the select list must identify all columns of the view (possibly by SELECT *) and must not include DISTINCT or arithmetic expressions.

Applying the Select List: Some of the results of applying the select list to R depend on whether or not GROUP BY or HAVING is used. Those results are described separately.

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 does not include column functions, then the select list is applied to each row of R and the result contains as many rows as there are rows in R.
- If the select list is a list of column functions, then R is the source of the arguments of the functions and the result of applying the select list is one row.

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.

In either case the *n*th column of the result contains the values specified by applying the *n*th expression in the operational form of the select list.

Null attributes of result columns: Result columns do not allow null values if they are derived from:

- A column
- A constant
- The COUNT function
- A host variable
- A scalar function or expression that does not allow null values.

Result columns do allow null values if they are derived from:

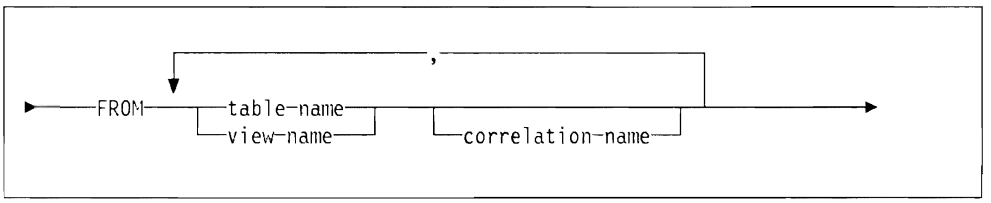
- Any column function but COUNT
- An arithmetic expression
- A scalar function that allows null values.

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 types of result columns: Each column of the result of SELECT acquires a data type from the expression from which it is derived.

When the expression is ...	The data type of the result column is ...
the name of any numeric column	the same as the data type of the column, with the same precision and scale for decimal, numeric, or binary columns.
an integer constant	INTEGER
a decimal or floating-point constant	the same as the data type of the constant. For a decimal constant, the precision and scale of the result are the same as the constant. For a floating-point constant, 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, numeric, or binary variables.
an arithmetic expression	the same as the data type of the result, with the same precision and scale for numeric results as described under "Expressions" on page 23.
any function	(see Chapter 3 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 variable	the same as the data type of the variable, with a length attribute equal to the length of the variable.
a character string constant of length *n*	variable length of length *n*.

from-clause



Names a single table or view, or produces an intermediate result table. The intermediate result table contains all possible combinations of the rows of the named 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 list of names in the FROM clause must conform to these rules:

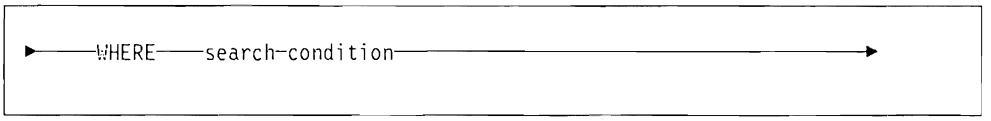
- Each *table-name* and *view-name* must name a table or view described in the collection.

- If the FROM clause specifies a view that contains a GROUP BY, HAVING, or DISTINCT clause, no other tables or views can be specified in that FROM clause.

The FROM clause also defines the meaning of correlation names. A *correlation-name* applies to the table or view named by the immediately preceding *table-name* or *view-name*. If a correlation name is specified, then that correlation name must be used elsewhere in the subselect statement to designate that table or view. For rules governing the use of correlation names, see "Qualified Column Names" on page 19.

Each correlation name specified in the same FROM clause must be unique and must not be the same as a table name or view name specified in the clause. When the same table name or view name is specified more than once in a FROM clause, a correlation name must be specified after each occurrence of the replicated 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.

where-clause

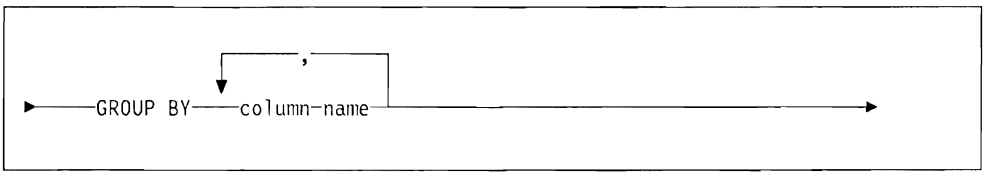


Produces an intermediate result table by applying *search-condition* to each row of R, where R is the result of the FROM clause. The result table contains the rows of R for which the search condition is true.

search-condition describes a search condition that conforms to these rules:

- The condition is formed as described in Chapter 2.
- Each *column-name* in the search condition unambiguously identifies a column of R.
- A *column-name* in the search condition does not identify a column that is derived from a column function. (A column of a view can be derived from a column function.)

group-by-clause



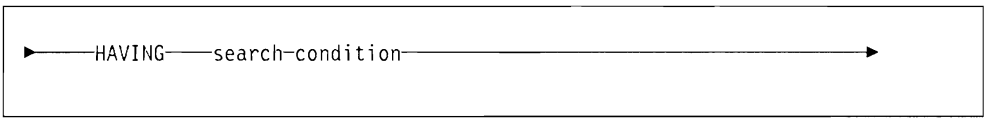
Produces an intermediate result table by grouping the rows of R, where R is the result of the previous clause.

column-name unambiguously names a column of R. Each column named is called a *grouping column*.

The result of GROUP BY is a set of groups of rows. 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.

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.

having-clause



Produces an intermediate result table by applying *search-condition* to each group of R where R is the result of the previous clause. If that clause is not GROUP BY, all rows of R are considered as one group. The result table contains those groups of R for which the search condition is true.

The *search-condition* describes a search condition that conforms to these rules:

- The condition is formed as described in Chapter 2.
- Each *column-name* in the *search-condition* must do one of the following:
  - Unambiguously identify a grouping column of R
  - Be specified within a column function.[1]

A group of R to which the search condition is applied supplies the argument for each function in the search condition.

Examples of a subselect

Example 1: Show all rows of CORPDATA.EMP

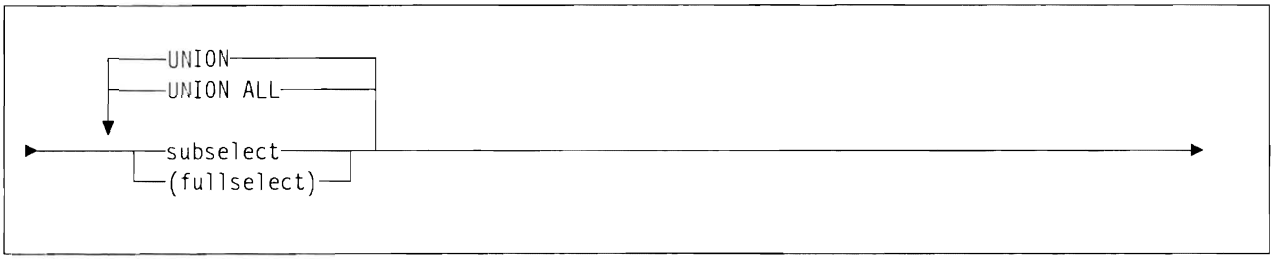
```
SELECT * FROM CORPDATA.EMP
```

Example 2: Show the job code, maximum salary, and minimum salary for each group of rows of CORPDATA.EMP with the same job code, but only for groups with more than one row and with a maximum salary greater than $50,000.

```
SELECT JOBCODE, MAX(SALARY), MIN(SALARY)
  FROM CORPDATA.EMP
  GROUP BY JOBCODE
  HAVING COUNT(*) > 1 AND MAX(SALARY) > 50000
```

[1] See Chapter 3, "Functions" on page 31 for restrictions that apply to the use of column functions.

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, however, 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.

Two rows are duplicates of one another only if each value in the first is equal to the corresponding value of the second.

Note that the UNION ALL operation is associative, and that

```
(SELECT PROJNO FROM CORPDATA.PROJ
UNION ALL
SELECT PROJNO FROM CORPDATA.TPROJEC)
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJA
```

will return the same results as

```
SELECT PROJNO FROM CORPDATA.PROJ
UNION ALL
(SELECT PROJNO FROM CORPDATA.TPROJEC
UNION ALL
SELECT PROJNO FROM CORPDATA.EMPPROJA)
```

When you include the UNION ALL operator in the same SQL statement as a UNION operator, however, the result of the operation depends on the order of evaluation. Where there are no parentheses, evaluation is from left to right. Where parentheses are included, the parenthesized subselect is evaluated first, followed, from left to right, by the other components of the statement.

Rules for columns: R1 and R2 must have the same number of columns. The description of the first column of R1 must be compatible with the description of the first column of R2, the description of the second column of R1 must be compatible with the description of the second column of R2, and so on.

In the following explanations, let *Column1* denote the *n*th column of R1, *Column2* the *n*th column of R2, and *Column3* the *n*th column of the result of a UNION or UNION ALL.

- ***String Columns:*** *Column3* will be a character string. If both *Column1* and *Column2* are fixed-length, *Column3* will be fixed-length. Otherwise, *Column3* will be varying-length. In either case, the length attribute of *Column3* will be the greater of the length attributes of *Column1* and *Column2*.
- ***Numeric Columns:*** *Column1* and *Column2* must both be numeric. The following rules govern the data type of *Column3*:
  - If *Column1* or *Column2* is floating-point, *Column3* is floating-point.
  - If *Column1* or *Column2* is floating-point, and the other is integer, numeric, or decimal, *Column3* is floating-point.
  - If *Column1* and *Column2* are decimal, *Column3* is decimal. If *p* and *s* are the precision and scale of *Column1*, and *p'* and *s'* are the precision and scale of *Column2*, the precision of *Column3* is MAX(*s*,*s'*)+MAX(*p*-*s*,*p'*-*s'*) and the scale of *Column3* is MAX(*s*,*s'*). The precision of *Column3* must not be greater than 31.
  - If *Column1* and *Column2* are numeric, *Column3* is numeric. The precision and scale of *Column3* can be calculated using the formulas above.
  - If *Column1* or *Column2* is decimal, and the other is integer or numeric, *Column3* is decimal. The precision and scale of *Column3* can be calculated using the formulas above.
  - If *Column1* or *Column2* is numeric, and the other is integer, *Column3* is numeric. The precision and scale of *Column3* can be calculated using the formulas above.
  - If *Column1* and *Column2* are large integer, *Column3* is large integer.
  - If *Column1* or *Column2* is large integer, and the other is small integer, *Column3* is large integer.
  - If *Column1* and *Column2* are small integer, *Column3* is small integer.
  - If *Column1* or *Column2* is nonzero scale binary, both *Column1* and *Column2* must be binary with the same scale.

In all cases, if *Column1* and *Column2* do not allow null values, *Column3* will not allow null values. Otherwise, *Column3* will permit null values. If the values of *Column1* or *Column2* must be converted to conform to *Column3*, the conversion operation is exactly the same as if the values were assigned to *Column3*. For example, if *Column1* is CHAR(10) and *Column2* is CHAR(5), *Column3* is CHAR(10) and values of *Column3* derived from *Column2* are padded on the right with five blanks.

Examples of a fullselect

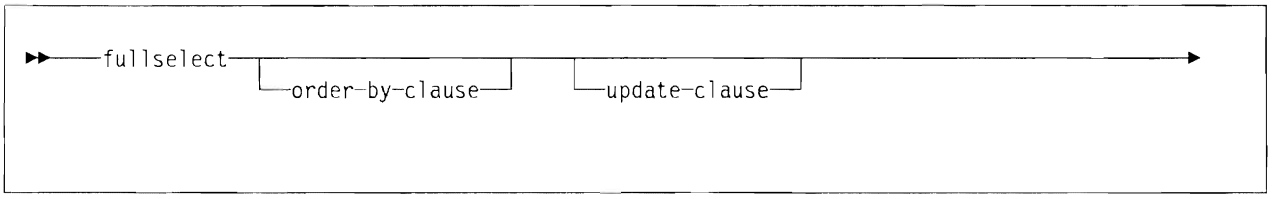
Example 1: Show all the rows from CORPDATA.EMP.

```
SELECT * FROM CORPDATA.EMP
```

Example 2: List the employee numbers of all employees whose department number begins with D (as determined from the employee table) OR who are assigned to projects whose project number begins with AD (as determined from the Employee-to-Project-Activity table).

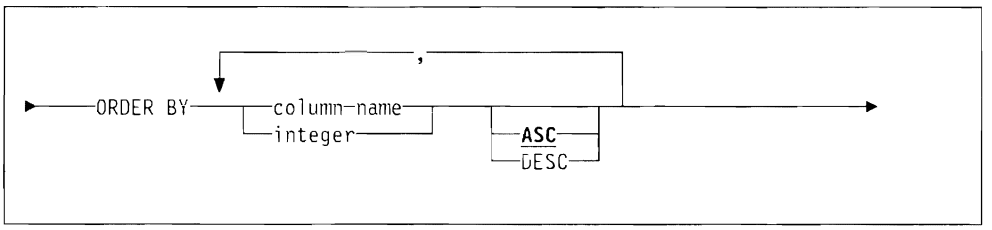
```
SELECT EMPNO FROM CORPDATA.EMP
  WHERE WORKDEPT LIKE 'D%'
  UNION
  SELECT EMPNO FROM CORPDATA.EMP.PROJA
  WHERE PROJNO LIKE 'AD%'
```

select-statement



The *select-statement* is the form of a query that can be prepared and subsequently executed by the use of an OPEN statement. It can also be issued interactively, using the interactive facility (STRSQL command), 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



Puts the rows of the result table in order by the values of the columns you identify. If you identify more than one column, the rows are ordered by the values of the first column you identify, then by the values of the second column, and so on.

column-name
Must unambiguously identify a column of the result table.

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.

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.

ASC
Uses the values of the column in ascending order. This is the default.

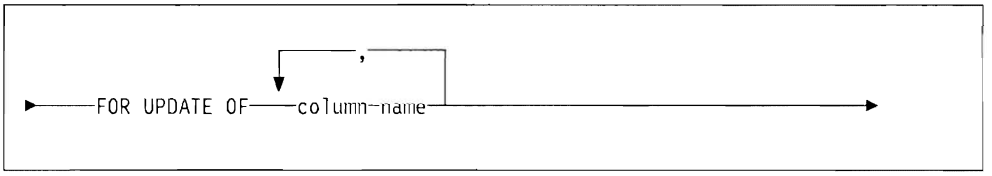
DESC
Uses the values of the column in descending order.

Ordering is performed in accordance with the comparison rules described in Chapter 2. If your ordering specification does not determine a complete ordering, rows with duplicate values of the last identified column have an arbitrary order.

With dynamic SQL, the ORDER BY clause, like the FOR UPDATE OF clause, must be specified when the SELECT statement is prepared, rather than on the DECLARE CURSOR statement.

The ORDER BY clause can contain up to 256 columns or 256 bytes. If the ORDER BY clause contains floating-point columns, only 120 columns or 120 bytes are allowed.

update-clause



The UPDATE statement can update only columns in the *column-name* list. Those columns must belong to the table or view named in the FROM clause of the fullselect. The column names must not be qualified.

If the FOR UPDATE OF clause is not specified and the ORDER BY clause is not specified, all columns can be updated.

With dynamic SQL, the FOR UPDATE OF clause, like the ORDER BY clause, must be specified when the SELECT statement is prepared, rather than on the DECLARE CURSOR statement.

The FOR UPDATE OF clause cannot be used if the result is read-only.

Examples of a select-statement

Example 1: Select all the rows from CORPDATA.TEMPL.

```
SELECT * FROM CORPDATA.TEMPL
```

Example 2: Select all the rows from CORPDATA.TEMPL in order by date of hiring.

```
SELECT * FROM CORPDATA.TEMPL ORDER BY HIREDATE
```

Chapter 5. Statements

This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the SQL statements listed in the table below.

Table 2 (Page 1 of 2). SQL Statements

SQL Statement	Function	Refer to
BEGIN DECLARE SECTION	Marks the beginning of a host variable declaration section.	p. 53
CLOSE	Closes a cursor.	p. 55
COMMENT ON	Replaces or adds a comment to the description of a table, view, or column.	p. 57
COMMIT	Terminates a unit of recovery and commits the database changes made by that unit of recovery.	p. 59
CREATE COLLECTION	Defines a collection.	p. 61
CREATE INDEX	Defines an index on a table.	p. 63
CREATE TABLE	Defines a table.	p. 65
CREATE VIEW	Defines a view of one or more tables or views.	p. 69
DECLARE CURSOR	Defines an SQL cursor.	p. 72
DECLARE STATEMENT	Declares names used to identify prepared SQL statements.	p. 75
DELETE	Deletes one or more rows from a table.	p. 76
DESCRIBE	Describes the result columns of a prepared statement.	p. 79
DROP	Deletes a collection, table, index, or view.	p. 81
END DECLARE SECTION	Marks the end of a host variable declaration section.	p. 83
EXECUTE	Executes a prepared SQL statement.	p. 85
EXECUTE IMMEDIATE	Prepares and executes an SQL statement.	p. 88
FETCH	Assigns values of a row to host variables.	p. 90
GRANT	Grants privileges on a table or view.	p. 93
INCLUDE	Inserts declarations into a source program.	p. 96
INSERT	Inserts one or more rows into a table.	p. 98
LABEL ON	Replaces or adds a label on the description of a table, view, or column.	p. 102
LOCK TABLE	Locks a table in shared or exclusive mode.	p. 104
OPEN	Opens a cursor.	p. 106
PREPARE	Prepares an SQL statement for execution.	p. 109
REVOKE	Revokes privileges on a table or view.	p. 113
ROLLBACK	Terminates a unit of recovery and backs out the database changes made by that unit of recovery.	p. 115
SELECT INTO	Specifies a result table of no more than one row and assigns the values to host variables.	p. 117

Table 2 (Page 2 of 2). SQL Statements		
SQL Statement	**Function**	**Refer to**
UPDATE	Updates the values of one or more columns in one or more rows of a table.	p. 119
WHENEVER	Defines actions to be taken on the basis of SQL return codes.	p. 123

How SQL Statements Are Invoked

Invocation

The SQL statements described in this chapter are classified as *executable* or *nonexecutable*. The 'Invocation' section in the description of each statement indicates whether or not the statement is executable.

An *executable statement* can be invoked in three ways:

- Embedded in an application program.
- Dynamically prepared and executed.
- Issued interactively.

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.

Besides the statements described in this chapter. there is one more SQL statement construct: the *select-statement*, as described under "select-statement" on page 47. It is not included in this chapter because it is used in a way different from other statements. A *select-statement* can be invoked in three ways:

- Included in DECLARE CURSOR and implicitly executed by OPEN.
- Dynamically prepared, referenced in DECLARE CURSOR, and implicitly executed by OPEN.
- Issued interactively.

The first two methods are called, respectively, the *static* and the *dynamic* invocation of *select-statement*.

The different methods of invoking an SQL statement are discussed below in more detail. For each method, the discussion includes: the mechanism of execution, the interaction with host variables, and testing whether the execution was successful or not.

Embedding a Statement in an Application Program

You may include SQL statements in a source program that will be submitted to the precompiler (by using the CRTSQLRPG, CRTSQLPLI, CRTSQLC, or CRTSQLCBL commands). Such statements are said to be *embedded* in the program. An embedded statement can be placed anywhere in the program where a host language statement would be allowed. You must precede each embedded statement with EXEC SQL.

Executable statements: An executable statement embedded in an application program is executed every time a statement of the host language would be executed if specified in the same place. (Thus, for example, a statement within a loop is executed every time the loop is executed, and a statement within a conditional construct is executed only when the condition is satisfied.)

An embedded statement may contain references to host variables. A host variable referenced in this way may 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, i.e., the variables are used as input. The treatment of other references is described individually for each statement.

The successful or unsuccessful execution of the statement is indicated by setting of the SQLCODE field in SQLCA. You should therefore follow all executable statements by a test of SQLCODE. 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.

Nonexecutable statements: An embedded nonexecutable statement is processed only by the precompiler. The precompiler reports any errors encountered in such statement. The statement is *never* executed, and acts as a no-operation if placed among executable statements of the application program. You should not, therefore, follow such statements by a test of the SQLCODE field in SQLCA.

Dynamic Preparation and Execution

Your application program may 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, obtained from a terminal). The statement so constructed can be prepared for execution by means of the (embedded) statement PREPARE, and executed by means of the (embedded) statement EXECUTE. Alternatively, you can use the (embedded) statement EXECUTE IMMEDIATE to prepare and execute a statement in one step.

A statement to be prepared must not contain references to host variables. It may instead contain parameter markers. (See "PREPARE" on page 109 for rules concerning the parameter markers.) When the prepared statement is executed, the parameter markers are effectively replaced by current values of the host variables specified in the EXECUTE statement. (See "EXECUTE" on page 85 for rules concerning this replacement.) Note that, once prepared, a statement can be executed several times, with different values of host variables.

The parameter markers are not allowed by EXECUTE IMMEDIATE.

The successful or unsuccessful execution of the statement is indicated by setting of the SQLCODE field in SQLCA after the EXECUTE (or EXECUTE

IMMEDIATE) statement. You should check it as described above for embedded statements.

Static Invocation of a select-statement

You may include a *select-statement* as a part of the (nonexecutable) statement DECLARE CURSOR. Such a statement is executed every time you open the cursor by means of the (embedded) statement OPEN.

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.

The successful or unsuccessful execution of *select-statement* is indicated by setting of the SQLCODE field in SQLCA after the OPEN. You should check it as described above for embedded statements.

Dynamic Invocation of a select-statement

Your application program may 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 expressed in terms of your application, obtained from a terminal). The statement so constructed may be prepared for execution by means of the (embedded) statement PREPARE, and referenced by a (nonexecutable) statement DECLARE CURSOR. The statement is then executed every time you open the cursor by means of the (embedded) statement OPEN.

The *select-statement* used in this way must not contain references to host variables. It may instead contain parameter markers. (See "PREPARE" on page 109 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 106 for rules concerning this replacement.)

The successful or unsuccessful execution of *select-statement* is indicated by setting of the SQLCODE field in SQLCA after the OPEN. You should check it as described above for embedded statements.

Interactive Invocation

A capability for entering SQL statements from a terminal is part of the architecture of the database manager. AS/400 system provides the STRSQL command for this facility. Other products are also available. A statement entered in this way is said to be issued interactively.

A statement issued interactively must not contain parameter markers or references to host variables, since these make sense only in the context of an application program. For the same reason, there is no SQLCA involved. Interactive SQL statements are processed using dynamic SQL and are therefore similarly restricted.

BEGIN DECLARE SECTION

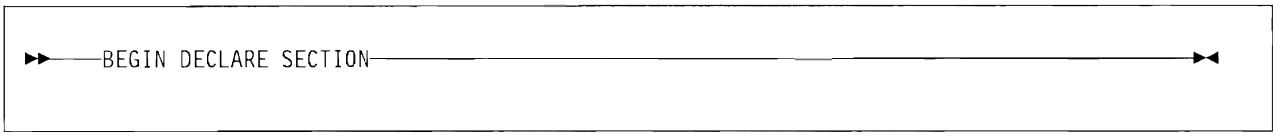
The BEGIN DECLARE SECTION statement marks the beginning of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.



Description

The BEGIN DECLARE SECTION statement may be coded in the application program wherever variable declarations can appear in accordance with the rules of the host language. The exception, however, is that it may not be coded in the middle of a host structure. The BEGIN DECLARE SECTION statement is used to indicate the beginning of a host variable declaration section. A host variable section ends with an END DECLARE SECTION statement, described on page 83.

Notes

If declare sections are specified in the program, only the variables declared within the declare sections can be used as host variables. If declare sections are not specified in the program, all variables in the program are eligible for use as host variables.

Host variable declaration sections should be specified for host languages so that the source program conforms to the SAA definition of SQL.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

No other SQL statements should be included within a declare section.

Variables referenced in SQL statements should be declared in a declare section and the section should appear before the first reference to the variable.

Variables declared outside a declare section should not have the same name as variables declared within a declare section.

More than one declare section can be specified in the program.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must not be specified in RPG programs.

Example

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   .
   .
 (host variable declarations)
   .
   .
EXEC SQL END DECLARE SECTION END-EXEC.
```

CLOSE

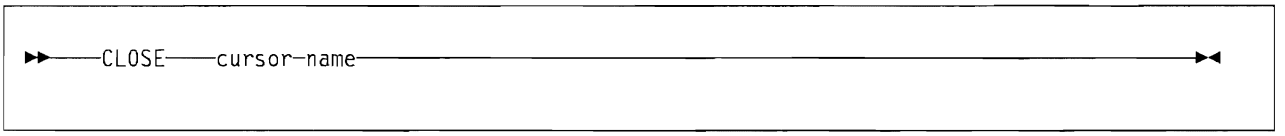
The CLOSE statement closes a cursor.

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 72 for the authorization required to use a cursor.



Description

cursor-name
Identifies the cursor to be closed. The *cursor-name* must identify a declared cursor as explained in the DECLARE CURSOR statement. When the CLOSE statement is executed, the cursor must be in the open state.

Notes

If cursors are not explicitly closed, the open cursors of an application process are implicitly closed:

- At the end of a unit of recovery unless HOLD is specified on the COMMIT or ROLLBACK statement.
- At the end of the first SQL program in the program stack. For example, if SQL program A calls SQL program B, any cursors opened by program B will be closed when program A ends.
- At the end of the job.

Explicitly closing cursors as soon as possible can improve performance. CLOSE is *not* a COMMIT or ROLLBACK operation.

Example

A cursor is used to fetch one row at a time into the program variables DNUM, DNAME, and MNUM. Finally, the cursor is closed. If the cursor is reopened, it is again located at the beginning of the rows to be fetched.

```
   EXEC SQL DECLARE C1 CURSOR FOR
      SELECT DEPTNO, DEPTNAME, MGRNO
      FROM CORPDATA.DEPT
      WHERE ADMRDEPT = 'A00'
      END-EXEC.

   EXEC SQL OPEN C1 END-EXEC.

   EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.

   IF SQLCODE = 100
      PERFORM DATA-NOT-FOUND
   ELSE
      PERFORM GET-REST-OF-DEPT
      UNTIL SQLCODE IS NOT EQUAL TO ZERO.

   EXEC SQL CLOSE C1 END-EXEC.

GET-REST-OF-DEPT.
   EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.
```

COMMENT ON

The COMMENT ON statement adds or replaces comments in the catalog descriptions of tables, views, or columns.

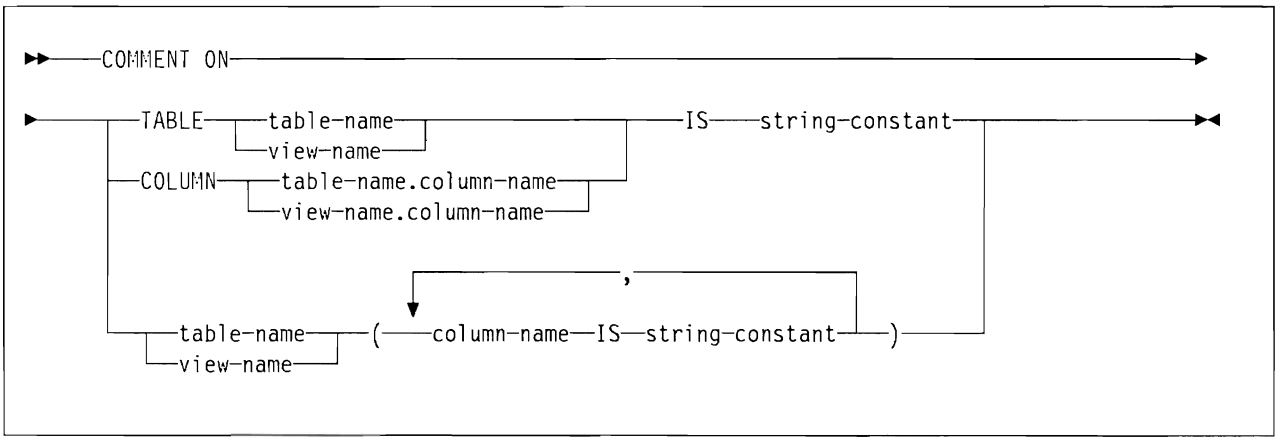
Invocation

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:

- The system authority *READ on the library containing the table or view, and
- Ownership of the table or view, or the system authorities of both *OBJOPR and *OBJMGT on the referenced table or view.



Description

TABLE
Indicates that you want to comment on a table or view.

table-name or *view-name*
Identifies the table or view to which the comment applies. The table or view must be described in the catalog.

COLUMN
Indicates that you want to comment on a column.

table-name.column-name or *view-name.column-name*
Is the name of the column, qualified by the name of the table or view in which it appears.

To comment on more than one column in a table or view, do not use TABLE or COLUMN. Give the table or view name and then, in parentheses, a list of this form:

```
column-name IS string-constant,
  column-name IS string-constant, ...
```

The column named must be described in the catalog, and in the referenced table or view.

IS

Introduces the comment you want to make.

string-constant

Can be any SQL character string constant of up to 254 characters. The constant may contain double-byte characters as well as EBCDIC characters.

Notes

The library that contains the object must be an SQL database.

Examples

Example 1: Enter a comment on table CORPDATA.EMP.

```
COMMENT ON TABLE CORPDATA.EMP
  IS 'REFLECTS 1ST QTR 81 REORG'
```

Example 2: Enter a comment on view CORPDATA.VDEPT.

```
COMMENT ON TABLE CORPDATA.VDEPT
  IS 'VIEW OF TABLE CORPDATA.DEPT'
```

Example 3: Enter a comment on the DEPTNO column of table CORPDATA.DEPT.

```
COMMENT ON COLUMN CORPDATA.DEPT.DEPTNO
  IS 'DEPARTMENT ID - UNIQUE'
```

Example 4: Enter comments on two columns in table CORPDATA.DEPT.

```
COMMENT ON CORPDATA.DEPT
  (MGRNO IS 'EMPLOYEE NUMBER OF DEPARTMENT MANAGER',
   ADMRDEPT IS 'DEPARTMENT NUMBER OF ADMINISTERING DEPARTMENT')
```

COMMIT

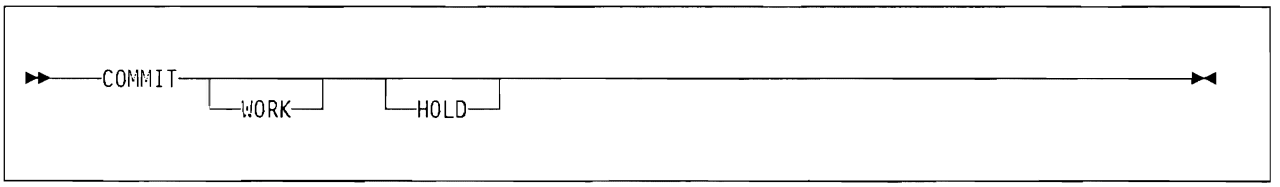
The COMMIT statement terminates a unit of recovery and commits the database changes that were made by that unit of recovery. The moment in the sequence of operations when that is done is called a *commit point*.

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.



Description

The unit of recovery in which the COMMIT statement is executed is terminated and a new unit of recovery is initiated. All changes made by DELETE, INSERT, and UPDATE statements executed during the unit of recovery are committed.

All locks acquired by the unit of recovery are released. All cursors that were opened during the unit of recovery are closed. All statements that were prepared during the unit of recovery are destroyed, and any cursors associated with the prepared statements are invalidated.

WORK
COMMIT WORK has the same effect as COMMIT. SQL/400 accepts the keyword WORK for compatibility with other database products.

HOLD
Indicates a hold on resources. If specified, currently open cursors are not closed, prepared SQL statements are preserved, and all resources acquired during the unit of recovery are held. Locks on specific rows acquired during the transaction, however, are released. If HOLD is omitted, open cursors are closed, prepared SQL statements discarded, and held resources released.

Notes

The termination of an application process is an implicit rollback. Thus, an explicit COMMIT or ROLLBACK should be issued before termination.

A unit of recovery (see "Application Processes, Concurrency, and Recovery" on page 3 for description) may include the processing of up to 4096 rows, including rows retrieved during a SELECT or FETCH statement[2], and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE

[2] Unless you specified COMMIT(*CHG), in which case these rows are not included in this total.

operations.[3] A unit of recovery is initiated by the initiation of a unit of work or by the termination of a previous unit of recovery. It is terminated by a commit operation, a rollback operation, or the termination of a unit of work. The commit and rollback operations do not affect any data definition statements, and these statements are not, therefore, allowed in an application program that also specifies COMMIT(*CHG) or COMMIT(*ALL). The data definition statements are:

- COMMENT
- CREATE COLLECTION
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- DROP COLLECTION
- DROP INDEX
- DROP TABLE
- DROP VIEW
- GRANT
- LABEL
- REVOKE

Commitment control is implicitly started by SQL, if necessary, using the system CL command STRCMTCTL. The lock level used is based on the COMMIT option specified on either the CRTSQL*xxx* (where *xxx* is RPG, CBL, C, or PLI) or the STRSQL command.

A COMMIT is not automatically performed when an application terminates or when interactive SQL terminates. In order to commit work performed by an application, you must issue a COMMIT from within the application, or from outside the application with the CL command COMMIT.

When a job ends, an implicit ROLLBACK is issued.

Example

Commit alterations to the database made since the last commit point.

```
COMMIT WORK
```

[3] This limit also includes any records accessed or changed through files opened under commitment control through high-level language file processing.

CREATE COLLECTION

The CREATE COLLECTION[4] statement defines a collection in which tables, views, and indexes may later be created.

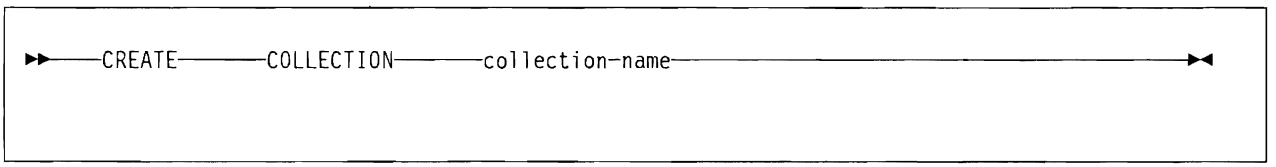
Invocation

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:

- Authority to the CL command CRTLIB, and
- Authority to the CL command CRTDTADCT (create data dictionary).



Description

collection-name
Names the collection. The name you supply must not be the name of an existing SQL collection or a library.

Notes

A collection is created as:

- A library: a library groups related objects, and allows you to find objects by name.
- A catalog: a catalog contains descriptions of the tables, views, and indexes in the collection. A catalog consists of a data dictionary and a set of views and logical files. For more information, see *SQL/400 Programmer's Guide*.
- A journal and journal receiver: a journal QSQJRN and journal receiver QSQJRN0001 is created in the collection, and is used to record changes to all tables subsequently created in the collection. For more information, see *Backup and Recovery Guide*.

If SQL names have been specified, the owner of the collection is the authorization ID of the statement. If SYSTEM names have been specified, the owner of the collection is the user profile (or the group user profile of the job) invoking the statement.

When it is created, the system authority *EXCLUDE is initially given to *PUBLIC. The owner is the only user having any authority to the collection. If other users require authority to the collection, the owner can grant authority to the objects created, using the CL command GRTOBJAUT (grant object authority). For more information on AS/400 system security, see *Programming: Security Concepts and Planning* and *Programming: Control Language Reference*.

[4] CREATE DATABASE may be used as a synonym to provide compatibility with a previous release.

Example

Create collection DBTEMP.

```
CREATE COLLECTION DBTEMP
```

CREATE INDEX

The CREATE INDEX statement creates an index on a table.

Invocation

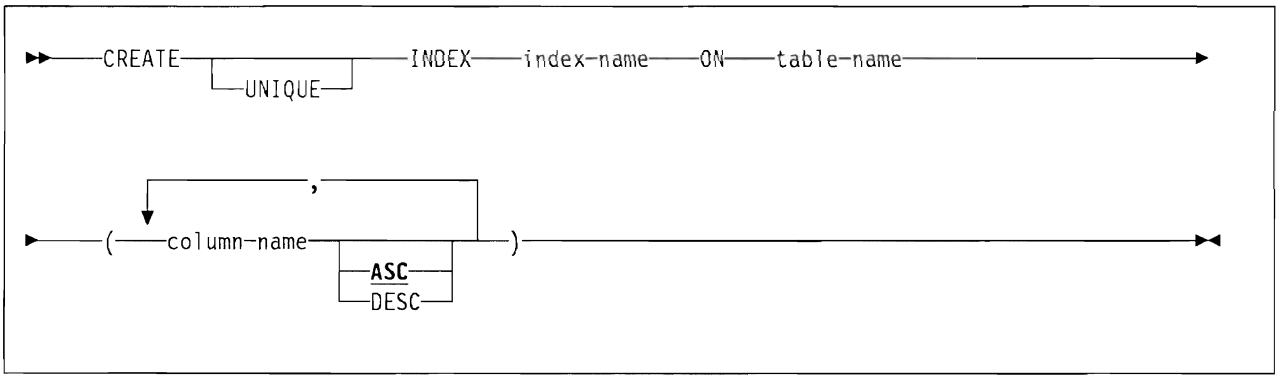
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include:

- Authority to the CL command CRTLF, and
- Authorities of *OBJOPR and *ADD on the library and data dictionary containing the referenced table, and
- The INDEX privilege, and one of the following privileges on the referenced table:
  - DELETE
  - INSERT
  - SELECT
  - UPDATE

If SQL names are specified and the authorization ID is explicitly specified and is different from the authorization ID of the statement, you must have *ADD system authority to the user profile named by the authorization ID qualifier.



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.

INDEX *index-name*
Names the index. If the *index-name* is qualified, the index will be created in the specified collection. Otherwise the index will be created in the collection specified by the implicit or explicit qualifier of the specified table. The name you give must not be the name of an index, table, view, or file that already exists in the collection.

If SQL naming is specified and the implicit or explicit qualifier also identifies a user profile, the "owner" of the index is that user profile. Otherwise the "owner" is the user profile or group user profile of the job invoking the statement.

ON *table-name*
Names the table on which you want the index to be created. The *table-name* must name a table (not a view) described in the catalog.

(column-name)
Names a column that is to be part of the index key.

Each *column-name* identifies a column of the table. Do not name more than 120 columns. The same column may be specified more than once. Do not qualify the *column-name*.

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.

Notes

An index is created as a keyed logical file. Indexes are created with the system authority of *EXCLUDE on *PUBLIC. The maximum length of an index entry is 120 bytes.

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 index always reflects the current condition of the table.

Example

Create a unique index, named XDEPT1, on table CORPDATA.DEPT. Index entries are to be in ascending order by the single column DEPTNO.

```
CREATE UNIQUE INDEX CORPDATA.XDEPT1
  ON CORPDATA.DEPT
    (DEPTNO ASC)
```

CREATE TABLE

The CREATE TABLE statement defines a table. You provide the name of the table and the names and attributes of its columns.

Invocation

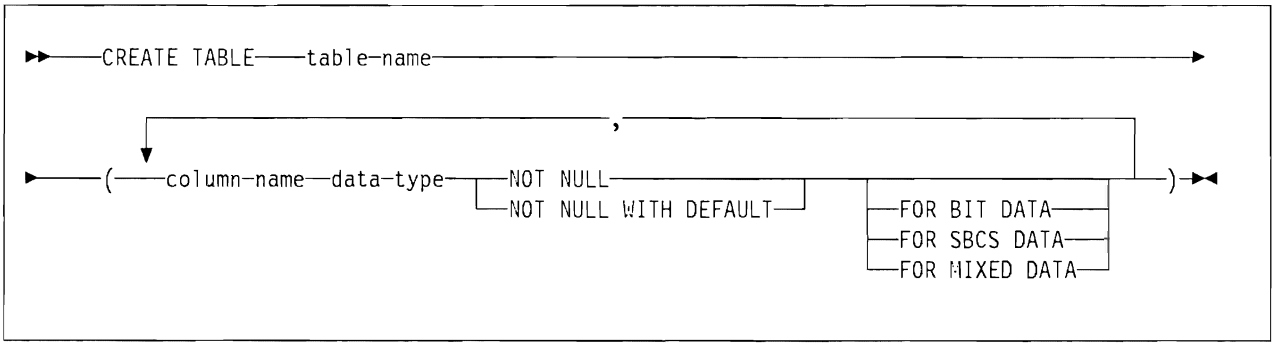
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include:

- The system authorities of *OBJOPR and *ADD on the library and data dictionary, and
- Authority to the CL command CRTPF.

If SQL names are specified and the authorization ID is explicitly specified and is different from the authorization ID of the statement, you must have *ADD system authority to the user profile named by the authorization ID qualifier.



Description

table-name

Is the name of the table. The name you supply, including the implicit or explicit qualifier, must not identify an index, table, view, or file that already exists in the collection.

If SQL names have been specified, the table will be created in the collection specified by the implicit or explicit qualifier. The qualifier is the "owner" of the table if a user profile with that name exists. Otherwise, the "owner" of the table is the user profile or group user profile of the job invoking the statement.

If system names have been specified, the table name must be qualified. The table will be created in the collection specified by the qualifier. The "owner" of the table is the user profile or group user profile of the job invoking the statement.

column-name

Is the name of 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.

You may define up to 8000 columns. The sum of the byte counts of the columns must not be greater than 32766. For information on the byte counts of columns according to data type see "Notes" on page 70.

data-type
Is one of the types in the following list. Use:

INTEGER or **INT**
For a large integer.

SMALLINT
For a small integer.

FLOAT(*integer***)** or **FLOAT**
For a floating-point number. If the integer is between 1 and 24 inclusive, the format is that of single precision floating-point. If the integer is between 25 and 53 inclusive, the format is that of double precision floating-point. If the integer is omitted from the specification, a value of 53 is assumed, and the number is double precision.

You may also specify:

REAL	for single precision floating-point
DOUBLE PRECISION	for double precision floating-point

NUMERIC(*integer***,***integer***)**
NUMERIC(*integer***)**
NUMERIC
For a zoned decimal number. The first integer is the precision of the number, that is, the total number of digits; it may range from 1 to 31. The second integer is the scale of the number, that is, the number of digits to the right of the decimal point; it may range from 0 to the precision.

You may use NUMERIC(*p*) for NUMERIC(*p*,0), and NUMERIC for NUMERIC(5,0).

DECIMAL(*integer***,***integer***)** or **DEC(***integer***,***integer***)**
DECIMAL(*integer***)** or **DEC(***integer***)**
DECIMAL or **DEC**
For a packed decimal number. The first integer is the precision of the number; that is, the total number of digits; it may range from 1 to 31. The second integer is the scale of the number; that is, the number of digits to the right of the decimal point; it may range from 0 to the precision.

You may use DECIMAL(*p*) for DECIMAL(*p*,0), and DECIMAL for DECIMAL(5,0). You may also specify DEC for decimal.

CHARACTER(*integer***)** or **CHAR(***integer***)**
CHARACTER or **CHAR**
For a fixed-length character string of length *integer*, which may range from 1 to 32766. If FOR MIXED DATA is specified, the range is 4 to 32766. If the length specification is omitted, a length of 1 character is assumed.

NOT NULL
Prevents the column from containing null values.

NOT NULL WITH DEFAULT
Prevents the column from containing null values, and allows a default value other than the null value. The default value used depends on the data type of the column, as follows:

Data type	Default value
Numeric	0
Character	blanks

FOR BIT DATA
Specifies that the character column contains hex data (that is, data that is not text of a particular code page). A zero is returned for the character set and code page in the SQL Descriptor Area (SQLDA) on a DESCRIBE or PREPARE statement for a character column defined with FOR BIT DATA.

FOR SBCS DATA
Specifies that the character column contains SBCS (single-byte character set) data. The system value QCHRID specifies the character set and code page of all SBCS data on the system. The character set and code page of a character column is returned in the SQL Descriptor Area (SQLDA) on DESCRIBE and PREPARE statements. FOR SBCS DATA is the default for CHAR columns if the system is not DBCS-capable or if the length of the column is less than 4. This is determined using the QIGC system value.

FOR MIXED DATA
Specifies that the character column contains both SBCS (single-byte character set) data, and DBCS (double-byte character set) data. The system value QCHRID specifies the character set and code page of the SBCS data. The character set and code page of a character column is returned in the SQL Descriptor Area (SQLDA) on DESCRIBE and PREPARE statements. FOR MIXED DATA is the default for CHAR columns if the system is DBCS-capable and the length of the column is greater than 3. This is determined using the QIGC system value. If the system is not DBCS-capable, and FOR MIXED DATA is specified, an error occurs. A FOR MIXED DATA column is used as a DBCS-Open database field.

Notes

Tables are created as physical files with the system authority of *EXCLUDE to *PUBLIC. When a table is created, journaling is automatically started on the journal named QSQJRN in the database.

Maximum record size: The "maximum record size" referred to in the description of *column-name* is 32766. To determine the length of a record, add the length of each column of that record based on the byte count of the data type.

The list that follows gives the byte counts of columns by data type.

Data type	Byte count
INTEGER	4
INT	4
SMALLINT	2
FLOAT(*n*)	If *n* is between 1 and 24, the byte count is 4. If *n* is between 25 and 53, the byte count is 8.
DOUBLE PRECISION	8
REAL	4
DECIMAL(*p*, *s*)	the integral part of (*p*/2) + 1

NUMERIC(*p*, *s*)	*p*
CHAR(*n*)	*n*

Precision as described to the database:

- Floating point fields are defined in the AS/400 system database with a decimal precision, not a bit precision. The algorithm used to convert the number of bits to decimal is *decimal precision = x(n/3.31)*, where *x* is the smallest integer greater than or equal to the argument, and *n* is the number of bits to convert. The decimal precision is used to determine how many digits to display using interactive SQL.
- SMALLINT fields are stored with a decimal precision of 4,0.
- INTEGER fields are stored with a decimal precision of 9,0.

Examples

Example 1: Create CORPDATA.DEPT. DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT are column names. CHAR means the column will contain character data. NOT NULL means that the column cannot contain a null value.

```
CREATE TABLE CORPDATA.DEPT
  (DEPTNO   CHAR(3)     NOT NULL WITH DEFAULT,
   DEPTNAME CHAR(36)    NOT NULL WITH DEFAULT,
   MGRNO    CHAR(6)     NOT NULL WITH DEFAULT,
   ADMRDEPT CHAR(3)     NOT NULL WITH DEFAULT)
```

Example 2: Create CORPDATA.PROJ. PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF, PRSTDATE, PRENDATE, and MAJPROJ are column names. CHAR means the column will contain character data. DECIMAL means the column will contain packed decimal data. 5,2 means the following: 5 indicates the number of decimal digits, and 2 indicates the number of digits to the right of the decimal point. NOT NULL means that the column cannot contain a null value.

```
CREATE TABLE CORPDATA.PROJ
  (PROJNO   CHAR(6)      NOT NULL WITH DEFAULT,
   PROJNAME CHAR(24)     NOT NULL WITH DEFAULT,
   DEPTNO   CHAR(3)      NOT NULL WITH DEFAULT,
   RESPEMP  CHAR(6)      NOT NULL WITH DEFAULT,
   PRSTAFF  DECIMAL(5,2) NOT NULL WITH DEFAULT,
   PRSTDATE DECIMAL(6)   NOT NULL WITH DEFAULT,
   PRENDATE DECIMAL(6)   NOT NULL WITH DEFAULT,
   MAJPROJ  CHAR(6)      NOT NULL WITH DEFAULT)
```

CREATE VIEW

The CREATE VIEW statement creates a view on one or more tables or views.

Invocation

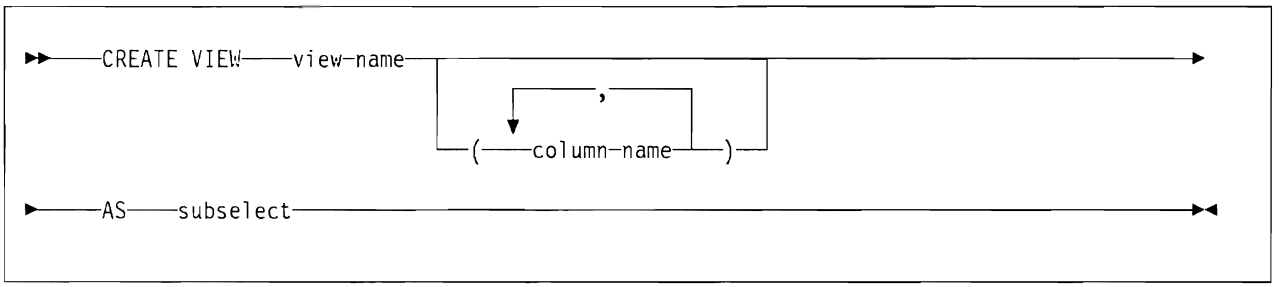
This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include:

- Authority to the CL command CRTLF (create logical files), and
- The system authorities *OBJOPR and *ADD on the library and data dictionary containing the referenced tables, and
- The SELECT, UPDATE, DELETE, or INSERT privilege on all the tables referenced either directly through the SELECT statement, or indirectly through views referenced in the SELECT statement.

If SQL names are specified and the authorization ID that is explicitly specified is different from the authorization ID of the statement, you must have *ADD authority on the user profile named by the authorization ID qualifier.



Description

view-name

Is the unqualified or qualified name of the view. The unqualified name must not be the same as any table, view, index, or file that already exists in the collection.

If SQL names have been specified, the view will be created in the collection specified by the implicit or explicit qualifier. If a user profile with the SQL name exists, the qualifier is the "owner" of the view. If a user profile with the SQL name does not exist, the "owner" of the view is the user profile (or group user profile of the job) invoking the statement.

If SQL names have been specified, the view will be created in the collection specified by the explicit qualifier. If the view name is not explicitly qualified, the view is created in the collection that contains the first table referenced in the subselect.

column-name

Is a list of one or more names for columns in the view. If you specify the list, it must consist of as many names as there are columns in the result table of the subselect. Each 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 executed.

subselect must not reference host variables . For an explanation of *subselect*, see Chapter 4, "Queries" on page 39.

Notes

Views are created as non-keyed logical files with system authority of *EXCLUDE to *PUBLIC.

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 keyword DISTINCT in the first SELECT clause
- A GROUP BY clause in the outer subselect
- A HAVING clause in the outer subselect
- A column function in the first SELECT clause
- The first FROM clause identifies a read-only view .

A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement.

A view cannot reference more than 32 real tables, including real tables referenced by underlying views.

A view cannot address more than 8000 columns. The number of referenced tables, the column name lengths. and the length of the WHERE clause further reduce this number.

Limitations

- FOR UPDATE OF, ORDER BY, and UNION cannot be used in the definition of a view.
- Host variables are not allowed in the definition of a view.
- USER or LENGTH cannot be used in the definition of a view.

Testing a view definition: You can test the semantics of your view definition by executing SELECT * FROM *view-name*.

Example

Create the view CORPDATA.VPROJRE1. PROJNO, PROJNAME, PROJDEP, RESPEMP, EMPNO, FIRSTNME, MIDINIT, and LASTNAME are column names. The view is a join of tables PROJ and CORPDATA.EMP, where a value in the RESPEMP column is equal to a value in the EMPNO column.

```
CREATE VIEW CORPDATA.VPROJRE1
  (PROJNO,PROJNAME,PROJDEP,RESPEMP,
   FIRSTNME,MIDINIT,LASTNAME)
  AS SELECT ALL
  PROJNO,PROJNAME,DEPTNO,EMPNO,
  FIRSTNME,MIDINIT,LASTNAME
  FROM CORPDATA.PROJ, CORPDATA.EMP
  WHERE RESPEMP = EMPNO
```

DECLARE CURSOR

The DECLARE CURSOR statement defines a cursor.

Invocation

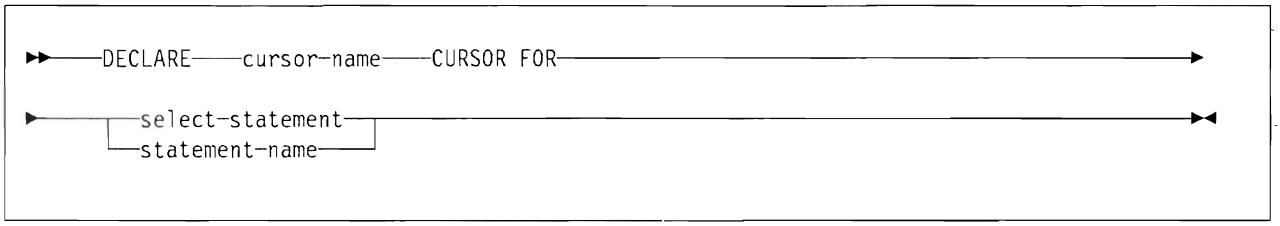
This statement can only be embedded in an application program. It is not an executable statement.

Authorization

No authorization is required to use this statement. However, to use OPEN or FETCH for the cursor, the privileges held by the authorization ID of the statement must include the SELECT privilege on every table and view identified in the SELECT statement of the cursor. Authority is checked at execution time during OPEN and during the processing of the statements that reference the open cursor.

The SELECT statement of the cursor is either:

- The SELECT statement identified by *select-statement*, in which case the authorization ID is the owner of the program, or
- The prepared SELECT statement identified by a *statement-name* clause, in which case the authorization ID is the run-time authorization ID.



Description

A cursor with the specified name is defined. The name must not be the same as the name of another cursor declared in your source program.

A cursor in the open state designates a *result table* and a position relative to the rows of that table. The table is the result table specified by the SELECT statement of the cursor.

The result table is read-only if:

- The SELECT statement includes:
  - The keyword DISTINCT in the first SELECT clause
  - A UNION operator
  - A column function in the first SELECT clause
  - A GROUP BY or HAVING clause.
- The first FROM clause of the SELECT statement identifies:
  - More than one table or view
  - A read-only view.

Specifying the SELECT Statement: If *select-statement* is specified, it identifies the SELECT statement of the cursor.

The *select-statement* must not include parameter markers, but may include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program. See "select-statement" on page 47 for an explanation of fullselect.

If the ORDER BY clause is not specified, the rows of the result table have an arbitrary order.

Naming the SELECT Statement: If a *statement-name* is specified, the SELECT statement of the cursor is the prepared SELECT statement identified by the *statement-name* when the cursor is opened. The *statement-name* must not be identical to a *statement-name* specified in another DECLARE CURSOR statement of your source program.

For an explanation of prepared SELECT statements, see "PREPARE" on page 109.

Notes

A SELECT statement is evaluated at the time the cursor is opened. If the same cursor is opened, closed, and then opened again, the results may be different. Multiple cursors using the same SELECT statement may be opened concurrently. They are each considered independent activities.

A cursor is automatically closed when the job terminates. A cursor is also closed whenever a COMMIT (no HOLD) or ROLLBACK (no HOLD) statement is issued, or when the last SQL program in the program stack ends.

If ORDER BY is specified and FOR UPDATE OF is not specified, the cursor is read-only. If ORDER BY is specified and FOR UPDATE OF is specified, the columns in the FOR UPDATE OF clause can not be the same as any columns specified in the ORDER BY clause.

The ORDER BY clause can contain up to 256 columns or 256 bytes. If the ORDER BY clause contains floating-point columns, only 120 columns or 120 bytes are allowed. If more than 120 bytes are used, the cursor is read-only.

If the FOR UPDATE OF clause is omitted, only the columns in the SELECT clause of the subselect that can be updated can be changed.

The DECLARE CURSOR statement must precede all statements that explicitly reference the cursor by name.

The scope of *cursor-name* is the source program in which it is defined; that is, the program submitted to the precompiler. Thus, you can only reference a cursor by statements that are precompiled with the cursor declaration. For example, a program called from another separately compiled program cannot use a cursor that was opened by the calling program.

Examples

Example 1: The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM CORPDATA.DEPT
   WHERE ADMRDEPT = 'A00'
   END-EXEC.
```

Example 2: The DECLARE CURSOR statement associates the cursor name C1 with the results of the SELECT. MGRNO and MGRNAME may be updated. FOR UPDATE OF can specify a column that is not in the select list.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO
   FROM CORPDATA.DEPT
   WHERE ADMRDEPT = 'A00'
   FOR UPDATE OF MGRNO, MGRNAME
   END-EXEC.
```

DECLARE STATEMENT

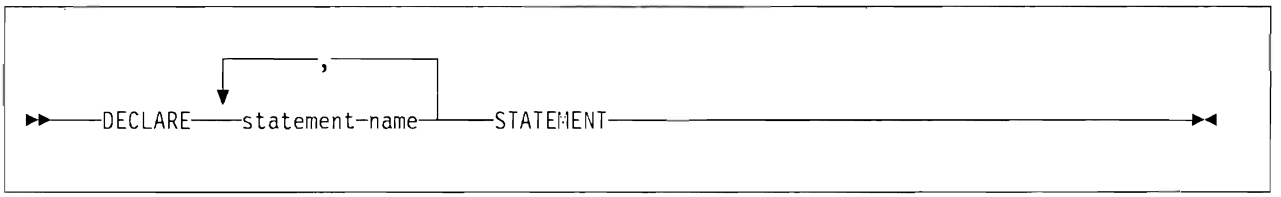
The DECLARE STATEMENT statement is used for program documentation. It declares names that are used to identify prepared SQL statements.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.



Description

statement-name **STATEMENT**
Lists one or more names that are used in your program to identify prepared SQL statements.

Example

This example shows the use of the DECLARE STATEMENT statement in a PL/I program.

```
EXEC SQL DECLARE OBJ_STMT STATEMENT;

( SOURCE_STATEMENT is "SELECT DEPTNO, DEPTNAME,
  MGRNO FROM CORPDATA.DEPT WHERE ADMRDEPT = 'A00'"  )

EXEC SQL INCLUDE SQLDA;
EXEC SQL DECLARE C1 CURSOR FOR OBJ_STMT;

EXEC SQL PREPARE OBJ_STMT FROM SRCE_STMT;
EXEC SQL DESCRIBE OBJ_STMT INTO SQLDA;

(Examine SQLDA)

EXEC SQL OPEN C1;

DO WHILE (SQLCODE = 0);
  EXEC SQL FETCH C1 USING DESCRIPTOR SQLDA;

(Print results)

END;

EXEC SQL CLOSE C1;
```

DELETE

The DELETE statement deletes rows from a table or view. Deleting a row from a view deletes the row from the table on which the view is based.

There are two forms of this statement:

- The *searched* DELETE form is used to delete one or more rows (optionally determined by a search condition).
- The *positioned* DELETE form is used to delete exactly one row (as determined by the current position of a cursor).

Invocation

A searched DELETE statement can be embedded in an application program or issued interactively. A positioned DELETE must be embedded in an application program.

Both forms are executable statements that can be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include the DELETE privilege on the specified table or view.

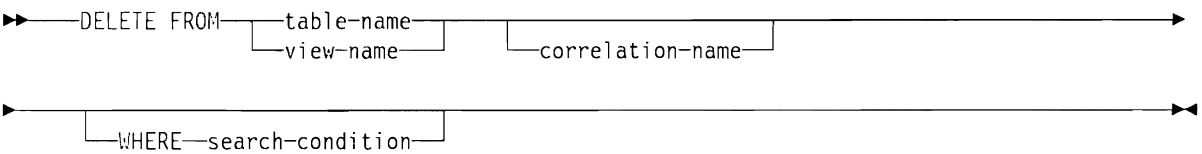
You have the DELETE privilege on a table if any of the following apply:

- You are the owner of the table
- You have been granted the DELETE privilege on the table
- You have been granted the system authorities *OBJOPR and *DLT on the table.

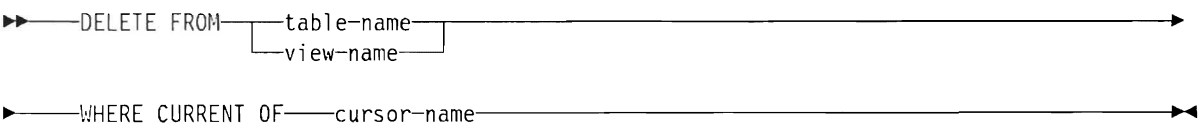
You have the DELETE privilege on a view if any of the following apply:

- You have been granted the DELETE privilege on the view
- You created the view, you had the DELETE privilege on its base table when the view was created, and you still have that DELETE privilege
- You have been granted the system authority *OBJOPR on the view and the system authority *DLT on the base table.

searched delete



positioned delete



Description

FROM *table-name* or *view-name*

Names the table or view from which you want to delete. It must have been created previously, but must not be 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 69.)

correlation-name

May be used within the *search-condition* to designate the table or view. (For an explanation of *correlation-name*, see Chapter 2.)

WHERE

Specifies a condition that selects 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 2. Each *column-name* in the search condition 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.

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 the Notes for the DECLARE CURSOR statement.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR" on page 72.)

When the DELETE statement is executed, the cursor must be positioned on a row: that row is the one deleted. After the deletion, the cursor is positioned before the next row of its result table. If there is no next row, the cursor is positioned after the last row.

Note that the deletion of a row WHERE CURRENT OF a specified cursor may leave other cursors pointing to the deleted record.

Notes

A maximum of 4096 rows may be deleted in any single DELETE operation when COMMIT(*ALL) or COMMIT(*CHG) has been specified.

If an error occurs during the execution of a DELETE statement and COMMIT(*ALL) or COMMIT(*CHG) was specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of recovery made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

When a DELETE statement is completed, the number of rows deleted is returned in SQLERRD(3) in the SQLCA. (For a description of the SQLCA, see "SQL Communication Area (SQLCA)" on page 127.)

One or more exclusive locks are acquired by the successful execution of a DELETE statement. Until the locks are released, they may prevent other application processes from performing operations on the table. For further

information about locking, see the description of the COMMIT, ROLLBACK, and LOCK TABLE statements. Refer also to *Database Guide*.

Examples

Example 1: Delete one row from table CORPDATA.DEPT.

```
DELETE FROM CORPDATA.DEPT
  WHERE DEPTNO = 'D11'
```

Example 2: From the table CORPDATA.EMP, delete all rows for departments E11 and D21.

```
DELETE FROM CORPDATA.EMP
  WHERE WORKDEPT = 'E11'
  OR WORKDEPT = 'D21'
```

DESCRIBE

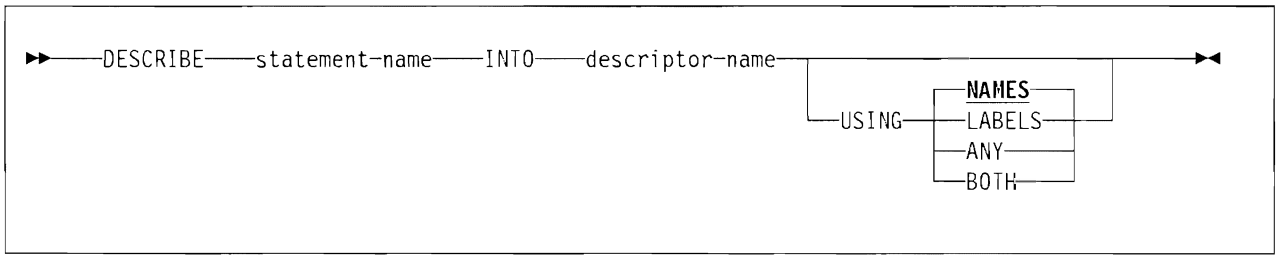
The DESCRIBE statement obtains information about a prepared statement. For an explanation of prepared statements, see "PREPARE" on page 109.

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 109 for the authorization required to create a prepared statement.



Description

statement-name
Names the statement about which you want to obtain information. When the DESCRIBE statement is executed, the name must identify a prepared statement.

INTO *descriptor-name*
Names an SQL descriptor area (SQLDA). When the DESCRIBE statement is executed, values are assigned to the variables of the SQLDA. For information about the format of an SQLDA, see "The SQL Descriptor Area (SQLDA)" on page 133.

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 will be needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2*n* on the PREPARE statement (where *n* is the number of columns in the result table). Then, on any later FETCH statement, set SQLN to *n*. 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

Information about a prepared statement can also be obtained by using the INTO clause of the PREPARE statement.

Before the DESCRIBE or PREPARE INTO statement is executed, the value of SQLN must be set to a value greater than or equal to zero to indicate how many occurrences of SQLVAR are provided in the SQLDA. (Enough storage must be allocated to allow for all occurrences of SQLN.) To obtain the description of the columns of the result table of a prepared SELECT statement, the number of occurrences of SQLVAR must not be less than the number of columns.

If USING BOTH is specified and SQLN is less than 2*SQLD, then SQLD is set to 2 * (number of columns). If USING BOTH is specified and SQLN is greater than or equal to 2*SQLD, then SQLD is set to the number of columns.

Because the maximum number of columns is 8000, a simple technique is to provide an SQLDA with 8000 occurrences of SQLVAR. However, such an SQLDA will occupy a good deal of space, and most of this space will not be needed for most prepared statements. Thus you might want to consider another technique, such as the following:

Execute a DESCRIBE or PREPARE INTO statement with an SQLDA that has no occurrences of SQLVAR. If SQLD is greater than zero, use the value to allocate an SQLDA with the necessary number of occurrences of SQLVAR and then execute a DESCRIBE statement using that SQLDA.

Example

This PL/I example uses the technique described above. SOURCE is a varying-length string variable and SHORTDA is an SQLDA with no occurrences of SQLVAR.

```
EXEC SQL INCLUDE SQLDA;

(Read an SQL statement into SOURCE)

EXEC SQL PREPARE OBJSTATE INTO :SHORTDA
  FROM :SOURCE;

(Check for successful execution. If the value of SQLN is greater than 0, the
source statement was SELECT; use the value of SQLN to allocate and initialize
SQLDA.)

EXEC SQL DESCRIBE OBJSTATE INTO :SQLDA;
```

DROP

The DROP[5] 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.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

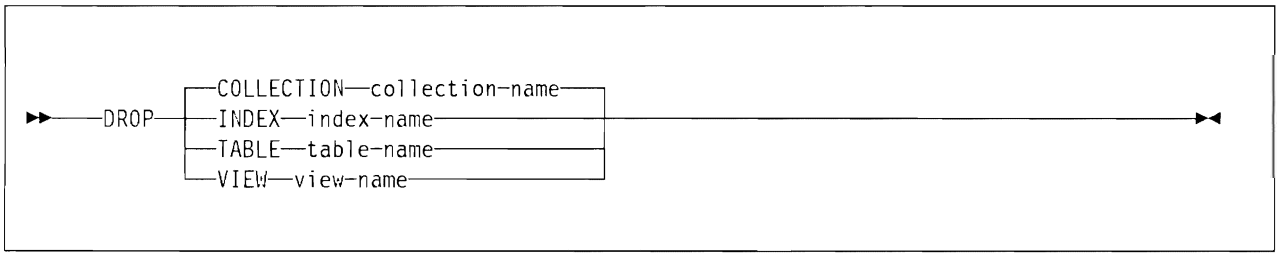
Authorization

To drop a table, view, or index, the privileges held by the authorization ID of the statement must include:

- The system authorities *OBJOPR and *DLT on the referenced library, and
- The system authority *OBJEXIST on the object to be dropped. For tables, you will also need the *OBJEXIST authority on all views, indexes, and logical files that reference the table.

To drop a collection, the privileges held by the authorization ID of the statement must include:

- The system authority *OBJEXIST on the collection to be dropped, and
- The system authority *OBJEXIST on all objects in the collection, and to any views, indexes, and logical files that reference those objects.



Description

COLLECTION *collection-name*
Identifies the collection you want to drop. All objects in the collection and the library are dropped. Any access plans that exist in programs that have dependencies on any object in the collection will be recreated when the program that contains the access plan is next invoked. If the referenced collection does not exist at that time, a negative value will be returned in the SQLCODE field of the SQLCA.

INDEX *index-name*
Identifies an index described in the catalog. Indexes can be dropped at any time except when they are in use. Any access plans that exist in programs that have dependencies on the index will be recreated when the program that contains the access plan is next run. If the referenced index does not

[5] DROP DATABASE may be used as a synonym for DROP COLLECTION to provide compatibility with a previous release.

exist at that time, a negative value will be returned in the SQLCODE field of the SQLCA.

TABLE *table-name*
Identifies the table you want to drop. The table specified must be described in the catalog and cannot be a catalog table. The specified table is deleted from the collection. All indexes, views, and logical files defined on the table are dropped. Any access plans that exist in programs that have dependencies on the table will be recreated when the program that contains the access plan is next run. If the referenced table does not exist at that time, a negative value will be returned in the SQLCODE field of the SQLCA.

VIEW *view-name*
Identifies an existing view other than a catalog view. 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. Any access plans that exist in programs that have dependencies on the view will be recreated when the program that contains the access plan is next run. If the referenced view does not exist at that time, a negative value will be returned in the SQLCODE field of the SQLCA.

Examples

Example 1: Drop table CORPDATA.DEPT.

```
DROP TABLE CORPDATA.DEPT
```

Example 2: Drop the view VDEPT.

```
DROP VIEW VDEPT
```

END DECLARE SECTION

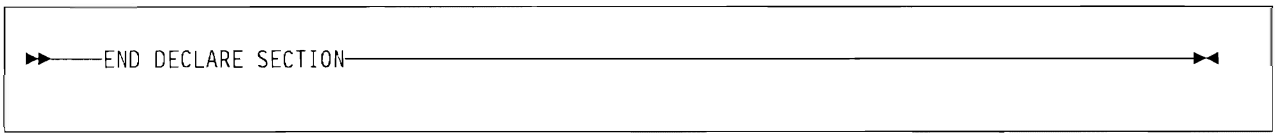
The END DECLARE SECTION statement marks the end of a host variable declare section.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.



Description

The END DECLARE SECTION statement may be coded in the application program wherever declarations can appear in accordance with the rules of the host language. It is used to indicate the end of a host variable declaration section. A host variable section starts with a BEGIN DECLARE SECTION statement described on page 53. If a BEGIN DECLARE SECTION is specified in the program, an END DECLARE SECTION is required.

Notes

If declare sections are specified in the program, only the variables declared within the declare sections can be used as host variables. When declare sections are not specified, all variables in the program are eligible for use as host variables.

Host variable declaration sections should be specified for host languages so that the source program conforms to the SAA definition of SQL.

The BEGIN DECLARE SECTION and the END DECLARE SECTION statements must be paired and may not be nested.

No other SQL statements should be included in the declare section.

Variables referenced in SQL statements should be declared in a declare section and the section should appear before the first reference to the variable.

Variables declared outside a declare section should not have the same name as variables declared within a declare section.

More than one declare section can be specified in the program.

The BEGIN DECLARE SECTION and END DECLARE SECTION statements must not be specified in RPG programs.

Example

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   .
   .
 (host variable declarations)
   .
   .
EXEC SQL END DECLARE SECTION END-EXEC.
```

EXECUTE

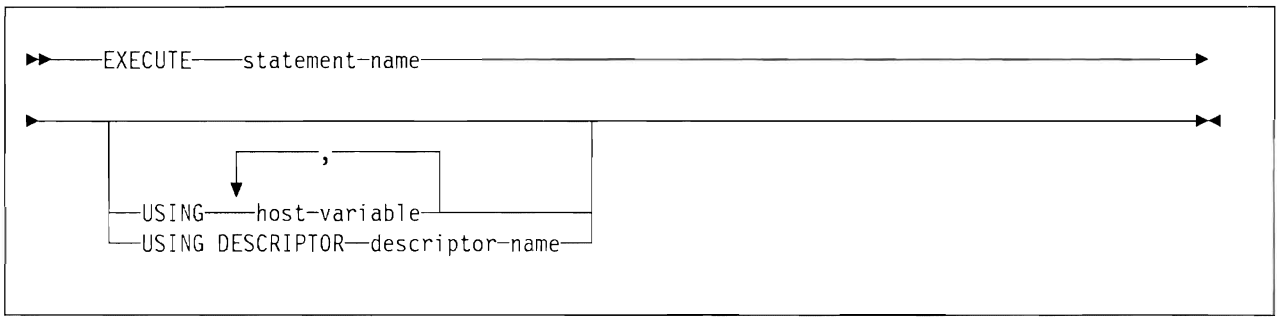
The EXECUTE statement executes a prepared SQL statement.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The authorization rules are those defined for the dynamic preparation of the SQL statement specified by EXECUTE. For example, see the description of INSERT for the authorization rules that apply when an INSERT statement is executed using EXECUTE.



Description

statement-name
Identifies the prepared statement to be executed. *statement-name* must identify a statement that was previously prepared within the unit of recovery and the prepared statement must not be a SELECT statement. The prepared statement may have been prepared in a previous unit of recovery if COMMIT HOLD or ROLLBACK HOLD have been used to preserve the prepared statement.

USING
Introduces a list of host variables whose values are substituted for the parameter markers (question marks) in the prepared statement. (For an explanation of parameter markers, see "PREPARE" on page 109.) If the prepared statement includes parameter markers, you must use USING. USING is ignored if there are no parameter markers.

host-variable
Identifies a variable that is described in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the EXECUTE statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA, and SQLDABC must be set to indicate the number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to

allow for all occurrences of SQLN. Thus, SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence, which is implementation-defined.

SQLD must be set to a value greater than or equal to 0 and less than or equal to SQLN. SQLD indicates the number of variables used in the SQLDA when executing this statement and must be the same as the number of parameter markers in the prepared statement. The nth variable described by the SQLDA corresponds to the nth parameter marker in the prepared statement. (For a description of the values that must be set in the SQLVAR occurrences of the SQLDA, see "The SQL Descriptor Area (SQLDA)" on page 133.)

Note that because RPG and COBOL do not provide the facility for setting pointers, and the SQLDA uses pointers to locate the appropriate host variables, you will have to set these pointers outside your RPG or COBOL application.

Notes

Before the prepared statement is executed, each parameter marker is effectively replaced by the value of its corresponding host variable. Each value that replaces a parameter marker must be compatible with operations applied to it during the execution of the prepared statement, as follows:

- If the parameter marker appears as the operand of an arithmetic operator, its value is converted to conform to the description of the other operand, if necessary, according to the rules described in Chapter 2. In the case of unary minus, the value is converted to double precision floating-point.
- If a parameter marker appears in place of a numeric value to be inserted in a column, its value is the number that would result if the host variable were assigned to the column, and the value must conform to the rules for assignments.
- If a parameter value is used as the operand of a comparison operator, it must be compatible with the other operand of that operator, and its length must not be greater than that of the other operand.

Example

In this example, an INSERT statement with parameter markers is prepared and executed.

```
MOVE 'INSERT INTO CORPDATA.QUOTATIONS VALUES(?,?,?,?)' TO HOLDER.

EXEC SQL PREPARE QUOTES FROM :HOLDER END-EXEC.

IF SQLCODE = 0
   PERFORM EXECUTE-INSERT
ELSE
   PERFORM ERROR-CONDITION.

EXECUTE-INSERT.
   MOVE   51 TO SUPPNO.
   MOVE  221 TO PARTNO.
   MOVE 0.30 TO PRICE.
   MOVE   50 TO QONORDER.

EXEC SQL EXECUTE QUOTES USING :SUPPNO,
   :PARTNO, :PRICE, :QONORDER
END-EXEC.
```

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement:

- Prepares an executable form of an SQL statement from a character string form of the statement.
- Executes the SQL statement.
- Destroys the executable form.

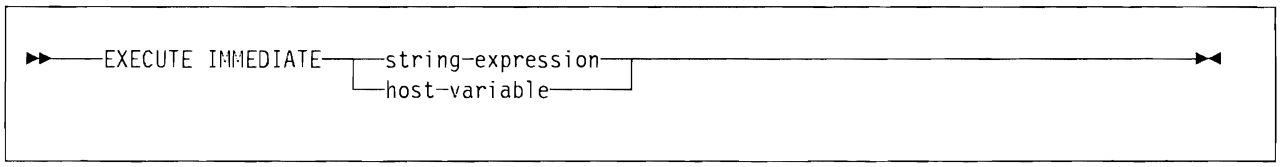
Invocation

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 dynamic preparation of the SQL statement specified by EXECUTE IMMEDIATE. For example, see "INSERT" on page 98 for the authorization rules that apply when an INSERT statement is executed using EXECUTE IMMEDIATE.

The run-time authorization ID must have the authorization required to execute the statement specified by the EXECUTE IMMEDIATE statement.



Description

string-expression
A *string-expression* is any expression that yields a character string. Note that a *string-expression* may be specified only in PL/I.

host-variable
Must identify a host variable that is described in the program in accordance with the rules for declaring character string variables.

The host variable must be of the form `:host-variable`. The form `:host-variable:indicator-variable` is not allowed.

Notes

The character string form of the statement is called a *statement-string*. The statement string is the value of the specified *string-expression* or the identified host variable.

The statement string must be one of the following SQL statements: COMMENT ON, COMMIT, CREATE COLLECTION, CREATE INDEX, CREATE TABLE, CREATE VIEW, DELETE, DROP, GRANT, INSERT, LABEL ON, LOCK TABLE, REVOKE, ROLLBACK, or UPDATE.

The statement string must not include parameter markers or references to host variables, must not begin with EXEC SQL, and must not terminate with END-EXEC or a semicolon.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed and the error condition that prevents its execution is reported in the SQLCA. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the SQLCA.

If the same SQL statement is to be executed more than once, it is more efficient to use the PREPARE and EXECUTE statements rather than the EXECUTE IMMEDIATE statement.

Example

In this COBOL example, the EXECUTE IMMEDIATE statement is used to execute a DELETE statement.

```
MOVE 'DELETE FROM QUOTATIONS WHERE
      PRICE > 1.00' TO HOLDER.

EXEC SQL EXECUTE IMMEDIATE :HOLDER END-EXEC.
```

FETCH

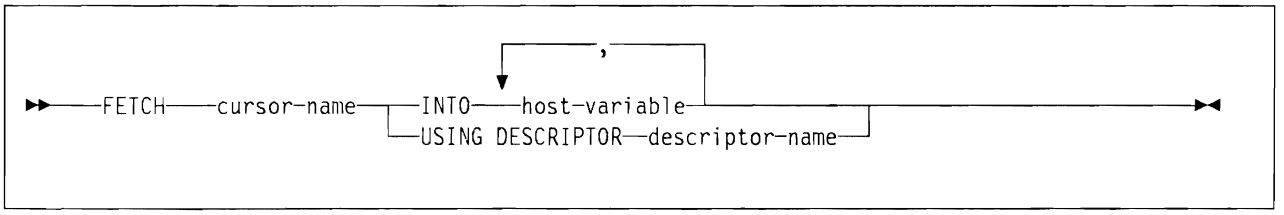
The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

The privileges held by the authorization ID of the statement must include the SELECT privilege on every table or view identified in the SELECT statement of the cursor.



Description

cursor-name
Identifies the cursor to be used in the fetch operation. The *cursor-name* must identify a declared cursor as explained in the Notes for the DECLARE CURSOR statement. When the FETCH statement is executed, the cursor must be in the open state.

If the cursor is currently positioned on or after the last row of its result table, the SQLCODE field of the SQLCA is set to +100, the cursor is positioned "after the last row," and values are not assigned to host variables.

If the cursor is currently positioned before a row, the cursor is positioned on that row and values from that row are assigned to the host variables specified by INTO or USING.

If the cursor is currently positioned on a row other than the last row, the cursor is positioned on the next row and values of that row are assigned to the host variables specified by INTO or USING.

INTO *host-variable*
If INTO is used, each host variable must identify a variable that is described in your program in accordance with the rules for declaring host variables.

The first value of a row corresponds to the first variable, the second value corresponds to the second variable, and so on.

USING DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the FETCH statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA, and SQLDABC must be set to indicate the number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to allow for all

occurrences of SQLN. Thus, SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence, which is implementation-defined.

SQLD must be set to a value greater than or equal to 0 and less than or equal to SQLN. SQLD indicates the number of variables used in the SQLDA when executing this statement and must be the same as the number of parameter markers in the prepared statement. The nth variable described by the SQLDA corresponds to the nth parameter marker in the prepared statement. (For a description of the values that must be set in the SQLVAR occurrences of the SQLDA, see "The SQL Descriptor Area (SQLDA)" on page 133.)

Note that because RPG and COBOL do not provide the facility for setting pointers, and the SQLDA uses pointers to locate the appropriate host variables, you will have to set these pointers outside your RPG or COBOL application.

The data type of a host 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. If the value is null, an indicator variable must be specified.

Assignments are made in sequence through the list. Each assignment to a variable is made according to the rules described in Chapter 2, "Language Elements" on page 5. If the number of variables is less than the number of values in the row, the SQLWARN3 field of the SQLDA is set to 'W'.

If an error occurs as the result of an arithmetic expression in the SELECT list (division by zero, overflow etc.) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned.

Notes

If the specified host variable is character and is not large enough to contain the result, 'W' is assigned to SQLWARN1 in the SQLCA. The actual length is returned in the indicator variable, if provided.

In C, if the specified host variable is null-terminated, and if the host variable is large enough to contain the result but not large enough to contain the null-terminator, then 'N' is assigned to SQLWARN1 in the SQLCA.

Example

The FETCH statement fetches the results of the SELECT statement into the program variables DNUM, DNAME, and MNUM.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO FROM CORPDATA.DEPT
   WHERE ADMRDEPT = 'A00'
   END-EXEC.

EXEC SQL OPEN C1 END-EXEC.

EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.

IF SQLCODE = 100
   PERFORM DATA-NOT-FOUND
ELSE
   PERFORM GET-REST-OF-DEPT
   UNTIL SQLCODE IS NOT EQUAL TO ZERO.

EXEC SQL CLOSE C1 END-EXEC.

GET-REST-OF-DEPT.
   EXEC SQL FETCH C1 INTO :DNUM, :DNAME, :MNUM END-EXEC.
```

GRANT

The GRANT statement grants table and view privileges to users.

Invocation

This statement can be embedded in an application program or issued interactively. It is an executable statement that can be dynamically prepared.

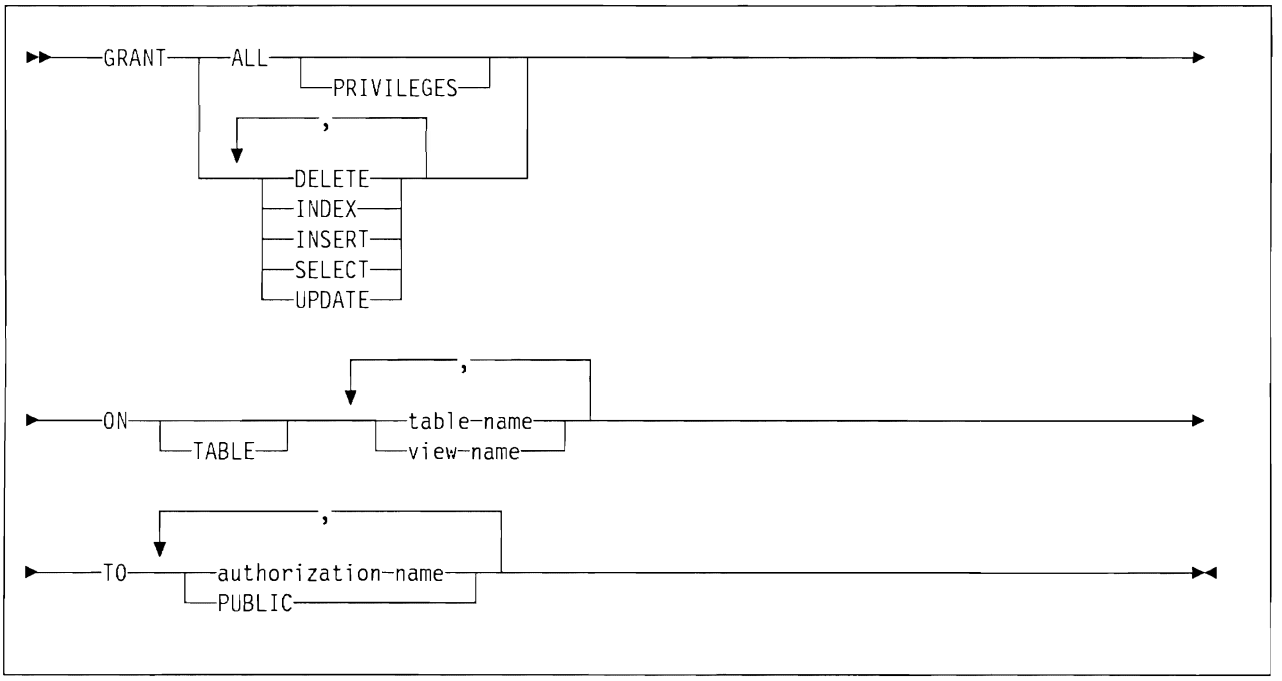
Authorization

The privileges held by the authorization ID of the statement must include:

- Every privilege specified, and ownership of the object, or
- Every privilege specified, and the system authorities *OBJMGT and *OBJOPR on the table or view.

To have every privilege you specify, you must:

- Be the owner of the object, or
- Have been granted the privileges.



Description

ALL or **ALL PRIVILEGES**

Grants all table or view privileges you currently have for all tables or views named in the ON clause. Note that granting ALL PRIVILEGES on a table or view is not the same as granting the system authority of *ALL.

If you do not use ALL, you must use one or more of the keywords in the following list.

Each keyword grants the privilege described, but only to the table or view named in the ON clause.

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.

SELECT
Grants the privilege to use the SELECT statement.

UPDATE
Grants the privilege to use the UPDATE statement.

ON or **ON TABLE**
Names the tables or views on which you are granting the privileges.

TO
Indicates to whom the privileges are granted.

authorization-name
Lists one or more authorization IDs.

PUBLIC
Grants the privileges to all users that have no privately granted privilege on the table or view.

Notes

Because the GRANT and REVOKE statements assign and remove AS/400 system security authorities for SQL objects, each SQL privilege can be said to correspond to specific AS/400 system rights. The tables that follow outline this correspondence: the left column lists all grantable SQL privileges, and the right column, or columns, list the equivalent AS/400 system object and data rights for views and for tables. System data rights are assigned to and removed from either the table specified or, if a view is specified, to the base table or tables on which the view is specified and on which the view is dependent.

Table 3. Privileges Granted to Tables	
SQL Privilege	**Corresponding AS/400 System Rights when Granting to a Table**
ALL (GRANT of ALL only grants those privileges you currently have)	*OBJMGT *OBJOPR *ADD *DLT *READ *UPD
DELETE	*OBJOPR *DLT
INDEX	*OBJMGT *OBJOPR
INSERT	*OBJOPR *ADD
SELECT	*OBJOPR *READ
UPDATE	*OBJOPR *UPD

Table 4. Privileges Granted to Views		
SQL Privilege	**Corresponding AS/400 System Rights Granted to View**	**Corresponding AS/400 System Rights Granted to Base Table**
ALL (GRANT of ALL only grants those privileges you currently have)	*OBJOPR	*ADD *DLT *READ *UPD
DELETE	*OBJOPR	*DLT
INDEX	N/A	N/A
INSERT	*OBJOPR	*ADD
SELECT	*OBJOPR	*READ
UPDATE	*OBJOPR	*UPD

If a view is read-only, only the SQL authority of SELECT can be granted on it. If inserts are not allowed on a view, the SQL authority of INSERT cannot be granted on it.

Example

Grant SELECT privileges on table CORPDATA.EMP to user PULASKI.

```
GRANT SELECT
  ON CORPDATA.EMP
  TO PULASKI
```

INCLUDE

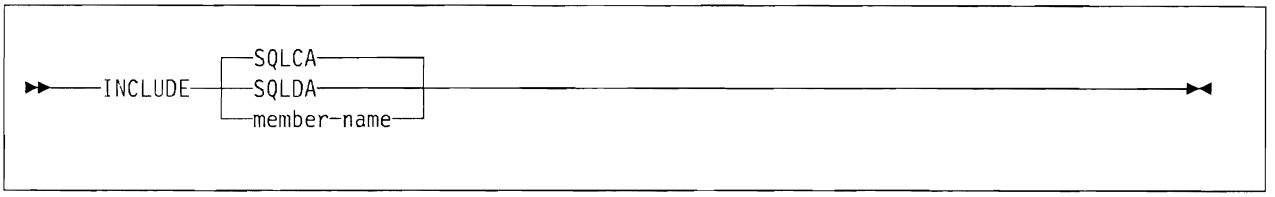
The INCLUDE statement inserts declarations into a source program.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.



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.

The SQLCA may be specified for COBOL, C, and PL/I. If the SQLCA is not specified, then the variable SQLCODE must appear in the program. The variable SQLCODE must be an elementary item and must have the attribute of *integer*.

The SQLCA must not be specified for RPG programs. The variable SQLCOD cannot be specified in an RPG program.

For a description of the SQLCA, see "SQL Communication Area (SQLCA)" on page 127.

SQLDA
Indicates the description of an SQL Descriptor Area (SQLDA) is to be included. It must not be specified in a COBOL or RPG program. For a description of the SQLDA, see "The SQL Descriptor Area (SQLDA)" on page 133.

member-name
Names a member to be included from the file specified on the INCFILE keyword of the CRTSQL*xxx* (where *xxx* is RPG, CBL, C, or PLI) command.

The member may contain any host language source statements and any SQL statements other than an INCLUDE statement.

Notes

When your program is precompiled, the INCLUDE statement is replaced by source statements. Thus the INCLUDE statement should be specified at a point in your program such that the resulting source statements are acceptable to the compiler.

Example

Include an SQLCA in a program.

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

INSERT

The INSERT statement inserts rows into a table or view. Inserting a row into a view inserts the row into the table on which the view is based.

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 INSERT privilege on the specified table or view.

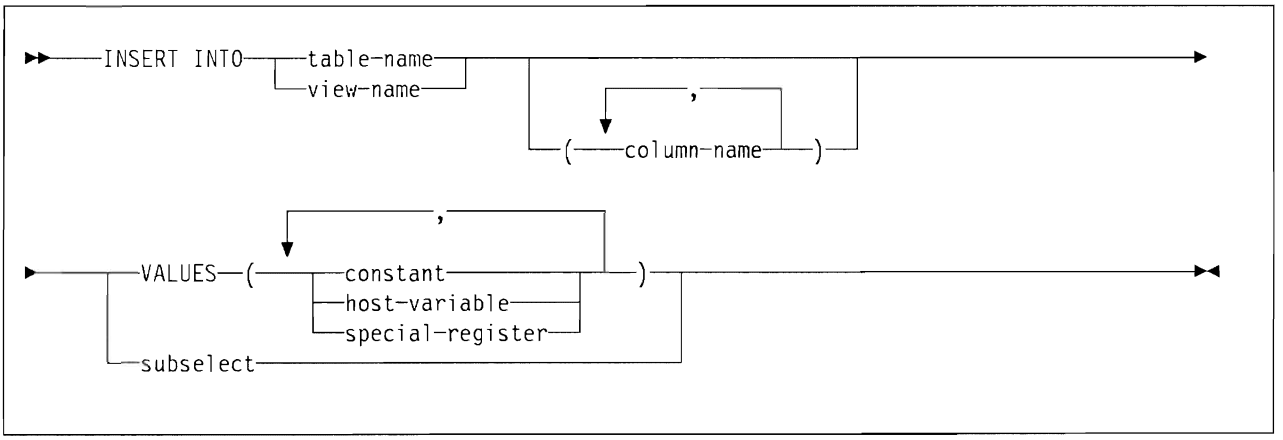
You have the INSERT privilege on a table if any of the following apply:

- You are the owner of the table
- You have been granted the INSERT privilege on the table
- You have been granted the system authorities *OBJOPR and *ADD on the table.

You have the INSERT privilege on a view if any of the following apply:

- You have been granted the INSERT privilege on the view
- You created the view, you had the INSERT privilege on its base table at that time, and you still have that INSERT privilege
- You have been granted the system authorities *OBJOPR and *ADD on the base table.

If a subselect is specified. the privileges held by the authorization ID of the statement must also include the SELECT privilege on every table or view identified in the subselect. Refer to SELECT to determine when you have the SELECT privilege.



Note: Refer to Chapter 4, "Queries" on page 39 for the syntax of *subselect*.

Description

INTO *table-name* or *view-name*

Names the table or view into which an insertion is to be made. The table or view must be described in the catalog, but it must not be a catalog table or any of the following types of view:

- A read-only view (for a description, see "CREATE VIEW" on page 69)
- A view of a catalog table

The following types of views are *not* allowed unless a column name has been specified:

- A view with a column that is derived from a constant or an arithmetic expression.
- A view with two columns derived from the same column of the underlying table.

column-name

Lists the names of one or more columns for which you provide insert values. You may name the columns in any order. Each must belong to the table or view named, and you may not name the same column more than once. The column names must not be qualified.

If you omit the column list, you are implicitly using a list of all the columns in the order they exist in the table or view.

The implicit column list is established at create program time if the referenced table or view exists at create program time. Otherwise, the implicit column list is established at the first successful run of the INSERT statement. Hence an INSERT statement embedded in an application program does not use any columns that might have been added to the table or view after create program time.

VALUES

Introduces one row of values to be inserted. The values of the row are the values of the keywords, constants, or host variables specified in the clause.

Each host variable you name must be described in your program in accordance with the rules for declaring host variables.

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*, see Chapter 2. For a description of *special-register*, see "Special Registers" on page 18.

subselect

Inserts the rows of the result table of a subselect. There may be one, more than one, or none. If there is none, SQLCODE is set to +100.

The base object of the INSERT, and the base object 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.

Insert Rules

A maximum of 4096 rows may be inserted in any single INSERT operation when COMMIT(*ALL) or COMMIT(*CHG) has been specified.

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 the default value of the column. Columns without a default value must be included in the column list. Similarly, if you insert into a view, the default 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 have default values.
- *Data Types*: The data type of the values to be inserted must be compatible with the data type defined for the corresponding columns.
- *Length*: If the insert value of a column is a number, the column must be a numeric column with the capacity to represent the integral part of the number. If the insert value of a column is a string, the column must be a string column with a length attribute at least as great as the length of the string.
- *Assignment*: Insert values are assigned to columns in accordance with the assignment rules described in Chapter 2.
- *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.

Notes

An INSERT statement may be used to insert rows that do not conform to the definition of the view. These rows will not appear in the view, but are inserted into the base table of the view.

If an error occurs during the execution of an INSERT statement and COMMIT(*ALL) or COMMIT(*CHG) was specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of recovery made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

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, and LOCK TABLE statements.

Examples

Example 1: Insert values into table CORPDATA.EMP.

```
INSERT INTO CORPDATA.EMP
  VALUES ('000205','MARY','T','SMITH','D11','2866',
          810810,42,16,'F',550522,16345)
```

Example 2: Load the temporary table SMITH.TEMPEMPL with data from table CORPDATA.EMP.

```
INSERT INTO SMITH.TEMPEMPL
  SELECT *
  FROM CORPDATA.EMP
```

Example 3: Load the temporary table SMITH.TEMPEMPL with data from Department D11 from CORPDATA.EMP.

```
INSERT INTO SMITH.TEMPEMPL
  SELECT *
  FROM CORPDATA.EMP
  WHERE WORKDEPT='D11'
```

LABEL ON

The LABEL ON statement adds or replaces labels in the catalog descriptions of tables, views, or columns.

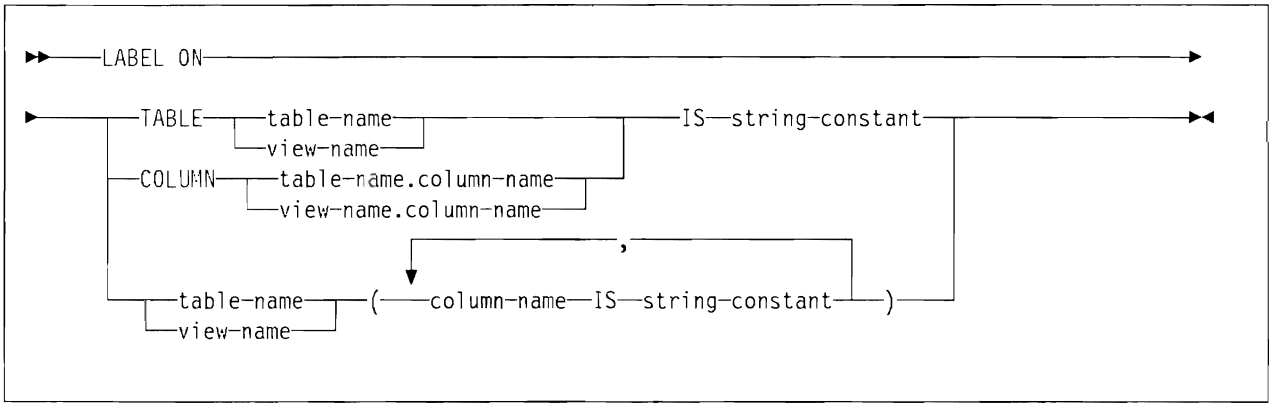
Invocation

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:

- The system authority *READ on the library containing the table or view, and
- Ownership of the table or view, or the system authorities of both *OBJOPR and *OBJMGT on the referenced table or view.



Description

TABLE
Indicates that the label is for a table or a view. Labels on tables or views are implemented as AS/400 system object text.

table-name or *view-name*
Must identify a table or view described in the local catalog.

COLUMN
Indicates that the label is for a column. Labels on columns are implemented as AS/400 system column headings, and can therefore be used when displaying or printing query results.

table-name.column-name or *view-name.column-name*
Is the name of the column, qualified by the name of the table or view in which it appears. The column named must be described in the catalog.

IS
Introduces the label you want to provide.

string-constant
Can be any SQL character string constant of up to 30 bytes in length for tables and views, or 20 bytes in length for columns. The constant may contain double-byte characters as well as EBCDIC characters.

Example

Enter a label on the DEPTNO column of table CORPDATA.DEPT.

```
LABEL ON COLUMN CORPDATA.DEPT.DEPTNO
  IS 'DEPARTMENT NUMBER'
```

LOCK TABLE

The LOCK TABLE statement acquires a shared or exclusive lock on the named table.

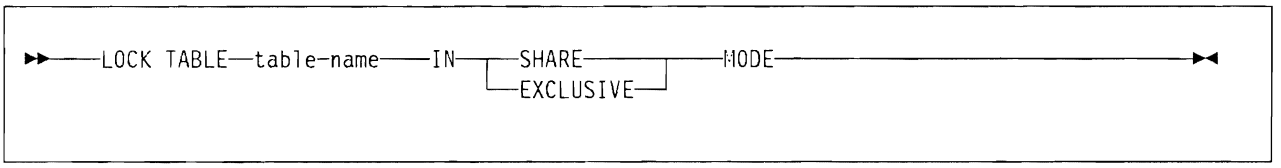
Invocation

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:

- Ownership of the table, or
- Any SQL privilege for the table, or
- The system authority *OBJOPR on the table.



Description

table-name
Names the table to be locked. The table must be a base table described in the catalog, but not a catalog table.

IN SHARE MODE
Acquires a shared lock (*SHRNUP) for the application process in which the statement is executed. The lock prevents concurrent application processes from executing any but read-only operations on the named table.

IN EXCLUSIVE MODE
Acquires an exclusive lock (*EXCL) for the application process in which the statement is executed. The lock prevents concurrent application processes from executing any operations at all on the identified table.

The lock is acquired when the LOCK TABLE statement is executed.

The lock acquired can be released in three ways:

1. By the termination of the unit of recovery unless the unit of recovery is terminated by SQL statements COMMIT HOLD or ROLLBACK HOLD, or the CL commands COMMIT or ROLLBACK.
2. By the ending of the first SQL program in the program stack. An SQL program is a program which has issued an embedded SQL statement.
3. By issuing the CL command DLCOBJ to unlock the table.

Because the statement is synchronous, conflicting locks already held by other application processes will cause your application to wait up to the default wait time.

Example

Obtain a lock on the table CORPDATA.EMP. Do not allow other programs either to read or update the table.

```
LOCK TABLE CORPDATA.EMP IN EXCLUSIVE MODE
```

OPEN

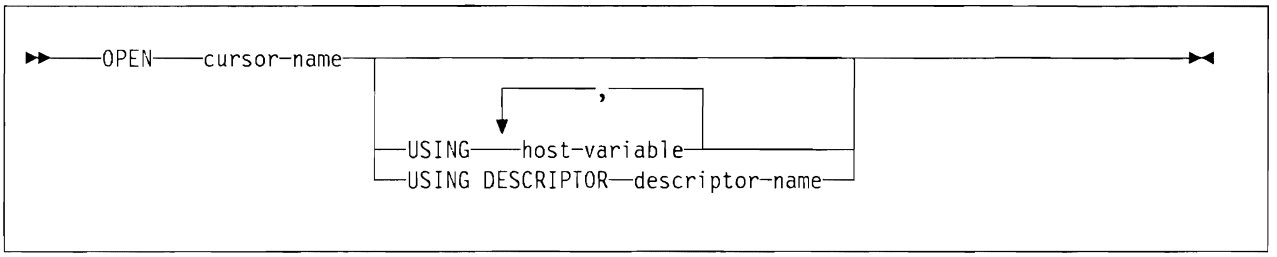
The OPEN statement opens a cursor so that it can be used to fetch rows from its result table.

Invocation

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 72 for the authorization required to use a cursor.



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 executed, the cursor must be in the closed state.

The SELECT statement associated with the cursor is either:

- the *select-statement* specified in the DECLARE CURSOR statement, or
- the prepared *select-statement* identified by the *statement-name* specified in the DECLARE CURSOR statement. If the identified SELECT statement has not been successfully prepared, the cursor cannot be successfully opened.

The result table of the cursor is derived by evaluating the 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 specified in the SELECT statement or 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 state of the cursor is effectively "after the last row".

USING
Introduces a list of host variables whose values are substituted for the parameter markers (question marks) of a prepared statement. (For an explanation of parameter markers, see "PREPARE" on page 109.) 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.

host-variable
Identifies a variable described in the program in accordance with the rules for declaring host variables. The number of variables must be the same as the number of parameter markers in the prepared statement. The *n*th variable corresponds to the *n*th parameter marker in the prepared statement.

DESCRIPTOR *descriptor-name*
Identifies an SQLDA that must contain a valid description of zero or more host variables.

Before the OPEN statement is executed, the value of SQLN must be set to indicate how many occurrences of SQLVAR are provided in the SQLDA, and SQLDABC must be set to indicate the number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to allow for all the occurrences of SQLN. Thus, SQLDABC must be greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence, which is implementation-defined.

SQLD must be set to a value greater than or equal to 0 and less than or equal to SQLN. SQLD indicates the number of variables used in the SQLDA when executing this statement and must be the same as the number of parameter markers in the prepared statement. The nth variable described by the SQLDA corresponds to the nth parameter marker in the prepared statement. (For a description of the values that must be set in the SQLVAR occurrences of the SQLDA, see "The SQL Descriptor Area (SQLDA)" on page 133.)

Note that because RPG and COBOL do not provide the facility for setting pointers, and the SQLDA uses pointers to locate the appropriate host variables, you will have to set these pointers outside your RPG or COBOL application.

When the SELECT statement of the cursor is evaluated, each parameter marker is effectively replaced by the value of its corresponding host variable. Each value that replaces a parameter marker must be compatible with operations applied to it during the execution of the prepared statement. For example,

- If the parameter marker appears as the operand of an arithmetic operator, the parameter value must be a number. Its value is converted to conform to the description of the other operand, if necessary, according to the rules described in Chapter 2.
- If the other operand is a column, the value of the parameter is the number that would result if the host variable were assigned to the column, and the value must conform to the rules for assignments described in Chapter 2.
- If a parameter value is used as the operand of a comparison operator, it must be compatible with the other operand of that operator, and its length must not be greater than that of the other operand. In the case of the BETWEEN and IN predicates, this "other operand" is the first operand that is not specified with a parameter marker.

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 recovery by executing a COMMIT or ROLLBACK statement without a HOLD option.

A cursor can also be in the closed state because a CLOSE statement was executed.

To retrieve rows from the result table of a cursor, you must execute a FETCH statement when the cursor is open. The only way to change the state of a cursor from closed to open is to execute an OPEN statement.

Effect of temporary tables: If the result table of a 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 transferred to a temporary table during the execution of the OPEN statement. When a temporary table is used, the results of a program can differ in these two ways:

- An error can occur during OPEN that would otherwise not occur until some later FETCH statement.
- The INSERT, UPDATE, and DELETE statements are not allowed while the cursor is open.

Conversely, if a temporary table is not used, INSERT, UPDATE, and DELETE statements executed while the cursor is open can affect the result table if issued from the same program. Your result table can also be affected by operations executed by your own unit of recovery, and the effect of such operations is not always predictable. For example, if cursor C is positioned on a row of its result table defined as SELECT * FROM T, and you insert a row into T, the effect of that insert on the result table is not predictable because its rows are not ordered. Thus a subsequent FETCH C may or may not retrieve the new row of T.

Example

The OPEN statement places the cursor at the beginning of the rows to be fetched.

```
EXEC SQL DECLARE C1 CURSOR FOR
   SELECT DEPTNO, DEPTNAME, MGRNO FROM CORPDATA.DEPT
   WHERE ADMRDEPT = 'A00'
   END-EXEC.

EXEC SQL OPEN C1 END-EXEC.
```

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 life of a prepared statement extends to one of the following:

- The end of the application program, or
- Until another PREPARE statement with the same statement name has been issued by the same instance of the program in the program stack (in the case of recursive program calls), or
- Until a COMMIT (no HOLD), or ROLLBACK (no HOLD) is issued.

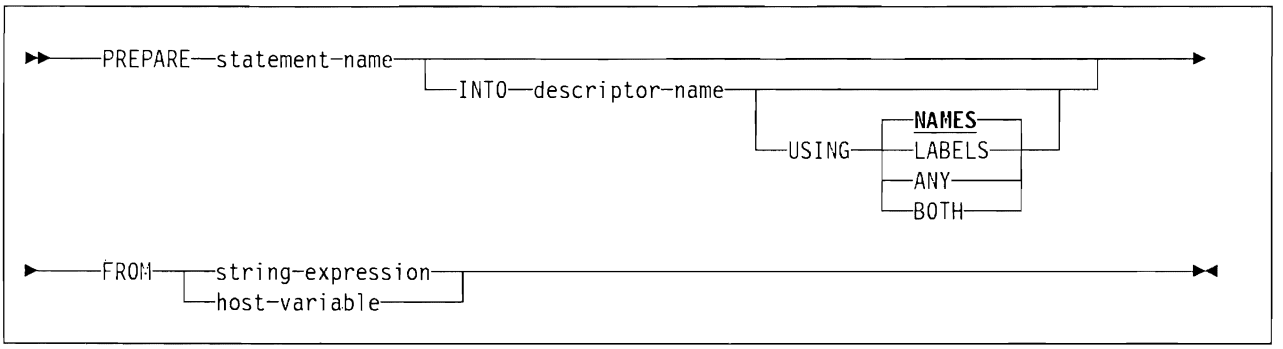
Invocation

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 dynamic preparation of the SQL statement specified by the PREPARE statement. For example, see Chapter 4, "Queries" on page 39 for the authorization rules that apply when a SELECT statement is prepared.

Authorization is not completely checked when DROP COLLECTION, DROP TABLE or DROP VIEW statements are prepared. The system authority *OBJEXIST on all objects in the collection is not required to prepare a DROP COLLECTION statement. The system authority *OBJEXIST is not required on all views, indexes, and logical files that reference the table in a DROP TABLE statement. The system authority of *OBJEXIST is not required on all views that reference the view in a DROP VIEW statement. The *OBJEXIST authority is checked when the EXECUTE statement is executed.



Description

statement-name

Names the prepared statement. If the name identifies an existing prepared statement, that prepared statement is destroyed, unless the statement is prepared in another instance of the same program or another program. The name must not identify a prepared statement that is the SELECT statement of an open cursor of this instance of the program.

INTO

If you use INTO, and the PREPARE statement is successfully executed, information about the prepared statement is placed in the SQLDA specified by the descriptor-name. Thus the PREPARE statement:

```
EXEC SQL PREPARE S1 INTO SQLDA FROM V1;
```

is equivalent to:

```
EXEC SQL PREPARE S1 FROM V1;
EXEC SQL DESCRIBE S1 INTO SQLDA;
```

See "DESCRIBE" on page 79 for an explanation of the information that is placed in the SQLDA.

descriptor-name

Is SQLDA or the name of an SQLDA.

USING

Indicates what value to assign to each SQLNAME variable in the SQLDA when INTO is used. If the requested value does not exist, SQLNAME is set to length 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 will be needed to accommodate the additional information. To specify this expansion of the SQLVAR array, set SQLN to 2*n* on the PREPARE statement (where *n* is the number of columns in the result table). Then, on any later FETCH statement, set SQLN to *n*. 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.

FROM

Introduces the statement string. The statement string is the value of the specified *string-expression* or the identified *host-variable*.

string-expression

Is any expression that yields a character string. String expressions are only allowed in PL/I.

host-variable

Must identify a host variable that is described in the program in accordance with the rules for declaring character string variables.

The host variable must be of the form *:host-variable*. The form *:host-variable:indicator-variable* is not allowed.

Notes

Rules for statement strings: The statement string must be one of the following SQL statements: COMMENT ON, COMMIT, CREATE COLLECTION, CREATE INDEX, CREATE TABLE, CREATE VIEW, DELETE, DROP, GRANT, INSERT, LABEL ON, LOCK TABLE, REVOKE, ROLLBACK, or UPDATE.

The statement string may also be a *select-statement*. For information on the *select-statement*, see "select-statement" on page 47.

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 executed. A parameter marker is a question mark (?) that appears where a host variable could appear if the statement string were a static SQL statement. For an explanation of how parameter markers are replaced by values, see "OPEN" on page 106 and "EXECUTE" on page 85.

Rules for parameter markers:

- Parameter markers must not appear:
  - In a select list (SELECT ? is invalid)
  - As an operand of the concatenation or substring operator
  - As both operands of a single arithmetic or comparison operator (WHERE ? = ? is invalid)
  - As an operand of a unary minus
- 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.
- 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.

When a PREPARE statement is executed, the statement string is parsed and checked for errors. If the statement string is invalid, a prepared statement is not created and a negative value is returned in the SQLCODE field of the SQLCA.

Prepared statements can be referred to in the following kinds of statements, with the restrictions shown:

In...	The prepared statement ...
DESCRIBE	has no restrictions
DECLARE CURSOR	must be SELECT
EXECUTE	must *not* be SELECT

A prepared statement can be executed many times. Indeed, if a prepared statement is not executed more than once and does not contain parameter markers, it is more efficient to use the EXECUTE IMMEDIATE statement rather than the PREPARE and EXECUTE statements.

All prepared statements created by a unit of recovery are destroyed when the unit of recovery is terminated, unless COMMIT HOLD or ROLLBACK HOLD was used.

A prepared statement can only be referenced in the same instance of the program in the program stack.

Example

In this COBOL example, an INSERT statement with parameter markers is prepared.

```
MOVE 'INSERT INTO CORPDATA.QUOTATIONS VALUES(?,?,?,?)' TO HOLDER.

EXEC SQL PREPARE QUOTES FROM :HOLDER END-EXEC.
```

REVOKE

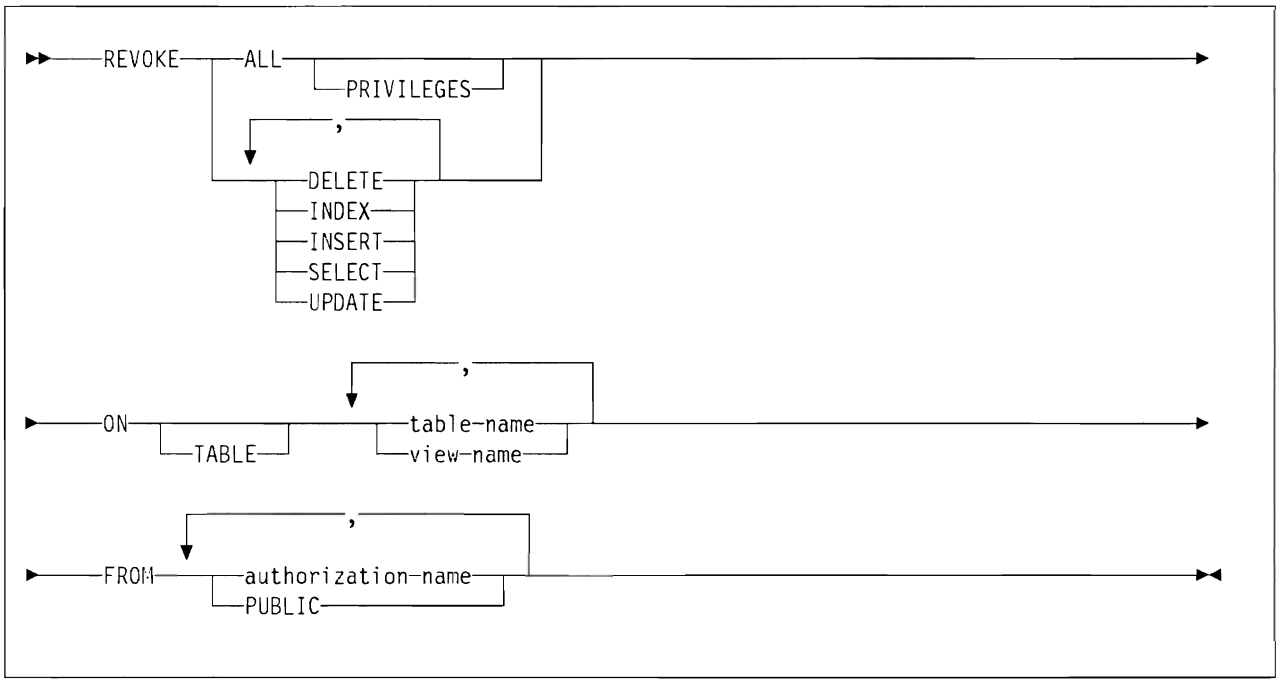
The REVOKE statement removes privileges on one or more tables or views.

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 *OBJMGT authority and the privileges you are revoking.



ALL or **ALL PRIVILEGES**
Revokes all privileges listed below for the specified tables or views.

If you do not use ALL, you must use one or more of the keywords listed below. Each keyword revokes the privilege described, but only as it applies to the tables or views named in the ON clause. The same keyword must not be specified more than once. For information on how these privileges relate to the AS/400 system object and data rights, see "Notes" on page 94.

DELETE
Revokes the privilege to use the DELETE statement.

INDEX
Revokes the privilege to use the CREATE INDEX statement.

INSERT
Revokes the privilege to use the INSERT statement.

SELECT
Revokes the privilege to use the SELECT statement.

UPDATE
Revokes the privilege to use the UPDATE statement.

ON or **ON TABLE**
Names one or more tables or views on which you are revoking the privileges. The list may be a list of unique table names or view names, or a combination of the two.

FROM
Identifies from whom the privileges are revoked.

authorization-name
Lists one or more authorization IDs. Do not specify the same *authorization-name* more than once.

PUBLIC
Revokes the specified privileges from PUBLIC.

System Object and Data Rights: When revoking authorities to a table, the *OBJOPR object rights are revoked only when all system data rights to that table have been revoked. For a view, these system object rights will only be revoked when all system data rights to the table or tables on which the view is dependent have been revoked.

When revoking authorities to a view, the system data rights will only be revoked from a base table if the specified user does not have the system authority of *OBJOPR to the base table.

Read-only Views: If inserts, updates, or deletes are not allowed on a view, then the respective SQL authority of INSERT, UPDATE, or DELETE cannot be revoked from the view.

Multiple Grants: If you granted the same privilege to the same user more than once, revoking that privilege from that user nullifies all those grants.

Examples

Example 1: Revoke SELECT privileges on table CORPDATA.EMP from user PULASKI.

```
REVOKE SELECT
  ON TABLE CORPDATA.TEMPL
  FROM PULASKI
```

Example 2: Revoke update privileges on table CORPDATA.EMP, previously granted to all local users. Note that grants to specific users are not affected.

```
REVOKE UPDATE
  ON TABLE CORPDATA.TEMPL
  FROM PUBLIC
```

Example 3: Revoke all privileges on table CORPDATA.EMP, from users KWAN and THOMPSON.

```
REVOKE ALL
  ON TABLE CORPDATA.TEMPL
  FROM KWAN,THOMPSON
```

ROLLBACK

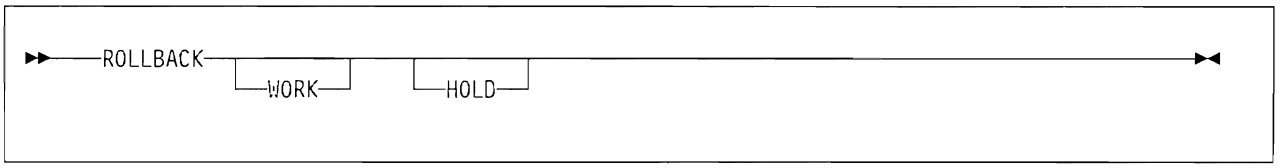
The ROLLBACK statement is used to terminate a unit of recovery and back out the database changes that were made by that unit of recovery.

Invocation

This statement can be embedded in an application program or it can be issued interactively. It is an executable statement that can be dynamically prepared.

Authorization

None required.



Description

The unit of recovery in which the ROLLBACK statement is executed is terminated and a new unit of recovery is initiated. All changes made by INSERT, UPDATE, and DELETE statements executed during the unit of recovery are backed out.

All locks acquired by the unit of recovery are released. All cursors that were opened during the unit of recovery are closed. All statements that were prepared during the unit of recovery are destroyed, and any cursors associated with the prepared statements are invalidated.

WORK
ROLLBACK WORK has the same effect as ROLLBACK. SQL/400 accepts the keyword WORK for compatibility with other database products.

HOLD
Indicates a hold on resources. If specified, currently open cursors are not closed, prepared SQL statements are preserved, and all resources acquired during the unit of recovery, except locks on the rows of tables, are held. Locks on specific rows acquired during the transaction, however, are released. If HOLD is omitted, open cursors are closed, prepared SQL statements discarded, and held resources released. At the end of a ROLLBACK, the cursor position is the same as it was at the start of the unit of recovery.

Notes

A unit of recovery (see "Application Processes, Concurrency, and Recovery" on page 3 for description) may include the processing of up to 4096 rows, including rows retrieved during a SELECT or FETCH statement[6], and rows inserted, deleted, or updated as part of INSERT, DELETE, and UPDATE

[6] Unless you specified COMMIT(*CHG), in which case these rows are not included in this total.

operations.[7] The commit and rollback operations do not affect any data definition statements, and these statements are not, therefore, allowed in an application program that also specifies COMMIT(*CHG) or COMMIT(*ALL). The data definition statements are:

- COMMENT
- CREATE COLLECTION
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- DROP COLLECTION
- DROP INDEX
- DROP TABLE
- DROP VIEW
- GRANT
- LABEL
- REVOKE

Commitment control is implicitly started by SQL, if necessary, using the system CL command STRCMTCTL. The lock level used is based on the COMMIT option specified on either the CRTSQL*xxx* (where *xxx* is RPG, CBL, C, or PLI) or the STRSQL command.

A ROLLBACK is automatically performed when:

1. An application process ends without a final COMMIT being issued.
2. A failure occurs that prevents the application from completing its work (such as a power failure).

If, within a unit of work, a CLOSE is followed by a ROLLBACK, all intervening changes are backed out. The CLOSE itself is not backed out and the file is not reopened.

Example

Delete the alterations made since the last commit point or rollback.

```
ROLLBACK WORK
```

[7] This limit also includes any records accessed or changed through files opened under commitment control through high-level language file processing.

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 does not assign values to the host variables.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

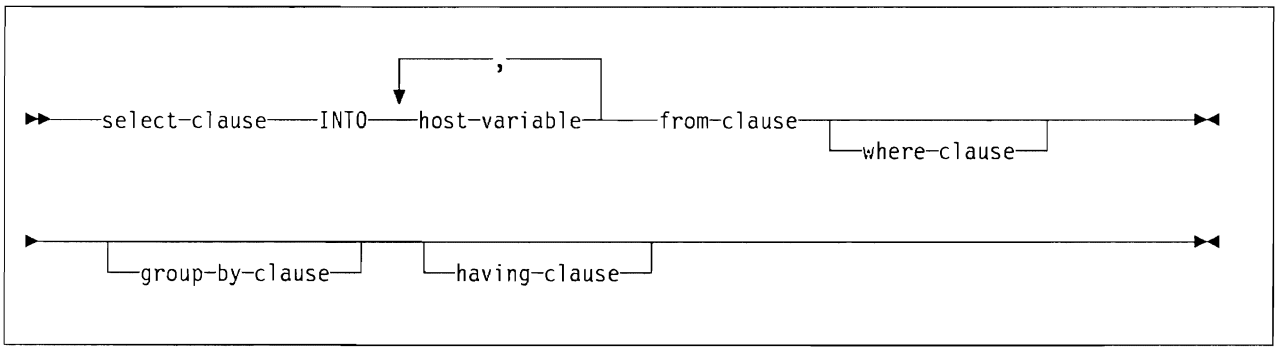
To use SELECT INTO, the privileges held by the authorization ID of the statement must include the SELECT privilege on every table and view identified in the statement.

You have the SELECT privilege on a table if any of the following apply:

- You are the owner of the table
- You have been granted the SELECT privilege on the table
- You have been granted the system authorities *OBJOPR and *READ on the table.

You have the SELECT privilege on a view if any of the following apply:

- You have been granted the SELECT privilege on the view
- You created the view, you had the SELECT privilege on its base table when the view was created, and you still have that SELECT privilege
- You have been granted the system authority *OBJOPR on the view and the system authority *READ on the base table.



Description

See Chapter 4, "Queries" on page 39 for a description of the *select-clause*, *from-clause*, *where-clause*, *group-by-clause*, and *having-clause*.

The *from-clause* must not identify a view that includes a *group-by-clause* or a *having-clause*. Note too that the grouping, as specified by the *group-by-clause*, strongly implies a result table of more than one row, and that a *having-clause* is probably needed to reduce the table to at most one row.

INTO
Introduces a list of host variables.

host variable

Names a structure or variable that is described in the program under the rules for declaring host structures and variables. A reference to a structure is replaced by a reference to each of its variables before the statement is executed.

The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on. If the number of host variables is less than the number of column values, the value 'W' is assigned to the SQLWARN3 field of the SQLCA. (See "SQL Communication Area (SQLCA)" on page 127.)

The data type of a variable must be compatible with the value assigned to it. If the value is numeric, the variable must have the capacity to represent the integral part of the value. 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 2. Assignments are made in sequence through the list.

If an error occurs as the result of an arithmetic expression in the SELECT list of a SELECT statement (division by zero, or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. If you do not provide an indicator variable, or some other type of error occurs, processing of the statement terminates when the error is encountered.

If an error occurs, no value is assigned to the host variable or to later variables, 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 are assigned to all host variables, but the row that is the source of the values is undefined and not predictable.

Examples

Example 1: Put the maximum salary in CORPDATA.EMP into the host variable MAXSALARY.

```
EXEC SQL SELECT MAX(SALARY)
  INTO :MAXSALRY
  FROM CORPDATA.EMP;
```

Example 2: Put the row for employee 528671, from CORPDATA.EMP, into the host structure EMPREC.

```
EXEC SQL SELECT * INTO :EMPREC
  FROM CORPDATA.EMP
  WHERE EMPNO = '528671'
END-EXEC.
```

UPDATE

The UPDATE statement updates the values of specified columns in rows of a table or view. Updating a row of a view updates a row of its base table.

The forms of this statement are:

- The searched UPDATE form is used to update one or more rows (optionally determined by a search condition).
- The positioned UPDATE form is used to update exactly one row (as determined by the current position of a cursor).

Invocation

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 forms are executable statements that can be dynamically prepared.

Authorization

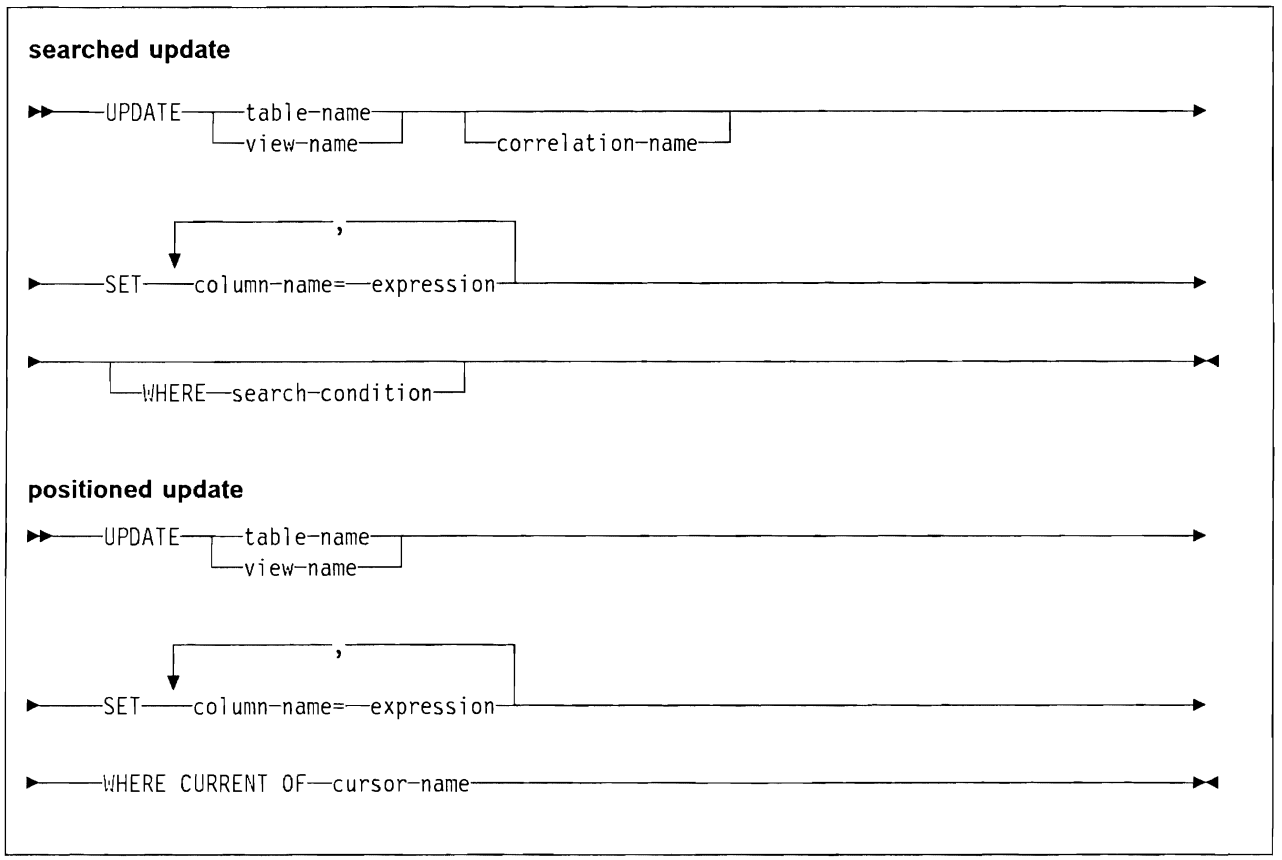
The privileges held by the authorization ID of the statement must include the UPDATE privilege on the specified table or view.

You have the UPDATE privilege on a table if any of the following apply:

- You are the owner of the table
- You have been granted the UPDATE privilege on the table
- You have been granted the system authorities *OBJOPR and *UPD on the table.

You have the UPDATE privilege on a view if any of the following apply:

- You have been granted the UPDATE privilege on the view
- You created the view, you had the UPDATE privilege on its base table at that time, and you still have that UPDATE privilege.
- You have been granted the system authorities *OBJOPR and *UPD on the base table.



Description

table-name or *view-name*
Is the name of the table or view to be updated. The name must identify a table or view described in the catalog, but not a catalog table, a view of a catalog table, or a read-only view. (For an explanation of read-only views, see "CREATE VIEW" on page 69.)

correlation-name
May be used within *search-condition* to designate the table or view. (For an explanation of *correlation-name*, see "Correlation Names" on page 19.)

SET
Introduces a list of column names and values. The column names must not be qualified, and a column must not be specified more than once.

In a cursor-controlled update, each column name in the list must also appear in the FOR UPDATE OF clause of the SELECT statement of the identified cursor, unless FOR UPDATE OF and ORDER BY were not specified.

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.

expression
Indicates the new value of the column. The *expression* is any expression of the type described in Chapter 2. It must not include a column function.

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.

WHERE

Introduces a condition that indicates what rows are updated. 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 updated.

search-condition

Is any search condition as described in Chapter 2. Each *column-name* in the search condition must name a column of the table or view.

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.

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 72.

The table or view named must also be named in the FROM clause of the SELECT statement of the cursor, and the result table of the cursor must not be read-only. (For an explanation of read-only result tables, see "DECLARE CURSOR" on page 72.)

When the UPDATE statement is executed, the cursor must be positioned on a row; that row is updated.

Notes

A maximum of 4096 rows may be updated in any single UPDATE operation when COMMIT(*ALL) or COMMIT(*CHG) has been specified.

Update values are assigned to columns under the assignment rules described in Chapter 2.

If the update value is ...	Then the column must ...
a number	be a numeric column with the capacity to represent the integral part of the number.
a character string	be a character string column with a length attribute that is not less than the length of the string.

If an update value violates any constraints, or if any other error occurs during the execution of the UPDATE statement, and COMMIT(*ALL) or COMMIT(*CHG) was specified, all changes made during the execution of the statement are backed out. However, other changes in the unit of recovery made prior to the error are not backed out. If COMMIT(*NONE) is specified, changes are not backed out.

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.

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 127.)

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, and LOCK TABLE statements.

Examples

Example 1: Change employee 000190's telephone number to 3565 in CORPDATA.EMP.

```
UPDATE CORPDATA.EMP
  SET PHONENO='3565'
  WHERE EMPNO='000190'
```

Example 2: Increase the job code by 10 for members of Department D11.

```
UPDATE CORPDATA.EMP
  SET JOBCODE = JOBCODE + 10
  WHERE WORKDEPT='D11'
```

Example 3: Change the project end date for project number AD3111 to 13 July 1984.

```
UPDATE CORPDATA.PROJ
  SET PRENDATE = '1984-07-13'
  WHERE PROJNO = 'AD3111'
```

WHENEVER

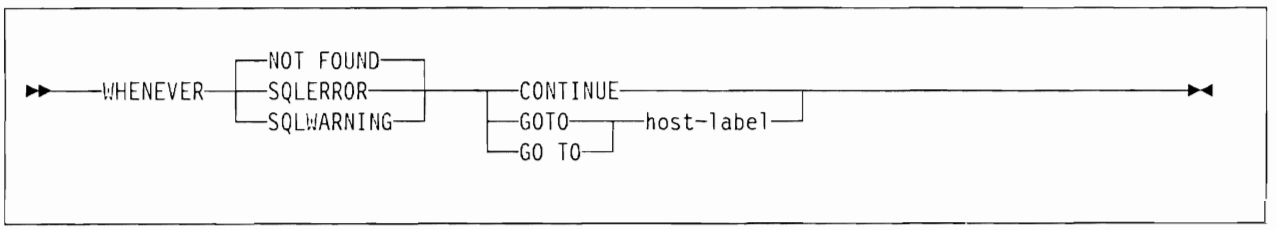
The WHENEVER statement specifies the action to be taken when a specified exception condition occurs.

Invocation

This statement can only be embedded in an application program. It is not an executable statement.

Authorization

None required.



Description

The NOT FOUND, SQLERROR, or SQLWARNING clause is used to identify the type of exception condition.

NOT FOUND
Identifies any condition that results in an SQL return code of +100.

SQLERROR
Identifies any condition that results in a negative SQL return code.

SQLWARNING
Identifies any condition that results in a warning condition (SQLWARN0 is 'W'), or that results in a positive SQL return code other than +100.

The CONTINUE or GO TO clause is used to specify what is to happen when the identified type of exception condition exists.

CONTINUE
Causes the next sequential instruction of the source program to be executed.

GOTO or **GO TO** *host-label*
Causes control to pass to the statement identified by *host-label*. For *host-label*, substitute a single token, optionally preceded by a colon. The form of the token depends on the host language. In COBOL, for example, it can be a *section-name* or an unqualified *paragraph-name*.

Notes

There are three types of WHENEVER statements:

WHENEVER NOT FOUND
WHENEVER SQLERROR
WHENEVER SQLWARNING

Every executable SQL statement in a program is within the scope of one implicit or explicit WHENEVER statement of each type. The scope of a

WHENEVER statement is related to the listing sequence of the statements in the program, not their execution sequence.

An SQL statement is within the scope of the last WHENEVER statement of each type that is specified before that SQL statement in the source program. If a WHENEVER statement of some type is not specified before an SQL statement, that SQL statement is within the scope of an implicit WHENEVER statement of that type in which CONTINUE is specified.

Example

If an error is produced, go to HANDLERR. If a warning code is produced, continue with the normal flow of the program. If no results are found, go to ENDDATA.

```
EXEC SQL WHENEVER SQLERROR GOTO HANDLERR END-EXEC.
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
EXEC SQL WHENEVER NOT FOUND GO TO ENDDATA END-EXEC.
```

Appendix A. SQL Limits

The table below describes certain limits imposed by SQL/400. The limits presented here are minimal values that, if used, will assist in making application programs readily portable from one system to another.

Table 5 (Page 1 of 2). SQL Limits

ITEM	SQL LIMIT
Longest SQL identifier	10
Longest authorization ID	10
Longest cursor name	18
Longest statement name	18
Most columns in a table	8000
Most columns in a view	8000
Maximum length of a row, in bytes, including all overhead	32766
Maximum byte count of a row	32766
Maximum byte count of CHAR	32766
Largest INTEGER value	2147483647
Smallest INTEGER value	-2147483648
Largest SMALLINT value	32767
Smallest SMALLINT value	-32768
Largest FLOAT value	Approximately 1.79×10^{308}
Smallest FLOAT value	Approximately -1.79×10^{308}
Smallest positive FLOAT value	Approximately 2.22×10^{-308}
Largest negative FLOAT value	Approximately -2.22×10^{-308}
Largest REAL value	Approximately 3.40×10^{38}
Smallest REAL value	Approximately -3.40×10^{38}
Smallest positive REAL value	Approximately 1.17×10^{-38}
Largest negative REAL value	Approximately -1.17×10^{-38}
Largest DECIMAL or NUMERIC value	999999999999999999999999999999.
Smallest DECIMAL or NUMERIC value	-999999999999999999999999999999.
Most table names in an SQL statement. (In a complex SELECT, the number of tables that can be joined may be significantly less.)	32
Maximum length of a host variable name	64

Table 5 (Page 2 of 2). SQL Limits	
ITEM	**SQL LIMIT**
Most host variables in a precompiled program	4000 or less[8]
Most host variables in an SQL statement	4000 or less[8]
Maximum total length of host and indicator variables pointed to in an SQLDA	storage
Longest host variable used for insert or update	32766 bytes
Longest SQL statement	32767 bytes
Most elements in a select list	8000 or less[9]
Most functions in a select list	8000 or less[9]
Most predicates in a WHERE or HAVING clause	8000 or less[9]
Maximum total length of columns in a GROUP BY clause	120
Maximum number of columns in a GROUP BY clause	120
Maximum total length of columns in an ORDER BY clause	256
Maximum number of columns in an ORDER BY clause	256
Most columns in an index key	120
Longest index key	120

[8] The limit is based on the number of pointers allowed in a program. Each host language allows a different number of pointers. At a minimum, each host language uses one pointer for each use of a host variable. AS/400 system allows a limit of 4040 pointers in any program.

[9] The limit is based on the size of internal structures generated for the parsed SQL statement.

Appendix B. SQLCA and SQLDA Control Blocks

This appendix describes the SQL communication area (SQLCA) and the SQL descriptor area (SQLDA).

SQL Communication Area (SQLCA)

An SQLCA is a collection of variables that is updated repeatedly during a program with information about the SQL statement most recently run. The SQL INCLUDE statement is used to provide the declaration of the SQLCA in COBOL, C, and PL/I. The SQLCA is provided for RPG by the SQL precompiler.

In COBOL, the name of the storage area must be SQLCA. In PL/I and C, the name of the structure must be SQLCA. Every SQL statement must be within the scope of its declaration.

When a stand-alone SQLCODE is specified in the program, the SQLCA must not be included. The precompiler will include an SQLCA with the name of the variable SQLCODE changed to SQLCADE (or SQLCOD changed to SQLCAD). The precompiler will add statements to the program to ensure that the stand-alone SQLCODE contains the correct values.

The stand-alone SQLCODE must not be specified in an RPG program.

Description of Fields

The names in the following table are those provided by the SQL INCLUDE statement. For the most part, COBOL, PL/I, and C use the same names. RPG names are different, because they are limited to 6 characters. Note one instance where PL/I names differ from COBOL.

Table 6 (Page 1 of 2). Names Provided by the SQL INCLUDE Statement

COBOL, PL/I or, C Name	RPG Name	Data Type	Description
SQLCAID sqlcaid	SQLAID	CHAR(8)	An "eye catcher" for storage dumps, containing 'SQLCA '.
SQLCABC sqlcabc	SQLABC	INTEGER (4-bytes)	Contains the length of the SQLCA, 136.
SQLCODE sqlcode	SQLCOD	INTEGER (4-bytes)	Contains the SQL return code. **Code** **Means** 0 Successful execution (though there may have been warning messages). positive Successful execution, but with an exception condition. negative Error condition.
SQLERRML [10] sqlerrml	SQLERL	SMALLINT (2-bytes)	Length indicator for SQLERRMC, in the range 0 through 70. 0 means that the value of SQLERRMC is not pertinent.
SQLERRMC [10] sqlerrmc	SQLERM	CHAR (70)	Contains message replacement text associated with the SQLCODE.
SQLERRP sqlerrp	SQLERP	CHAR(8)	Provides diagnostic information, such as the name of a module.
SQLERRD sqlerrd	SQLERR Defined as 24 characters (not an array) that are redefined by the fields SQLER1 through SQLER6. The fields are full-word binary.	Array	6 INTEGER variables that provide diagnostic information. SQLERRD(1) may contain the last four characters of the CPF escape message if SQLCODE is less than 0. For example, if the message is CPF5715, X'F5F7F1F5' is placed in SQLERRD(1).[11] SQLERRD(2) may contain the last four characters of a CPD diagnostic message if the SQL code is less than 0.[11] SQLERRD(3) shows the number of rows affected after INSERT, UPDATE, and DELETE.
SQLWARN sqlwarn	SQLWRN Defined as 8 characters (not an array).	CHAR(8)	A set of 8 warning indicators, each containing blank or 'W'.
SQLWARN0 sqlwarn0	SQLWN0	CHAR(1)	Blank if all other indicators are blank; contains 'W' if at least one other indicator contains 'W'.
SQLWARN1 sqlwarn1	SQLWN1	CHAR(1)	Contains 'W' if the value of a string column was truncated when assigned to a host variable. Contains 'N' if the value of a string column was assigned to a C null-terminated host variable and if the host variable was large enough to contain the result but not large enough to contain the null-terminator.
SQLWARN2 sqlwarn2	SQLWN2	CHAR(1)	Reserved
SQLWARN3 sqlwarn3	SQLWN3	CHAR(1)	Contains 'W' if the number of columns and the number of host variables are not the same.
SQLWARN4 sqlwarn4	SQLWN4	CHAR(1)	Contains 'W' if a prepared UPDATE or DELETE statement does not include a WHERE clause.
SQLWARN5 sqlwarn5	SQLWN5	CHAR(1)	Reserved

[10] In COBOL, SQLERRM includes SQLERRML and SQLERRMC. In PL/I, the varying-length string SQLERRM is equivalent to SQLERRML prefixed to SQLERRMC.

[11] SQLERRD(1) and (2) are set only if appropriate.

Table 6 (Page 2 of 2). Names Provided by the SQL INCLUDE Statement

COBOL, PL/I or, C Name	RPG Name	Data Type	Description
SQLWARN6 sqlwarn6	SQLWN6	CHAR(1)	Reserved
SQLWARN7 sqlwarn7	SQLWN7	CHAR(1)	Reserved
SQLWARN8 sqlwarn8	SQLWN8	CHAR(1)	Reserved
SQLWARN9 sqlwarn9	SQLWN9	CHAR(1)	Reserved
SQLWARNA sqlwarna	SQLWNA	CHAR(1)	Reserved
SQLSTATE sqlstate	SQLSTT	CHAR(5)	Reserved

The Included SQLCA

The following is a description of the SQLCA that is given by INCLUDE SQLCA

In **COBOL**:

```
01 SQLCA.
   05 SQLCAID      PIC X(8).
   05 SQLCABC      PIC S9(9) COMP-4.
   05 SQLCODE      PIC S9(9) COMP-4.
   05 SQLERRM.
      49 SQLERRML  PIC S9(4) COMP-4.
      49 SQLERRMC  PIC X(70).
   05 SQLERRP      PIC X(8).
   05 SQLERRD      OCCURS 6 TIMES
                   PIC S9(9) COMP-4.
   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).
```

INCLUDE SQLCA must not be specified in other than the working storage section.

In **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);
```

In **C**, INCLUDE SQLCA declarations are equivalent to the following:

```
#ifndef SQLCODE
struct sqlca
{
       unsigned char  sqlcaid[8];
       long           sqlcabc;
       long           sqlcode;
       short          sqlerrml;
       unsigned char  sqlerrmc[7];
       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;
```

In **RPG**: The SQLCA data structure is generated by the SQL precompiler and is not specified by the user.

```
ISQLCA          DS
I                                       1   8 SQLAID            SQL
I                                     B   9  120SQLABC            SQL
I                                     B  13  160SQLCOD            SQL
I                                     B  17  180SQLERL            SQL
I                                        19  88 SQLERM            SQL
I                                        89  96 SQLERP            SQL
I                                        97 120 SQLERR            SQL
I                                     B  97 1000SQLER1            SQL
I                                     B 101 1040SQLER2            SQL
I                                     B 105 1080SQLER3            SQL
I                                     B 109 1120SQLER4            SQL
I                                     B 113 1160SQLER5            SQL
I                                     B 117 1200SQLER6            SQL
I                                       121 131 SQLWRN            SQL
I                                       121 121 SQLWN0            SQL
I                                       122 122 SQLWN1            SQL
I                                       123 123 SQLWN2            SQL
I                                       124 124 SQLWN3            SQL
I                                       125 125 SQLWN4            SQL
I                                       126 126 SQLWN5            SQL
I                                       127 127 SQLWN6            SQL
I                                       128 128 SQLWN7            SQL
I                                       129 129 SQLWN8            SQL
I                                       130 130 SQLWN9            SQL
I                                       131 131 SQLWNA            SQL
I                                       132 136 SQLSTT            SQL
```

The SQL Descriptor Area (SQLDA)

An SQLDA is a collection of variables that is required for execution of the SQL DESCRIBE statement, and may optionally be used by the PREPARE, OPEN, FETCH, and EXECUTE statements. An SQLDA communicates with dynamic SQL; it can be used in a DESCRIBE statement, modified with the addresses of host variables, and then reused in a FETCH statement.

The meaning of the information in an SQLDA depends on its use. In PREPARE and DESCRIBE, an SQLDA provides information to an application program about a prepared statement. In OPEN, EXECUTE, and FETCH, an SQLDA provides information about host variables.

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, and EXECUTE, each occurrence of SQLVAR describes a host variable. In DESCRIBE and PREPARE, they describe columns of a result table.

PL/I Name	C Name	Data Type	Usage in DESCRIBE and PREPARE (Set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (Set by the user prior to executing the statement)
SQLDAID	sqldaid	CHAR(8)	An "eye catcher" for storage dumps, containing 'SQLDA '.	Not used.
SQLDABC	sqldabc	INTEGER	Length of the SQLDA, equal to SQLN*LENGTH(SQLVAR)+16.	Number of bytes of storage allocated for the SQLDA. Enough storage must be allocated to allow for all occurrences of SQLN. SQLDABC must be set to a value greater than or equal to 16 + SQLN*(N), where N is the length of an SQLVAR occurrence, which is implementation-defined.
SQLN	sqln	SMALLINT	Unchanged by the database manager. Must be set to a value greater than or equal to zero prior to use by DESCRIBE or PREPARE. 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 0.
SQLD	sqld	SMALLINT	The number of columns described by occurrences of SQLVAR.	SQLD must have a value less than or equal to SQLN prior to use by FETCH, OPEN, or EXECUTE.
SQLVAR	sqlvar	ARRAY	Set of six fields.	Not used.

Fields in an Occurrence of SQLVAR

PL/I Name	C Name	Data Type	Usage in DESCRIBE and PREPARE (Set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (Set by the user prior to executing the statement)
SQLTYPE	sqltype	SMALLINT	Tells the data type of the column and whether or not it has an associated indicator variable. "Values of SQLTYPE" on page 136 lists the allowable values and their meanings.	Same as usage in DESCRIBE and PREPARE, except that SQLTYPE must be set prior to use by FETCH, OPEN, or EXECUTE.
SQLLEN	sqllen	SMALLINT	Gives the length attribute of the column, as follows. **Data Type** — **Content** Character — Length attribute in bytes. Decimal, Numeric, or Integer — Byte 1 = precision; byte 2 = scale. Float — 4 for single precision; 8 for double precision. Integer — 2 for SMALLINT; 4 for INTEGER. **Note**: Binary numbers may be represented in the SQLDA as either length of 2 or 4 or with precision and scale. If the first byte is greater than X'00', it indicates precision and scale.	Same as usage in DESCRIBE and PREPARE, except that SQLTYPE must be set prior to use by FETCH, OPEN, or EXECUTE.
SQLRES	sqlres	reserved area (12-byte)	Provides boundary alignment. Pointers must be on a quad-word boundary.	Same as usage in DESCRIBE and PREPARE.
SQLDATA	sqldata	pointer (16-byte)	The third and fourth bytes contain a small integer that indicates whether the column is FOR BIT DATA. If the small integer is -1, the column is bit data (FOR BIT DATA). The remaining 12 bytes are reserved.	SQLDATA must have the address of the host variables prior to use by FETCH, OPEN, or EXECUTE.

PL/I Name	C Name	Data Type	Usage in DESCRIBE and PREPARE (Set by the database manager except for SQLN)	Usage in FETCH, OPEN, and EXECUTE (Set by the user prior to executing the statement)
SQLIND	sqlind	pointer (16-byte)	Reserved.	SQLIND must have the address of an indicator variable prior to use by FETCH, OPEN, or EXECUTE. A negative value indicates null and a non-negative values indicates not null. The following cases might result in a null value, even though null values cannot be stored in tables: • If a column function (MIN, MAX, COUNT, etc.) are specified in the SELECT list, and the GROUP BY clause was not specified, and the result of COUNT is 0, then a null value (-1) is returned in the indicator variable for the column functions other than COUNT. • If a decimal data error occurred when evaluating an expression in the SELECT list, but a successful determination could still be made as to whether the resulting row should be selected, as many valid results will be returned as possible, and items that encountered errors will be returned as a null value (-2).
SQLNAME	sqlname	Varying-length character string: maximum length is 30 characters.	Contains the name or label of the column.	Not used.

Values of SQLTYPE

The table below lists allowable values of the SQLTYPE field of an SQLDA, and their meanings. There are two values for each data type:

- For DESCRIBE and PREPARE statements, the first value does not allow null values, but the second value does allow null values.
- For FETCH, OPEN, and EXECUTE statements, the first value does not have an indicator variable, but the second value does have an indicator variable.

Table 7. Allowable SQLTYPE Field Values

Values	Data Type	DESCRIBE and PREPARE — Null Indicator?	FETCH, OPEN, and EXECUTE — Indicator Variable?
448/449	varying-length character string	no/yes	no/yes
452/453	fixed-length character string	no/yes	no/yes
456/457	long varying-length character string (greater than 254 bytes)	no/yes	no/yes
460/461	C only. Varying length, null terminated character string	no/yes	no/yes
480/481	floating point	no/yes	no/yes
484/485	decimal	no/yes	no/yes
488/489	numeric (zoned)	no/yes	no/yes
496/497	large integer	no/yes	no/yes
500/501	small integer	no/yes	no/yes

The Included SQLDA

In **PL/I**, INCLUDE SQLDA specifies:

```
DCL 1 SQLDA BASED(SQLDAPTR),
      2 SQLDAID     CHAR(8),
      2 SQLDABC     BIN FIXED(31),
      2 SQLN        BIN FIXED,
      2 SQLD        BIN FIXED,
      2 SQLVAR      (99),
        3 SQLTYPE   BIN FIXED,
        3 SQLLEN    BIN FIXED,
        3 SQLRES    CHAR(12),
        3 SQLDATA   PTR,
        3 SQLIND    PTR,
        3 SQLNAME   CHAR(30) VAR;
DCL SQLDAPTR PTR;
```

In **C**, INCLUDE SQLDA specifies:

```
#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;
                     unsigned char  data[30];
              } sqlname;
       } sqlvar[1];
};
#define  SQLDASIZE(n) (sizeof(struct sqlda)+(n-1) * sizeof(struct sqlvar))
#endif
```

Appendix C. Reserved Words

ALL
AND
ANY
AS

BETWEEN
BY

COLLECTION
COLUMN
COUNT
CURRENT
CURSOR

DATABASE
DELETE
DESCRIPTOR
DISTINCT

END-EXEC[12]
EXECUTE

FOR
FROM

GO
GOTO
GROUP

HAVING

IMMEDIATE
IN
INDEX
INSERT
INTO
IS

LIKE

NOT
NULL

OF
ON
OR
ORDER

PRIVILEGES

SELECT
SET

TABLE
TO

UNION
UPDATE
USER
USING

VALUES
VIEW

WHERE
WITH

[12] COBOL only

Glossary

access path. The path used to locate data specified in SQL statements. An access path can be either indexed or sequential, or a combination of both.

access plan. The control structure produced during compile time that is used to process SQL statements encountered when the program is run.

ANSI. American National Standards Institute

application. A program or set of programs that perform a task; for example, a payroll application.

attribute. In database design, a characteristic of an entity; for example, the telephone number of an employee is one of that employee's attributes.

authorization ID. A user profile. A name identifying a user to whom privileges can be granted.

automatic bind. When an application program is being run and the access plan is not valid, binding takes place automatically; that is, without a user issuing a CRTSQL*xxx* command, where *xxx* is RPG, PLI, CBL, or C.

binary. An SQL data type indicating that the data is a binary number with a precision of 15 (halfword) or 31 (fullword) bits.

bind. The process by which the output from the SQL precompiler is converted to a usable structure called an access plan. This process is the one during which access paths to the data are selected and some authorization checking is performed. There are two types of bind used by SQL/400: automatic and dynamic (see *automatic bind* and *dynamic bind*).

catalog. Tables, maintained by the database manager, that contain descriptions of objects, such as tables, views, and indexes.

catalog views. A set of views containing information about the objects in a collection, such as tables, views, indexes, and column definitions.

character string. A sequence of bytes or characters associated with a single-byte character set.

clause. A distinct part of a statement in the language structure, such as a SELECT clause or a WHERE clause.

collection. A set of objects created by SQL/400 that contains tables, views, indexes, and other system objects (such as a program) created by the user. An SQL collection consists of a library; a data dictionary that contains description and information for all tables, views, indexes, and files created into the library; an SQL catalog; and a journal and journal receiver that are used to journal changes to all tables created into the collection.

column. The vertical part of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. A process that calculates a value from a set of values and expresses it as a function name followed by an argument enclosed in parentheses.

commit. The process that data changed by one application or user to be used by other applications or users. When a commit operation occurs, the locks are released to allow other applications to use the changed data.

commit point. The point in time when data is considered to be consistent.

comparison operator. A symbol (such as =, >, <) used to specify a relationship between two values.

concurrency. The shared use of resources by multiple interactive users or application programs at the same time.

correlation name. An identifier that designates a table, a view, or an individual row of a table or view within a single SQL statement. The name can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cursor. A named control structure used by an application program to point to a row of data. The position of the row is within a table or view, and the cursor is used to interactively select rows from the columns.

data type. An attribute of columns, constants, and host variables.

DBCS. See *double-byte character set (DBCS)*.

default value. A predetermined value, attribute, or option that is assumed when no other value is explicitly specified. For example, the value of a column is a nonnull value determined by the data type of the column.

delimited identifier. A sequence of one or more characters of the standard character set enclosed within SQL escape characters used to form a name.

delimiter token. A string constant, a delimited identifier, a symbol (for example, ||, /, *, +, or −), or other special characters (for example, period, comma, parentheses).

double-byte character set (DBCS). A set of characters used by national languages, such as Japanese and Chinese, that have more symbols than can be represented by the 256 single-byte EBCDIC positions. Each character is 2 bytes in length, and therefore requires special hardware to be displayed or printed. Contrast with single-byte character set.

dynamic bind. When SQL statements are entered interactively, binding is done dynamically (that is, as the SQL statements are entered).

dynamic SQL. SQL statements that are prepared and processed within a program while the program is running. The SQL source statements are contained in host-language variables rather than being coded directly into the application program. The SQL statement might change several times while the program is running.

EBCDIC. See *extended binary coded decimal interchange code (EBCDIC)*.

embedded SQL. SQL statements that are embedded within a program and are prepared during the program preparation process before the program is run. After it is prepared, the statement itself does not change, although values of host variables specified within the statement might change.

escape character. The symbol used to enclose a delimited identifier. This symbol is the quotation mark ("), except in COBOL programs where the symbol can be assigned by the user as either a quotation mark or an apostrophe.

expression. An operand, or a collection of operators and operands, that yields a single value.

extended binary coded decimal interchange code (EBCDIC). A coded character set of 256 8-bit characters.

fixed-length string. A character string whose length is specified and cannot be changed. Contrast with varying-length string.

fullword binary. A binary number with a precision of 31 bits. See also *integer*.

full select. That form of the select-statement that includes ORDER BY or UNION operators.

function. A column function or a scalar function.

halfword binary. A binary number with a precision of 15 bits.

host language. Any programming language, such as COBOL, PL/I, C, and RPG, in which you can embed SQL statements.

host structure. In an application program, a structure referred to by embedded SQL statements. In RPG, this is called a *data structure*; in PL/I and C, this is known as a *structure*; in COBOL, this is called a *group item*.

host variable. In an application program, a variable referred to by embedded SQL statements. In RPG, this is called a *field name*; in PL/I and C, this is known as a *variable*; in COBOL, this is called a *data item*.

identifier. See *delimited identifier* and *ordinary identifier*.

index. A set of pointers that are logically arranged by the values of a key. Indexes provide quick access to data and can enforce uniqueness on the rows in a table.

index key. The set of columns in a table used to determine the order of indexed entries.

indicator variable. A variable used to represent the null value in an application program. For example, if the value for the results column is null, SQL puts a negative value in the indicator variable.

integer. An SQL data type indicating that the data is a binary number with a precision of 31 bits.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values.

key. A column or an ordered set of columns identified in the description of an index.

keyword. A name that identifies a parameter used in an SQL statement or SQL precompiler command. See also *parameter*.

lock. The process by which integrity of data is ensured. The prevention of concurrent users from accessing inconsistent data.

long string. A string whose actual length, or a varying-length string whose maximum length, is greater than 254 bytes or 127 double-byte characters.

mixed data string. A character string that can contain both single-byte and double-byte characters.

null. A special value that indicates the absence of information.

object. Anything that can be created or manipulated with SQL statements, such as collections, tables, views, or indexes.

ordinary identifier. A letter followed by zero or more characters, each of which is a letter ($, @, #, a-z, and A-Z), a number, or the underscore character used to form a name. An ordinary identifier must not be identical to a reserved word.

ordinary token. A numeric constant, and ordinary identifier, a host variable, or a keyword.

page. A unit of storage equal to 512 bytes.

parameter. The *keywords* and *values* that further define SQL precompiler commands and SQL statements.

plan. See *access plan*.

precompile. A processing of programs containing SQL statements that takes place before a compile. SQL statements are replaced with statements that will be recognized by the host language compiler. The output from this precompile includes source code that can be submitted to the compiler and used in the bind process.

predicate. An element of a search value that expresses or implies a comparison operation.

prepared SQL statement. A named object that is the form of an SQL statement that was processed by the PREPARE statement.

privilege. A capability given to a user by the processing of a GRANT statement.

rebind. The creation a new access plan for a program that was previously bound. If, for example, you add an index for a table that is used by your application program, SQL/400 may automatically bind the application again to take advantage of that index.

real table. A physical file or a table created by SQL.

recovery. The process of rebuilding databases after a system failure.

relational database. A data structure perceived by its users as a collection of tables.

result column. An expression in a SELECT clause that SQL selects for an application program.

result table. The set of rows that SQL selects for an application program. The program uses a cursor to retrieve the rows one by one into a host structure or a set of host variables.

rollback. The process of restoring data changed by an application to the state at its last commit point.

row. The horizontal part of a table. A row consists of a sequence of values, one for each column of the table.

SBCS. See *single-byte character set (SBCS)*.

scalar function. An operation that produces a single value from another value and expresses it in the form of a function name followed by a list of arguments enclosed in parentheses.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

short string. A string whose actual length, or a varying-length string whose maximum length, is less than or equal to 254 bytes.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code.

small integer. An SQL data type indicating that the data is a binary number with a precision of 15 bits.

special register. A storage area whose primary use is to store information produced in conjunction with the use of specific SQL functions. The SQL/400 special register is (named) USER.

SQL. See *Structured Query Language*.

SQLCA. See *SQL communication area (SQLCA)*.

SQLDA. See *SQL descriptor area (SQLDA)*.

SQL communication area (SQLCA). A collection of variables that are used by SQL to provide an application program with information about the processing of SQL statements within the program.

SQL descriptor area (SQLDA). A collection of variables that are used in the processing of certain SQL statements. The SQLDA is intended for dynamic SQL programs.

static SQL. SQL statements that are embedded within a program, and are prepared during the program preparation process before the program is run. After being prepared, the statement itself does not change (although values of host variables specified by the statement might change).

string. A character string.

string delimiter. A symbol used to enclose an SQL string constant. This symbol is the apostrophe ('), except in COBOL applications, in which case the

symbol (either an apostrophe or a quotation mark) may be assigned by the user.

Structured Query Language (SQL). A language that can be used within host programming languages or interactively to access data and to control access to resources.

subselect. That form of a query that does not include ORDER BY or UNION operators.

table. A named data object consisting of a specific number of columns and some number of unordered rows.

token. See *delimited token* and *ordinary token*.

union. An SQL operation that combines the results of two subselects. Union is often used to merge lists of values obtained from several tables.

unique index. An index that assures that no identical key values are stored in a table.

unit of recovery. A sequence of operations within a unit of work between two commit points.

unlock. To release an object or system resource that was previously locked and return it to general availability.

user profile. An object with a unique name that contains the user's password, the list of special authorities assigned to a user, and the objects the user owns. See also *authorization ID*.

value. Smallest unit of data manipulated in SQL.

varying-length string. A character string whose length is not fixed, but variable within limits. Contrast with *fixed-length string*.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns contained in the table or tables on which it is defined.

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Special Characters

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your IBM-approved remarketer. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your IBM representative or your IBM-approved remarketer to request additional publications.

Name _____

Company or Organization _____

Address _____

_____

City State Zip Code

Phone No. _____

Area Code

No postage necessary if mailed in the U.S.A.

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

IBM®

Cut or Fold
Along Line

IBM®

Program Number
5728-ST1

21F2757

SC21-9608-1

Printed in U.S.A.