Ingres® 9.3

OpenSQL Reference Guide

INGRES

ING-93-OSR-02

Contents

Chapter 5: Embedded OpenSQL 101

Chapter 6: Dynamic OpenSQL 135

Chapter 1: Introduction

This section contains the following topics:

In This Guide

The *OpenSQL Reference Guide* describes OpenSQL usage and syntax.

OpenSQL was specifically designed to be compatible across several SQL dialects. OpenSQL is functionally equivalent to Ingres SQL without the extensions specific to Ingres. This guide is designed for programmers who write applications that are portable across all EDBC, Enterprise Access, and Ingres servers.

Note: If you are working through an Enterprise Access product, see your Enterprise Access product documentation for information about syntax that may differ from that described in this guide.

Audience

The *OpenSQL Reference Guide* is intended for programmers and OpenSQL users who have a basic understanding of how Ingres and relational database systems work. In addition, the reader should have a basic understanding of the operating system. This guide is also intended as a reference for the database system administrator.

Syntax Conventions Used in This Guide

This guide uses the following conventions to describe command and statement syntax:

Convention	Usage
`Monospace`	Indicates keywords, symbols, or punctuation that you must enter as shown.
Italics	Represent a variable name for which you must supply a value. This convention is used in explanatory test, as well as syntax.
[ ] brackets	Indicate an optional item.
{ } braces	Indicate an optional item that you can repeat as many times as appropriate.
| (vertical bar)	Indicates a list of mutually exclusive items (that is, you can select only one item from the list).

Conventions for Embedded OpenSQL Examples

Examples of embedded OpenSQL code provided in this guide use the following conventions:

Convention	Usage
Margins	None are used.
; (semicolon)	Represents the statement terminator.
Labels	Appear on their own line and are followed by a colon (:). Control passes to the statement following the label.
Host language comments	Indicated by the OpenSQL comment indicator; for example: /* This is a comment. */
' ' (single quotes)	Surround character strings.
pseudocode	Represents host language statements in embedded OpenSQL. For example: exec sql begin declaration; *variable declarations* exec sql end declaration;

To determine the correct syntax for your programming language, see the *Embedded SQL Companion Guide*.

Terminology Used in This Guide

This guide uses the following terminology:

command

A *command* is an operation that you execute from an OpenROAD Development menu or at the operating system level.

statement

A *statement* is an operation that you place in a program or called procedure. Statements can be written in OpenROAD's fourth-generation language (4GL), a database query language (such as SQL), or a 3GL (like C or COBOL).

Chapter 2: Overview of OpenSQL

This section contains the following topics:

What Is OpenSQL?

SQL (Structured Query Language) is a language that allows you to manipulate and maintain data in a relational database. OpenSQL is a version of SQL that was specifically designed to be compatible across several SQL dialects. OpenSQL allows you to create applications that run on the following servers:

- The Ingres DBMS (for Ingres databases)
- Enterprise Access products (for access to other database management systems)
- Ingres Star (for distributed databases)
- EDBC servers

OpenSQL statements can be used in the following contexts:

- Terminal Monitor
- Embedded OpenSQL programs
- Applications built with Vision (a forms-based application development tool)
- Applications built with OpenROAD (a graphical user interface application development tool)

Use OpenSQL statements in the interactive Terminal Monitor or in embedded OpenSQL programs. OpenSQL statement syntax and results are consistent across supported host programming languages. This guide does not include specific information about host languages. For details, see the *Embedded SQL Companion Guide*.

Forms statements allow you to write embedded applications that interact with users through Visual Forms Editor forms. For details about forms statements, see the *Forms-based Application Development Tools User Guide*.

Enterprise Access Products

Enterprise Access products (sometimes referred to as Gateways) are interfaces between Ingres applications and database management systems other than Ingres. The Enterprise Access products provide a variety of services, including:

- Translating between OpenSQL and host query interfaces, such as Rdb/VMS (for Alpha VMS), or DB2 UDB (for IBM DB2 Universal Database)
- Emulating SQL functions for non-relational databases such as IMS and RMS
- Converting between OpenSQL data types and data types that are native to other host database management systems
- Translating host DBMS error messages to Ingres generic errors

Enterprise Access products are *transparent*, meaning that host databases are presented as if they are Ingres databases.

Ingres Star

Ingres Star provides a single, consistent system view of multiple databases managed by either the Ingres DBMS Server or by Enterprise Access products.

Interactive OpenSQL

Interactive OpenSQL provides full access to Ingres databases, distributed databases (through Ingres Star), and other types of databases (through Enterprise Access products). Because interactive OpenSQL statements are similar to their embedded versions, use interactive OpenSQL to test the queries to be used in embedded programs. The user interface to interactive OpenSQL is the Terminal Monitor.

Embedded OpenSQL

Using embedded OpenSQL, OpenSQL statements can be mixed with host language statements. Use host language variables to specify values required by embedded OpenSQL statements. For information about the requirements of a specific host language, see the *Embedded SQL Companion Guide*.

How Embedded OpenSQL Programs Are Built

The Embedded SQL preprocessor converts embedded OpenSQL statements in your program into host language source code statements. The resulting statements are calls to a runtime library that provides the interface to Ingres, Ingres Star, and Enterprise Access products. Non-SQL host language statements are passed through the preprocessor without being altered. After the program has been preprocessed, it must be compiled and linked as appropriate for the host language. For details on preprocessing an embedded OpenSQL program, see the *Embedded SQL Companion Guide*.

SQLCA—Status Information Retrieval

Status information is available to an embedded program from the SQL Communications Area (SQLCA). The SQLCA is a data structure that can be included in the program. The SQLCA contains information concerning the results of the last executed OpenSQL statement. Statements in embedded OpenSQL programs can refer to data in the SQLCA for execution of conditional actions. For information on the language-specific data structure of the SQLCA, see the *Embedded SQL* Companion *Guide*.

Dynamic OpenSQL—Specifying Parameters at Runtime

OpenSQL enables you to execute queries that are formulated at runtime (rather than before preprocessing). This is known as *dynamic OpenSQL*.

Differences Between Embedded and Interactive OpenSQL

Embedded OpenSQL builds on the features and statements available in interactive OpenSQL. However, embedded OpenSQL differs from interactive OpenSQL in the following areas:

- **Host language variables** - Embedded OpenSQL allows host variables to be used in place of many syntactic elements. (There are no variables in interactive OpenSQL.)
- **Error and status information** - In interactive OpenSQL, error and status messages are sent directly to the terminal screen. In embedded OpenSQL, the SQL Communications Area (SQLCA) structure receives error and status information.
- **Data manipulation statements** - There are two embedded versions of the select statement. The first version is similar to the interactive select statement. The second version allows the retrieval and updating of an indeterminate number of rows, using cursors. The update and delete statements also have cursor versions. For more information about cursors, see Data Manipulation with Cursors in the chapter "Embedded OpenSQL."
- **Dynamic OpenSQL statements** - Embedded OpenSQL creates statements dynamically from individual components specified in program variables. These statements can be executed repeatedly with different values.
- **Additional database access statements** - Embedded OpenSQL includes several statements not required in interactive OpenSQL. These additional statements enable your embedded application to connect to a database and to control cursors.
- **Repeated queries** - A repeated query executes more quickly than other queries, because the server retains the query execution plan. Embedded OpenSQL allows you to specify a select, insert, update, or delete statement as repeated.

OpenSQL Features

The availability of some OpenSQL features depend on the version of OpenSQL supported by the host DBMS to which your application connects. To determine which version of OpenSQL the host DBMS supports, select the row containing the OPEN/SQL_LEVEL capability from the iidbcapabilities system catalog.

The following OpenSQL features are only available when the OPEN/SQL_LEVEL value in the iidbcapabilities system catalog is 00605 or higher:

- Create schema statement
- Grant and revoke statements
- Create table statement: column constraints and defaults
- Schema.table syntax
- Delimited identifiers
- The escape clause in the like predicate
- Database procedures

For Enterprise Access, newer OpenSQL features may be available even though the Open/SQL_LEVEL value is not at 00605 or higher. For the current availability of OpenSQL features, see the Enterprise Access documentation.

Rules for Naming Objects

The rules for naming OpenSQL objects (such as tables, columns, and views) are as follows:

- All keywords are reserved and cannot be used as variable or object names in OpenSQL. In addition, embedded OpenSQL reserves all words beginning with "ii". Enforcement of keywords may vary by Enterprise Access product.
- Names can contain only alphanumeric characters and must begin with an alphabetic character or an underscore (_). In OpenSQL, alphabetic characters are A through Z, and numeric characters are 0 through 9, regardless of the II_CHARSETxx setting.
- All names are converted as necessary to the proper case for the host DBMS. The host DBMS stores names in the system catalogs in one of three formats: uppercase, lowercase, or mixed case. For more information, see The iidbcapabilities Catalog in the chapter "OpenSQL Standard Catalogs."
- The maximum length of an OpenSQL object name is 32 bytes. In an installation that uses the UTF8 or any other multi-byte character set, the maximum length of a name may be less than 32 bytes because some glyphs use multiple bytes.

  To ensure application portability, limit names to a maximum of 18 characters. For names of objects managed by the Ingres tools (such as Query-By-Forms, Report-By-Forms, Vision, and Visual Forms Editor), the maximum is 32 bytes. Examples of objects managed by the user interfaces are:

  - Forms
  - JoinDefs
  - QBFnames
  - Graphs
  - Reports

For more information about objects managed by the user interfaces, see the Character-*based Querying and Reporting Tools User Guide* or the guide that documents the specific user interfaces.

Regular and Delimited Identifiers

Identifiers in OpenSQL statements specify names for the following objects:

- User
- Column
- Correlation name
- Cursor
- Database procedure
- Database procedure parameter
- Index
- Prepared query
- Schema
- Table
- View

Specify these names using *regular* (unquoted) identifiers or *delimited* (double-quoted) identifiers. For example:

- Table name in a select statement specified using a regular identifier:

  ```
  select * from employees
  ```

- Table name in a select statement specified using a delimited identifier:

  ```
  select * from "my table"
  ```

Delimited identifiers allow special characters to be embedded in object names. OpenSQL restricts the use of special characters in regular identifiers.

Restrictions on Identifiers

For ANSI/ISO Entry SQL-92 standards compliance, identifiers should be no longer than 18 characters. The following table lists restrictions for each type of identifier:

Restriction	Regular Identifiers	Delimited Identifiers
Quotes	Specified without quotes	Specified in double quotes
Keywords	Cannot be a keyword	Can be a keyword
Case	Depends on host DBMS	Is significant
Valid Special	"At" sign (@)	Ampersand (&)

Restriction	Regular Identifiers	Delimited Identifiers
Characters	(Ingres only) Crosshatch (#) (Ingres only) Dollar sign ($) (Ingres only) Underscore (_)	Asterisk (*) "At" sign (@) Colon (;) Comma (,) Crosshatch (#) Dollar sign ($) Double quotes (") Equal sign (=) Forward slash (/) Left and right caret (< >) Left and right parentheses Minus sign (-) Percent sign (%) Period (.) Plus sign (+) Question mark (?) Semicolon (;) Single quote (') Space Underscore (_) Vertical bar (|)

The following characters cannot be embedded in object names using either regular or delimited identifiers:

- Backslash (\)
- Caret (^)
- Braces ({ })
- DEL (ASCII 127 or X'7F')
- Exclamation point (!)
- Left quote (ASCII 96 or X'60')
- Tilde (~)

To specify double quotes in a delimited identifier, the quotes must be repeated. For example:

```
"""Identifier""Name"""
```

is interpreted by OpenSQL as:

```
"Identifier"Name"
```

Case Sensitivity of Identifiers

Case sensitivity for regular and delimited identifiers depends on the underlying DBMS. For compliance with ANSI/ISO Entry SQL-92 standards, delimited identifiers must be case sensitive.

OpenSQL treats database and user names without regard to case.

Comment Delimiters

To indicate comments in interactive OpenSQL, use the following delimiters:

- "/*" and "*/" (left and right delimiters, respectively). For example:

  ```
  /* This is a comment */
  ```

  When using "/*...*/" to delimit a comment, the comment can continue over more than one line. For example,

  ```
  /* Everything from here...
  ...to here is a comment */
  ```

- "--" (left side only). For example,

  ```
  --This is a comment.
  ```

  The "--" delimiter indicates that the rest of the line is a comment. A comment delimited by "--" cannot be continued to another line.

To indicate comments in embedded OpenSQL, use the following delimiters:

- "--", with the same usage rules as interactive OpenSQL.
- Host language comment delimiters. For information about comment delimiters, see the *Embedded SQL Companion Guide*.

Statement Terminators

Statement terminators separate one OpenSQL statement from another. In interactive OpenSQL, the statement terminator is the semicolon (;). Statements must be terminated with a semicolon when entering two or more OpenSQL statements before issuing the go command (\g), selecting the Go menu item, or issuing some other Terminal Monitor command.

In the following example, the first and second statements are terminated by semicolons. The third statement need not be terminated with a semicolon, because it is the final statement.

```
select * from addrlst;
select * from emp
where fname = 'john';
select * from emp
where mgrname = 'dempsey'\g
```

If only one statement is entered, the statement terminator is not required. For example, the following single statement does not require a semicolon:

```
select * from addrlst\g
```

In embedded OpenSQL applications, the use of a statement terminator is determined by the rules of the host language. For details, see the *Embedded SQL Companion* Guide.

Correlation Names

Correlation names are used in queries to clarify the table (or view) to which a column belongs. For example, the following query uses correlation names to join a table with itself:

```
select a.empname from emp a, emp b
  where a.mgrname = b.empname
  and a.salary > b.salary;
```

Correlation names can also be used to abbreviate long table names.

Specify correlation names in select statements. A single query can reference a maximum of 126 correlation and table names (including all base tables referenced by views specified in a query).

Note: The maximum number of tables referenced in a single query is dependent on the host DBMS. The 126 maximum listed here is for the Ingres DBMS; other DBMSs supported by Enterprise Access and EDBC may have a higher or lower limit.

If a correlation name is not specified, the table name implicitly becomes the correlation name. For example, in the following query:

```
select * from employee
  where salary > 100000;
```

OpenSQL assumes the correlation name, employee, for the salary column and interprets the preceding query as:

```
select * from employee
  where employee.salary > 100000;
```

If a correlation name is specified for a table, the correlation name (and not the actual table name) must be used within the query. For example, the following query generates a syntax error:

```
/*incorrect*/
select * from employee e
where employee.salary > 35000;
```

A correlation name must be unique. For example, the following statement is illegal because the same correlation name is specified for different tables:

```
/*incorrect*/
select e.ename from employee e, manager e
  where e.dept = e.dept;
```

A correlation name that is the same as a table that you own cannot be specified. If you own a table called mytable, the following query is illegal:

```
select * from othertable mytable...;
```

In nested queries, OpenSQL resolves unqualified column names by checking the tables specified in the nearest from clause, then the from clause at the next higher level, and so on, until all table references are resolved.

For example, in the following query, the dno column belongs to the deptsal table, and the dept column to the employee table:

```
select ename from employee
  where salary >
       (select avg(salary) from deptsal
     where dno = dept);
```

Because the columns are specified without correlation names, OpenSQL performs the following steps to determine to which table the columns belong:

Column	Action
dno	OpenSQL checks the table specified in the nearest from clause (the deptsal table). The dno column does belong to the deptsal table. OpenSQL interprets the column specification as deptsal.dno.
dept	OpenSQL checks the table specified in the nearest from clause (deptsal). The dept column does not belong to the deptsal table. OpenSQL checks the table specified in the from clause at the next higher level (the employee table). The dept column does belong to the employee table. OpenSQL interprets the column specification as employee.dept.

OpenSQL does not search across subqueries at the same level to resolve unqualified column names. For example, given the query:

```
select * from employee

where
  dept = (select dept from sales_departments
      where mgrno=manager)
   or
   dept = (select dept from mktg_departments
      where mgrno=manager_id);
```

OpenSQL checks the description of the sales_departments table for the mgrno and manager columns. If they are not found, OpenSQL checks the employee table next, but will not check the mktg_departments table. Similarly, OpenSQL first checks the mktg_departments table for the mgrno and manager_id columns. If they are not found, OpenSQL will check the employee table, but will never check the sales_departments table.

Chapter 3: OpenSQL Data Types

This section contains the following topics:

OpenSQL Data Types

The following table lists the OpenSQL data types:

Class	Category	Data Type (Synonyms)
Character	Fixed length	character (char)
	Variable length	varchar (character varying)
		long varchar (clob, character large object, char large object)
Unicode	Fixed length	nchar
	Variable length	nvarchar
		long nvarchar (clob, nclob, nchar large object, national character large object)
Numeric	Exact numeric	integer (integer4, int)
		smallint (integer2)
		bigint (integer8)
		tinyint (integer1)
		decimal (dec, numeric)
	Approximate numeric	float (float8, double precision)
		real (float4)
Abstract	(none)	date
		money

Class	Category	Data Type (Synonyms)
Binary		long byte (blob, binary large object)

Character Data Types

Character data types are strings of ASCII characters. Upper and lower case alphabetic characters are accepted literally. OpenSQL supports one fixed-length character data type, character, and two variable-length character data types, varchar and long varchar.

The maximum size of a character column varies according to the DBMS being accessed. Additional space requirements for character columns are as follows:

- Varchar columns require two additional bytes to store a length specifier.
- Nullable columns require one additional byte to store a null indicator.

Note: Unicode data types are an extended feature of OpenSQL, which means that not all OpenSQL servers support Unicode. If an OpenSQL server supports Unicode, the iidbcapabilities catalog has a row with a cap_capability of NATIONAL_CHARACTER_SET and a cap_value of Y. To see if a particular OpenSQL server supports Unicode, refer to the documentation for that server.

Character Data Type

Character strings are fixed-length strings that can contain any printing or non-printing character, and the null character ('\0'). For example, if you enter ABC into a character(5) column, five bytes will be stored, as follows:

```
'ABC  '
```

Leading and embedded blanks are significant when comparing character strings. For example, OpenSQL considers the following character strings to be different:

```
'A B C'
'ABC'
```

When selecting character strings using the underscore (_) wildcard character of the like predicate, any trailing blanks you want to match must be included. For example, to select the following character string:

```
'ABC '
```

the wildcard specification must also contain trailing blanks:

```
'___  '
```

Length is not significant when comparing character strings. The shorter string is (logically) padded to the length of the longer. For example, OpenSQL considers the following character strings equal:

```
'ABC'
'ABC '
```

Char is a synonym for character.

Varchar Data Type

Varchar strings are variable-length strings, returned to applications as a 2-byte length specifier followed by character data. The varchar data type can contain any character, including non-printing characters and the ASCII null character ('\0').

Blanks are significant in the varchar data type. For example, OpenSQL does not consider the following two varchar strings equal:

```
'the store is closed'
```

and

```
'thestoreisclosed'
```

If the strings being compared are unequal in length, the shorter string is padded with trailing blanks until it equals the length of the longer string.

For example, the following two varchar strings:

```
'abcde' and 'abcd'
```

are compared as

```
'abcde' and  'abcd '
```

Long Varchar Data Type

The long varchar data type has the same characteristics as the varchar data type, but can accommodate strings up to two GB in length. Do not declare a length for long varchar columns. In embedded SQL *data handlers* can be created, which are routines to read and write the data for long varchar (and long byte) columns. For details about data handlers, see Data Handlers for Large Objects in the "Embedded SQL" chapter, and the *Embedded SQL Companion Guide*.

Restrictions on Long Varchar Columns

The following restrictions apply to long varchar columns:

- They cannot be part of a table key.
- They do not declare a length.
- They cannot be part of a secondary index.
- They cannot be used in the order by or group by clause in a select statement.
- They cannot have query optimization statistics. For details about query optimization statistics, see the discussion of the optimizedb utility in the *Command Reference Guide*.
- The following string functions do not work with long varchar columns:
  - Locate
  - Pad
  - Shift
  - Squeeze
  - Trim
  - Notrim
  - Charextract
- These columns cannot be directly compared to other string data types. To compare a long varchar column to another string data type, apply a coercion function.
- A string literal of more than 2000 characters cannot be assigned to a long varchar column. Details about assigning long strings to these columns are found in the description of data handlers in the *Embedded SQL Companion Guide* or the *OpenAPI User Guide*.

Unicode Data Types

Unicode data types support the coercion of local character data to Unicode data, and of Unicode data to local character data. Coercion function parameters are valid Character data types (for example, char, c, varchar and long varchar) and valid Unicode data types (nchar, nvarchar, and long nvarchar.)

The maximum row length is dependent on the Ingres page size and server configuration, and can be configured to a maximum of 32,767 bytes. For further discussion of page and row size configuration, see the *Database Administrator Guide*. The maximum length of a character column is limited by the maximum row width configured but not exceeding 32,000. Long nvarchar columns are an exception, with a maximum length of 2 GB.

Nchar Data Type

Fixed-length nchar strings can contain any printing or non-printing character, and the null character ('\0'). In uncompressed tables, nchar strings are padded with blanks to the declared length. (If the column is nullable, nchar columns require an additional byte of storage.) For example, if ABC is entered into a nchar(5) column, five bytes are stored, as follows:

```
'ABC  '
```

Leading and embedded blanks are significant when comparing nchar strings. For example, the following nchar strings are considered different:

```
'A B C'
'ABC'
```

When selecting nchar strings using the underscore (_) wildcard character of the like predicate, include any trailing blanks to be matched. For example, to select the following nchar string:

```
'ABC '
```

the wildcard specification must also contain trailing blanks:

```
'____'
```

Length is not significant when comparing nchar strings; the shorter string is (logically) padded to the length of the longer. For example, the following nchar strings are considered equal:

```
'ABC'
'ABC '
```

Nvarchar Data Type

Nvarchar strings are variable-length strings, stored as a 2-byte (smallint) length specifier followed by data. In uncompressed tables, nvarchar columns occupy their declared length. For example, if ABC is entered into an nvarchar(5) column, the stored result is:

```
'03ABCxx'
```

where 03 is a 2-byte length specifier, ABC is three bytes of data, and xx represents two bytes containing unknown (and irrelevant) data.

If the column is nullable, nvarchar columns require an additional byte of storage.

In compressed tables, nvarchar columns are stripped of trailing data. For example, if “ABC” is entered into an nvarchar(5) column in a compressed table, the stored result is:

```
'03ABC'
```

The nvarchar data type can contain any character, including non-printing characters and the ASCII null character ('\0').

Blanks are significant in the nvarchar data type. For example, the following two nvarchar strings are not considered equal:

```
'the store is closed'
```

and

```
'thestoreisclosed'
```

If the strings being compared are unequal in length, the shorter string is padded with trailing blanks until it equals the length of the longer string.

For example, consider the following two strings:

```
'abcd\001'
```

where '\001' represents one ASCII character (ControlA) and

```
'abcd'
```

If they are compared as nvarchar data types, then

```
'abcd' > 'abcd\001'
```

because the blank character added to 'abcd' to make the strings the same length has a higher value than ControlA ('\040' is greater than '\001').

Long Nvarchar Data Type

The long nvarchar data type has the same characteristics as the nvarchar data type, but can accommodate strings up to 2 GB in length. Do not declare a length for long nvarchar columns. In embedded SQL *data handlers* can be created, which are routines to read and write the data for long nvarchar (and long byte) columns. For details about data handlers, see Data Handlers for Large Objects in the "Embedded SQL" chapter, and the *Embedded SQL Companion Guide*.

Restrictions on Long Nvarchar Columns

The following restrictions apply to long nvarchar columns:

- They cannot be part of a table key.
- They do not declare a length.
- They cannot be part of a secondary index.
- They cannot be used in the order by group, or by clause in a select statement.
- They cannot have query optimization statistics. For details about query optimization statistics, see the discussion of the optimizedb utility in the *Command Reference Guide*.
- The following string functions do not work with long nvarchar columns:
  - Locate
  - Pad
  - Shift
  - Squeeze
  - Trim
  - Notrim
  - Charextract
- These columns cannot be directly compared to other string data types. To compare a long nvarchar column to another string data type, apply a coercion function.
- A string literal of more than 1000 characters cannot be assigned to a long nvarchar column. Details about assigning long strings to these columns are found in the description of data handlers in the *Embedded SQL Companion Guide* or the *OpenAPI User Guide*.

Numeric Data Types

OpenSQL has two categories of numeric data types: *exact* and *approximate*. The exact numeric data types are the integer data types and the decimal data type. The approximate numeric data types are the floating-point data types.

Integer Data Type

There are four integer data types:

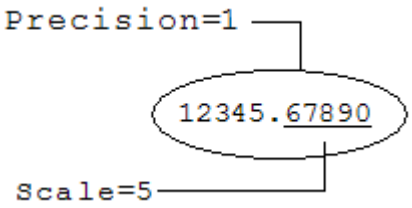
- tinyint (one-byte)
- smallint (two-byte)
- integer (four-byte)
- bigint (eight-byte)

The following table lists the ranges of values for each integer data type:

Integer Data Type	Lowest Possible Value	Highest Possible Value
tinyint (integer1)	-128	+127
smallint (integer2)	-32,768	+32,767
integer (integer4)	-2,147,483,648	+2,147,483,647
bigint (integer8)	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

Decimal Data Type

The decimal data type is an exact numeric data type defined in terms of its *precision* (total number of digits) and *scale* (number of digits to the right of the decimal point). The following figure illustrates precision and scale in decimal values:



The minimum precision for a decimal value is 1 and the maximum is 39. The scale of a decimal value cannot exceed its precision. Scale can be 0 (no digits to the right of the decimal point).

Specify the declaration using the following syntax:

decimal(*p*,*s*)

where

p=precision

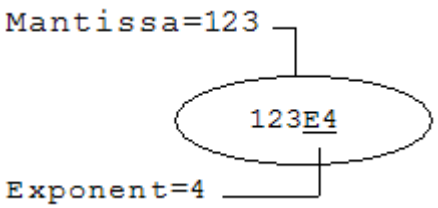
s=scale

Valid synonyms for the decimal data type are dec and numeric.

Note: The decimal data type is suitable for storing currency data. Note that, for display purposes, a currency sign cannot be specified for decimal values.

Floating-point Data Type

A floating-point value is represented either as whole plus fractional digits (like decimal values) or as a mantissa plus an exponent. The following figure illustrates the mantissa and exponent parts of floating-point values:



There are two floating-point data types:

- real (4-byte)
- float (8-byte)

Note: Float4 is a synonym for real. Float8 and double precision are synonyms for float.

Floating-point numbers are double-precision quantities stored in four or eight bytes. The range of float values is processor-dependent, and the precision is approximately 16 significant digits.

Specify the precision (number of significant bits) for a floating-point value using the following (optional) syntax:

float(*n*)

where *n* is a value from 0 to 53.

OpenSQL allocates storage according to the precision you specify, depending on the host DBMS and hardware. For information about the correct notation for a floating-point numeric literal, see Numeric Literals (see page 49).

Abstract Data Types

The abstract data types include the date and money data types.

Date Data Type

OpenSQL supports date data types for sessions connected to:

- The Ingres DBMS
- An Enterprise Access product to a host DBMS that supports date data types (for example, DB2 UDB, Oracle, or Rdb)

If the host DBMS supports date data types, the iidbcapabilities standard catalog table includes a row where cap_capability is set to OPEN_SQL_DATES, and cap_value is set to LEVEL 1.

Tables created in OpenSQL with date columns are mapped to the date format of the host DBMS. For example, OpenSQL date would map to Rdb date and to IBM timestamp.

On input, date constants in queries must be specified using the OpenSQL date() function.

OpenSQL supports the following operations on date data:

- Ordering on date columns
- Comparing two date columns
- Comparing a date column to a date constant

Absolute Date Input Formats

Dates are specified as quoted character strings. A date can be entered by itself or together with a time value. For more information about date and time display, see Date and Time Display Formats (see page 45) in this chapter.

The legal formats for absolute date values are determined by the II_DATE_FORMAT setting, summarized in the following table. If II_DATE_FORMAT is not set, the US formats are the default input formats. II_DATE_FORMAT can be set on a session basis. For information on setting II_DATE_FORMAT, see the *System Administrator* Guide.

II_DATE_FORMAT Setting	**Valid Input Formats**	**Output**
US (default format)	*mm/dd/yy* *mm/dd/yyyy* *dd-mmm-yyyy* *mm-dd-yyyy* *yyyy.mm.dd* *yyyy_mm_dd* *mmddyy* *mm-dd* *mm/dd*	*dd-mmm-yyyy*
MULTINATIONAL	*dd/mm/yy* all US formats except *mm/dd/yyyy*	*dd/mm/yy*
ISO	*yymmdd* *ymmdd* *yyyymmdd* *mmdd* *mdd* all US input formats except *mmddyy*	*yymmdd*

II_DATE_FORMAT Setting	Valid Input Formats	Output
SWEDEN/FINLAND	*yyyy-mm-dd* all US input formats except *mm-dd-yyyy*	*yyyy-mm-dd*
GERMAN	*dd.mm.yyyy* *ddmmyy* *dmmyy* *dmmyyyy* *ddmmyyyy* and all US input formats except *yyyy.mm.dd* and *mmddyy*	*dd.mm.yyyy*
YMD	*mm/dd* *yyyy-mm-dd* *mmdd* *yymdd* *yymmdd* *yyyymdd* *yyyymmdd* *yyyy-mmm-dd*	*yyyy-mmm-dd*
DMY	*dd/mm* *dd-mm-yyyy* *ddmm* *ddmyy* *ddmmyy* *ddmyyyy* *ddmmyyyy* *dd-mmm-yyyy*	*dd-mmm-yyyy*
MDY	*mm/dd* *dd-mm-yyyy* *mmdd* *mddyy* *mmddyy* *mddyyyy* *mmddyyyy* *mmm-dd-yyyy*	*mmm-dd-yyyy*

Year defaults to the current year. In formats that include delimiters (such as forward slashes or dashes), specify the last two digits of the year. The first two digits default to the current century (2000). For example, if you enter the following date:

```
'03/21/00'
```

using the format *mm/dd/yyyy*, OpenSQL assumes that you are referring to March 21, 2000.

In three-character month formats, for example, *dd-mmm-yy*, OpenSQL requires three-letter abbreviations (for example, mar, apr, may).

To specify the current system date, use the constant today. For example:

```
select date('today');
```

To specify the current system time, use the constant now.

Absolute Time Input Formats

The legal format for inputting an absolute time is

'*hh*:*mm*[:*ss*] [**am**|**pm**] [*timezone*]'

Input formats for absolute times are assumed to be on a 24-hour clock. If a time is entered with an am or pm designation, then OpenSQL automatically converts the time to a 24-hour internal representation.

If *timezone* is omitted, OpenSQL assumes the local time zone designation. Times are displayed using the time zone adjustment specified by II_TIMEZONE_NAME. For details about time zone settings and valid time zones, see the *Getting Started* guide.

If an absolute time is entered without a date, OpenSQL assumes the current system date.

Combined Date and Time Input

Any valid absolute date input format can be paired with a valid absolute time input format to form a valid date and time entry. The following table shows some examples of valid date and time entries using the US absolute date input formats:

Format	Example
mm/dd/yy hh:mm:ss	11/15/00 10:30:00
dd-mmm-yy hh:mm:ss	15-nov-98 10:30:00

Format	Example
mm/dd/yy hh:mm:ss	11/15/99 10:30:00
dd-mmm-yy hh:mm:ss gmt	15-nov-00 10:30:00 gmt
dd-mmm-yy hh:mm:ss [am|pm]	15-nov-98 10:30:00 am
mm/dd/yy hh:mm	11/15/99 10:30
dd-mmm-yy hh:mm	15-nov-00 10:30
mm/dd/yy hh:mm	11/15/98 10:30
dd-mmm-yy hh:mm	15-nov-99 10:30

Date and Time Display Formats

OpenSQL outputs date values as strings of 25 characters with trailing blanks inserted.

To specify the output format of an absolute date and time, II_DATE_FORMAT must be set. For a list of II_DATE_FORMAT settings and associated formats, see Absolute Date Input Formats (see page 42). The display format for absolute time is:

hh:mm:ss

OpenSQL displays 24-hour times for the current time zone, which is determined when OpenSQL is installed. Dates are stored in Greenwich Mean Time (GMT) and adjusted for your time zone when they are displayed.

If seconds are omitted when entering a time, OpenSQL displays zeros in the seconds' place.

Money Data Type

The money data type is an abstract data type. Money values are stored significant to two decimal places. These values are rounded to their amounts in dollars and cents or other currency units on input and output, and arithmetic operations on the money data type retain two-decimal-place precision.

Money columns can accommodate the following range of values:

$-999,999,999,999.99 to $999,999,999,999.99

A money value can be specified as either:

- **A character string literal**-The format for character string input of a money value is $*sddddddddddddd.dd*. The dollar sign is optional and the algebraic sign(s) defaults to + if not specified. There is no need to specify a cents value of zero (.00).
- **A number-**Any valid integer or floating point number is acceptable. The number is converted to the money data type automatically.

On output, money values display as strings of 20 characters with a default precision of two decimal places. The display format is:

$[-]ddddddddddddd.dd

where:

$ is the default currency symbol
d is a digit from 0 to 9

The following settings affect the display of money data. For details, see the *System Administrator Guide*.

Variable	Description
II_MONEY_FORMAT	Specifies the character displayed as the currency symbol. The default currency sign is the dollar sign ($). II_MONEY_FORMAT also specifies whether the symbol appears before of after the amount.
II_MONEY_PREC	Specifies the number of digits displayed after the decimal point; valid settings are 0, 1, and 2.
II_DECIMAL	Specifies the character displayed as the decimal point; the default decimal point character is a period (.). II_DECIMAL also affects FLOAT, FLOAT4, and the DECIMAL data types. **Note:** If II_DECIMAL is set to comma, be sure that when SQL syntax requires a comma (such as a list of table columns or SQL functions with several

Variable	Description
	parameters), that the comma is followed by a space. For example: select col1, ifnull(col2, 0), left(col4, 22) from t1:

Binary Data Types

Binary columns can contain data such as graphic images, which cannot easily be stored using character or numeric data types.

Long Byte Data Type

The long byte data type has the same characteristics as the byte varying data type, but can accommodate binary data up to 2 GB in length. In embedded SQL, *data handlers* can be created, which are routines to read and write the data for long byte columns. For details about data handlers, see Data Handlers for Large Objects in the chapter "Embedded SQL" and the *Embedded SQL Companion Guide*.

Storage Formats of Data Types

The following table lists storage formats for OpenSQL data types:

Notation	Type	Range
character(1) - character(*n*)	character	*n* represents the lesser of the maximum configured row size and 32,000.
varchar(1) - varchar(*n*)	character	*n* represents the lesser of the maximum configured row size and 32,000.
long varchar	character	A string of 1 to 2 GB characters
nchar	Unicode	A string of 1 to maximum configured row size, but not exceeding 16,000 characters (32,000 bytes).
nvarchar	Unicode	A string of 1 to maximum configured row size, but not exceeding 16,000 characters (32,000 bytes).
long nvarchar	Unicode	A string of 1 to a maximum of 1 GB Unicode characters (that is, 2 bytes to a maximum of 2 GB in length).
tinyint	1-byte integer	-128 to +127
smallint	2-byte integer	-32,768 to +32,767

Notation	Type	Range
integer	4-byte integer	-2,147,483,648 to +2,147,483,647
bigint	8-byte integer	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
decimal(*p, s)*	fixed-point exact numeric	Depends on precision and scale; default is (5,0): -99999 to +99999. Maximum number of digits is 39.
real	4-byte floating	-1.0e+38 to +1.0e+38 (7 digit precision)
float	8-byte floating	-1.0e+38 to +1.0e+38 (16 digit precision)
date	date (12 bytes)	1-jan-0001 to 30-dec-9999
money	money (8 bytes)	$-999,999,999,999.99 to $999,999,999,999.99
long byte	binary	1 to 2 GB of binary data

Note: If your hardware supports the IEEE standard for floating-point numbers, then the float type is accurate to 14 decimal precision ($-ddddddddddddd.dd to $+ddddddddddddd.dd) and ranges from -10**308 to +10**308.

Literals

A literal is an explicit representation of a value. OpenSQL supports two types of literals:

- String
- Numeric

String Literals

String literals are specified by one or more characters enclosed in single quotes. The default data type for string literals is varchar, but a string literal can be assigned to any character data type or to the money or date data types without using a data type conversion function.

To compare a string literal with a non-character data type (A), you must either cast the string literal to the non-character data type A, or cast the non-character data type to the string literal type. Failure to do so may result in unexpected results if the non-character data type contains the 'NUL (0) value.

Quotes in Strings

To include a single quote inside a string literal, it must be doubled. For example:

```
'The following letter is quoted: ''A''.'
```

which evaluates to

```
The following letter is quoted: 'A'.
```

Numeric Literals

Numeric literals specify numeric values. There are three types of numeric literals:

- Integer
- Decimal
- Floating-point

A numeric literal can be assigned to any of the numeric data types or the money data type without using an explicit conversion function. OpenSQL automatically converts the literal to the appropriate data type, if necessary.

By default, OpenSQL uses the period (.) to indicate the decimal when needed. This default can be changed by setting II_DECIMAL. For information about setting II_DECIMAL, see the *Database Administrator Guide*.

Note: If II_DECIMAL is set to comma, be sure that when OpenSQL syntax requires a comma (such as a list of table columns or OpenSQL functions with several parameters), that the comma is followed by a space. For example:

```
select col1, ifnull(col2, 0), left(col4, 22) from t1:
```

Integer Literals

Integer literals are specified by a sequence of up to 10 digits and an optional sign, in the following format:

[**+**|**-**] *digit* {*digit*} [**e** *digit*]

Integer literals are represented internally as either an integer or a smallint, depending on the value of the literal. If the literal is within the range -32,768 to +32,767, it is represented as a smallint. If its value is within the range -2,147,483,648 to +2,147,483,647 but outside the range of a smallint, then it is represented as an integer. Values that exceed the range of integers are represented as decimals.

Integers can be specified using a simplified scientific notation, similar to the way floating-point values are specified. To specify an exponent, follow the integer value with the letter "e" and the value of the exponent. This notation is useful for specifying large values. For example, to specify 100,000 use the following exponential notation:

```
1e5
```

Decimal Literals

Decimal literals are specified as signed or unsigned numbers of 1 to 39 digits that include a decimal point. The *precision* of a decimal number is the total number of digits, including leading and trailing zeros. The *scale* of a decimal literal is the total number of digits to the right of the decimal point, including trailing zeros. Decimal literals that exceed 39 digits are treated as floating-point values.

Examples of decimal literals are:

```
3.
-10.
1234567890.12345
001.100
```

Floating-point Literals

A floating-point literal must be specified using scientific notation. The format is:

[+|-] {*digit*} [.{*digit*}] e|E [+|-] {*digit*}

For example:

```
2.3e-02
```

At least one digit must be specified, either before or after the decimal point.

OpenSQL Constants

OpenSQL provides the following constants:

Special Constant	Meaning
now	Current date and time. Specify this constant in quotes only for servers and Enterprise Access products that support the date data type.
null	Indicates a missing or unknown value in a table.
today	Current date. Specify this constant in quotes. Valid only for servers and Enterprise Access products that support the date data type.
user	Effective user for the current session (the host DBMS user identifier, not the operating system user identifier).

These constants can be used in queries and expressions. For example:

```
select date('now');
insert into sales_order
(item_number, clerk, billing_date)
values ('123', user, date('today'));
```

Nulls

A null represents an undefined or unknown value and is specified by the keyword *null*. A null is not the same as a zero, a blank, or an empty string. A null can be assigned to any nullable column when no other value is specifically assigned. More information about defining nullable columns is provided in the Create Table section in the "OpenSQL Statements" chapter.

The IS NULL predicate allows nulls to be handled in queries. For details, see Exists Predicate in the "OpenSQL Statements" chapter.

Nulls and Comparisons

Because a null is not a value, it cannot be compared to any other value (including another null value). For example, the following where clause evaluates to false if one or both of the columns is null:

```
where columna = columnb
```

Similarly, the where clause

```
where columna < 10 or columna >= 10
```

is true for all numeric values of columna, but false if columna is null. The one exception, count(), is described in the next section.

Nulls and Aggregate Functions

When executing an aggregate function against a column that contains nulls, the function ignores the nulls. This prevents unknown or inapplicable values from affecting the result of the aggregate.

For example, if you apply the aggregate function, avg(), to a column that holds the ages of your employees, you want to be sure that any ages that have not been entered in the table are not treated as zeros by the function. This would distort the true average age. If a null is assigned to any missing ages, then the aggregate returns a correct result: the average of all known employee ages.

Aggregate functions, except count(), return null for an aggregate over an empty set, even when the aggregate includes columns that are not nullable (in this case, count() returns 0). In the following example, the select returns null, since there are no rows in test:

```
create table test (col1 integer not null);
select max(col1) as x from test;
```

When specifying a column that contains nulls as a grouping column (that is, in the group by clause) for an aggregate function, OpenSQL considers all nulls in the column as equal for the purposes of grouping. This is the one exception to the rule that nulls are not equal to other nulls. For information about the group by clause, see The Group By Clause in the “OpenSQL Statements” chapter.

Chapter 4: Elements of OpenSQL Statements

This section contains the following topics:

This chapter identifies the differences in syntax between embedded and interactive OpenSQL. If the embedded syntax is dependent on the host language, you are referred to the *Embedded SQL Companion Guide*.

Operators

OpenSQL supports three types of operators:

- Arithmetic
- Comparison
- Logical

Arithmetic Operators

Arithmetic operators are used to combine numeric expressions arithmetically to form other numeric expressions. Valid OpenSQL arithmetic operators are (in descending order of precedence):

Arithmetic Operator	Description
+ and -	plus, minus (unary)
* and /	multiplication, division (binary)
+ and -	addition, subtraction (binary)

Unary operators group from right to left and binary operators group from left to right. The unary minus (-) can be used to reverse the algebraic sign of a value.

Use parentheses to force the desired order of precedence. For example:

```
(job.lowsal + 1000) * 12
```

is an expression in which the parentheses force the addition operator (+) to take precedence over the multiplication operator (*).

Comparison Operators

Comparison operators allow you to compare two expressions.

OpenSQL includes the following comparison operators:

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

All comparison operators are of equal precedence.

The equal sign (=) also serves as the assignment operator in assignment operations. For details, see Assignment Operations (see page 58).

Logical Operators

OpenSQL has three logical operators:

- Not (highest precedence)
- And (next precedence)
- Or (lowest precedence)

Parentheses can be used to change the precedence. For example, assume that the following appears in a query:

exprA **or** *exprB* **and** *exprC*

OpenSQL evaluates the above as if it were:

exprA **or** (*exprB* **and** *exprC*)

However, by using parentheses, the order in which OpenSQL evaluates the expressions can be changed. For example:

(*exprA* **or** *exprB*) **and** *exprC*

When parenthesized as shown, (*exprA* or *exprB)* is evaluated first, then the operator is used for that result with *exprC*.

Operations

The following basic operations can be performed:

- Assignment operations
- Arithmetic operations

Assignment Operations

An assignment operation is an operation that places a value in a column or variable. Assignment operations occur during the execution of insert, update, fetch, create table as...select, and embedded select statements.

When an assignment operation occurs, the data types of the assigned value and the receiving column or variable must either be the same or compatible. If the data types are compatible but not the same, OpenSQL performs a default type conversion.

All character data types are compatible with one another. A value from a string can be assigned to a date data item if the value in the string is formatted in a valid OpenSQL date input format. For information about valid input formats, see Absolute Date Input Formats in the chapter "OpenSQL Data Types."

Money is compatible with all of the numeric and string types.

All numeric types are compatible with one another. For example, assuming that the following table is created:

```
create table emp
(name     character(20),
salary    float not null,
hiredate  date not null);
```

then this insert statement

```
insert into emp (name, salary, hiredate)
values ('John Smith', 40000, date('10/12/98'));
```

assigns the varchar string literal, 'John Smith', to the character name column, the integer literal 40000 to the float salary column, and the varchar string literal '10/12/98' to the date column, hiredate.

Other examples of assignments are:

```
update emp set name = 'Mary Smith'
where name = 'Mary Jones';
create table emp2 (name2, hiredate2) as
select name, hiredate from emp;
```

In the following embedded OpenSQL example, the value in the name column is assigned to the variable, name_var, for each row that fulfills the where clause:

```
exec sql select name into :name_var from emp
where empno = 125;
```

The following sections present some specific guidelines for assignments into each of the general data types, as well as null assignments. In addition, see the *Embedded* SQL *Companion Guide* for information about which host language data types are compatible with which OpenSQL data types if you are assigning to a host language variable.

Character String Assignment

The character and varchar character types are compatible. Any character string can be assigned to any column or variable of character data type. (If an assignment results in truncation, OpenSQL returns a warning.) The result of the assignment depends on the types of the assignment string and the receiving column or variable:

- If a character string is assigned to a varchar column or variable, trailing blanks are trimmed from the character string before it is assigned.

  If the length of the receiving string is shorter than the fixed length string, OpenSQL truncates the fixed length string (from the right end) and, if the assignment was to a variable, a warning condition is indicated. For a discussion of the SQLWARN indicators, see The SQL Communications Area (SQLCA) in the chapter "OpenSQL Features."

- If a string is assigned to a column or variable that is shorter than the fixed-length string, OpenSQL truncates the fixed-length string from the right end. If a fixed-length string is assigned to a fixed-length column or variable that is longer than the fixed-length string, OpenSQL pads it with blanks. If the assignment is to a variable and the string is truncated, a warning is indicated in the SQLCA.

Numeric Assignment

Any numeric data type can be assigned to any other numeric data type. In addition, a money value can be assigned to any numeric data type. OpenSQL may truncate leading zeros or all or part of the fractional part of a number if necessary. If it is necessary to truncate the non-fractional part of a value (other than leading zeros), an overflow error results. When a float or decimal value is assigned to an integer column or variable, the fractional part is truncated.

Date Assignment

Date values can be assigned to a date column. In addition, a string literal, a string host variable, or a string column value can be assigned to a date column if its value conforms to the valid OpenSQL input formats for dates.

When assigning character strings to date columns in OpenSQL, specify the string using the date() function. For example:

```
insert into transaction_log (employee, trxtime,
     trxid) values (user, date('now'), 42);
```

When assigning a date value to a character string, OpenSQL converts the date to the standard OpenSQL output date format. For more information about date output formats, see Date and Time Display Formats in the chapter "OpenSQL Data Types."

Null Assignment

A null can be assigned to a column of any data type if the column was defined as a nullable column. A null can also be assigned to a host language variable if there is an indicator variable associated with the host variable. For more information about indicator variables, see Indicator Variables in the chapter "Embedded OpenSQL."

Arithmetic Operations

An arithmetic operation combines two or more expressions using the arithmetic operators to form a resulting numeric expression.

Before performing any arithmetic operation, OpenSQL converts the participating expressions to identical data types. The result is returned as the selected data type. The following sections describe this data type conversion.

Default Type Conversion

When two numeric expressions are combined, the Enterprise Access product converts as necessary to make the data types of the expressions identical and assigns that same data type to the resulting expression. If it is necessary to convert the data type of an expression, the DBMS converts the expression having the data type of lower precedence to that of the higher.

The order of precedence among the numeric data types is, in highest-to-lowest order:

- Money
- Float
- Real
- Decimal
- Integer
- Smallint

For example, when OpenSQL operates on an integer and a floating-point number, the integer is converted to a floating-point number. If OpenSQL operates on two integers of different sizes, the smaller is converted to the size of the larger. All conversions are done before the operation is performed.

The following table summarizes the possible results of numeric combinations:

	smallint	**integer**	**decimal**	**real**	**float**	**money**
smallint	integer	integer	decimal	real	float	money
integer	integer	integer	decimal	real	float	money
decimal	decimal	decimal	decimal	real	float	money
real	real	real	real	real	float	money
float	float	float	float	float	float	money
money	money	money	money	money	money	money

For example, for this expression:

```
(job.lowsal + 1000) * 12
```

the first operator (+) combines a float expression (job.lowsal) with a smallint constant (1000). The result is float. The second operator (*) combines the float expression with a smallint constant (12), resulting in a float expression.

For money data type, if the above table conflicts with Host DBMS default type conversion, Host DBMS default type conversion has higher priority.

Arithmetic Operations on Decimal Data Types

In expressions that combine decimal values and return decimal results, the precision (total number of digits) and scale (number of digits to the right of the decimal point) of the result can be determined, as shown in the following table:

	Precision	Scale
Addition and subtraction	Larger number of fractional digits plus largest number of non-fractional digits + 1 (to a maximum of 39)	Scale of operand having the largest scale
Multiplication	Total of precisions to a maximum of 39	Total of scales to a maximum of 39
Division	39	(39 precision of first operand) + (scale of first operand) (scale of second operand)

For example, in the following decimal addition operation:

```
1.234 + 567.89
```

the scale and precision of the result is calculated as follows:

Precision = 7

Calculated as 3 (larger number of fractional digits) + 3 (larger number of non-fractional digits) + 1 = 7

Scale = 3

The first operand has the larger number of digits to the right of the decimal point

Result:

0569.124

Note: If the result of arithmetic using decimal data exceeds the declared precision or scale of the column to which it is assigned, OpenSQL truncates the result and does not issue an error.

Functions

OpenSQL provides scalar and aggregate functions.

Function Support for Enterprise Access Products

The overall level of function support for each Enterprise Access product is listed in the OPENSQL_SCALARS entry of the OpenSQL Standard Catalog Interface catalog. The Standard Catalog Interface catalog is a read-only view built on the system catalog of the underlying DBMS. The OPENSQL_SCALARS entry, located in the iidbcapabilities catalog section, can be one of three values: 'NATIVE', 'FULL' or 'LEVEL 1'. The default value is 'NATIVE'. 'NATIVE' indicates only native DBMS scalar functions are supported. 'FULL' indicates full Ingres function support is provided. 'LEVEL 1' indicates *some* mapping of Ingres functions. When OPENSQL_SCALARS is set to 'LEVEL 1', an additional table, iigwscalars, is provided which shows support details for individual functions.

To further determine the level of support provided for specific functions, see the documentation for your Enterprise Access product.

Scalar Functions

There are six types of scalar functions:

- Data type conversion
- Numeric
- String
- Date
- Bit-wise
- Random number

The scalar functions require either one or more single-value arguments. In most instances, scalar functions can be nested to any level. Certain restrictions apply when using some Enterprise Access products. For details, see the documentation provided with your Enterprise Access product.

Note: If II_DECIMAL is set to comma, be sure that when OpenSQL syntax requires a comma (such as a list of table columns or OpenSQL functions with several parameters), that the comma is followed by a space.

For example:

```
Select col1, ifnull(col2, 0), left(col4, 22) from t1:
```

Data Type Conversion Functions

The following table lists the data type conversion functions. (When converting decimal values to strings, the length of the result depends on the precision and scale of the decimal column.)

Name	Operand Type	Result Type	Description
byte(*expr* [, *len*])	any	byte	Converts the expression to byte binary data. If the optional length argument is specified, the function returns the leftmost *len* bytes. *Len* must be a positive integer value that does not exceed the length of the *expr* argument.
c(*expr* [, *len*])	any	c	Converts argument to c string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
char(*expr* [, *len*])	any	char	Converts argument to char string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive

Name	Operand Type	Result Type	Description
			integer value that does not exceed the length of the *expr* string.
date(*expr*)	c, text, char, varchar	date	Converts a c, char, varchar or text string to internal date representation.
decimal(*expr* [,*precision*[,*scale*]])	any except date	decimal	Converts any numeric expression to a decimal value. If *scale* (number of decimal digits) is omitted, the scale of the result is 0. If *precision* (total number of digits) is omitted, the precision of the result is determined by the data type of the operand, as follows: Operand Datatype — Default Precision smallint 5 integer1 5 integer 11 float 15 float4 15 decimal 15 money 15 Decimal overflow occurs if the result contains more digits to the left of the decimal point than the specified or default precision and scale can accommodate.
dow(*expr*)	date	c	Converts an absolute date into its day of week (for example, 'Mon,' 'Tue'). The result length is 3.
float4(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	float4	Converts the specified expression to float4.
float8(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	float	Converts the specified expression to float.

Name	Operand Type	Result Type	Description
hex(*expr*)	varchar, char, c, text	varchar	Returns the hexadecimal representation of the argument string. The length of the result is twice the length of the argument, because the hexadecimal equivalent of each character requires two bytes. For example, hex('A') returns '61' (ASCII) or 'C1' (EBCDIC).
int1(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	integer1	Converts the specified expression to integer1. Decimal and floating-point values are truncated. Numeric overflow will occur if the integer portion of a floating-point or decimal value is too large to be returned in the requested format.
int2(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	smallint	Converts the specified expression to smallint. Decimal and floating-point values are truncated. Numeric overflow will occur if the integer portion of a floating-point or decimal value is too large to be returned in the requested format.
int4(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	integer	Converts the specified expression to integer. Decimal and floating-point values are truncated. Numeric overflow will occur if the integer portion of a floating-point or decimal value is too large to be returned in the requested format.
long_byte (*expr*)	any	long byte	Converts the expression to long byte binary data.
long_varchar (*expr*)	any	long varchar	Converts the expression to a long varchar.
money(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	money	Converts the specified expression to internal money representation. Rounds floating-point and decimal values, if necessary.

Name	Operand Type	Result Type	Description
nchar(*expr* [, *len*])	any	nchar	Converts argument to nchar unicode string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
nvarchar(*expr* [, *len*])	any	nvarchar	Converts argument to nvarchar unicode string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
long_varchar (*expr*)	c, char, varchar, text, long varchar, long byte	long varchar	Converts the expression to a long varchar.
object_key(*expr*)	varchar, char, c, text	object_key	Converts the operand to an object_key.
table_key(*expr*)	varchar, char, c, text	table_key	Converts the operand to a table_key.
text(*expr* [, *len*])	any	text	Converts argument to text string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
unhex(*expr*)	varchar, c, text	varchar	Returns the opposite of the hex function. For example, unhex(x'61626320') returns 'abc' and unhex(x'01204161') returns '\001Aa'. Exceptions can occur when a "c" data type suppresses the display of certain stored characters, or when the output data type differs from the input type. **Note:** Typically one character is generated for every two hex digits being converted to a printable character. If the hex digit pair being converted does not translate to a printable character, then the value is converted to a backslash (\), followed by the numeric value of the hex digit pair as a three-digit octal value.

Name	Operand Type	Result Type	Description
varbyte(*expr* [, *len*])	any	byte varying	Converts the expression to byte varying binary data. If the optional length argument is specified, the function returns the leftmost *len* bytes. *Len* must be a positive integer value that does not exceed the length of the *expr* argument.
varchar(*expr* [, *len*])	any	varchar	Converts argument to varchar string. If the optional length argument is specified, the function returns the leftmost *len* characters. *Len* must be a positive integer value that does not exceed the length of the *expr* string.

If the optional length parameter is omitted, the length of the result returned by the data type conversion functions c(), char(), varchar(), and text() are as follows:

Data Type or Argument	Result Length
byte	Length of operand
byte varying	Length of operand
c	Length of operand
char	Length of operand
date	25 characters
decimal	Depends on precision and scale of column
float & float4	11 characters; 12 characters on IEEE computers
integer1 (smallint)	6 characters
integer	6 characters
integer4	13 characters
long varbyte	Length of operand
long varchar	Length of operand
money	20 characters
text	Length of operand
varchar	Length of operand

Numeric Functions

OpenSQL supports the numeric functions listed in the following table:

Name	Operand Type	Result Type	Description
abs(*n*)	all numeric types and money	same as *n*	Absolute value of *n*.
atan(*n*)	all numeric types	float	Arctangent of *n*; returns a value from (-pi/2) to pi/2.
cos(*n*)	all numeric types	float	Cosine of *n*; returns a value from -1 to 1.
exp(*n*)	all numeric types and money	float	Exponential of *n*.
log(*n*) ln(*n*)	all numeric types and money	float	Natural logarithm of *n*.
mod(*n*,*b*)	integer, smallint, integer1, decimal	same as *b*	*n* modulo *b*. The result is the same data type as *b*. Decimal values are truncated.
power(*x*,*y*)	all numeric types	float	*x* to the power of *y* (identical to x ** y)
sin(*n*)	all numeric types	float	Sine of *n*; returns a value from -1 to 1.
sqrt(*n*)	all numeric types and money	float	Square root of *n*.

For trigonometric functions (atan(), cos(), and sin()), specify arguments in radians. To convert degrees to radians, use the following formula:

```
radians = degrees/360 * 2 * pi
```

To obtain a tangent, divide sin() by cos().

String Functions

String functions perform a variety of operations on character data. String functions can be nested. For example:

```
left(right(x.name, size(x.name) - 1), 3)
```

returns the substring of x.name from character positions 2 through 4, and

```
concat(concat(x.lastname, ', '), x.firstname)
```

concatenates x.lastname with a comma and then concatenates x.firstname with the first concatenation result. The **+** operator can also be used to concatenate strings:

```
x.lastname + ', ' + x.firstname
```

The following string functions do not accept long varchar or long byte columns:

- Locate
- Pad
- Shift
- Squeeze
- Trim
- Notrim
- Charextract

To apply any of the preceding functions to a long varchar or long byte column, first coerce the column to an acceptable data type. For example:

```
squeeze(varchar(long_varchar_column))
```

If a coercion function is applied to a long varchar or long byte value that is longer than 2008 characters or bytes, the result is truncated to 2008 characters or bytes.

The following table lists the string functions supported in OpenSQL. The expressions c1 and *c2*, representing the arguments, can be any of the string types, except where noted. The expressions *len* and *nshift* represent integer arguments.

Name	Result Type	Description
charextract(*c1*,*n*)	varchar	Returns the *n*th byte of *c1*. If *n* is larger than the length of the string, then the result is a blank character.

Name	Result Type	Description
concat(*c1*,*c2*)	any character data type, byte	Concatenates one string to another. The result size is the sum of the sizes of the two arguments. If the result is a c or char string, it is padded with blanks to achieve the proper length. To determine the data type results of concatenating strings, see the table regarding results of string concatenation.
left(*c1*,*len*)	any character data type	Returns the leftmost *len* characters of *c1*. If the result is a fixed-length c or char string, it is the same length as *c1*, padded with blanks. The result format is the same as *c1*.
length(*c1*)	smallint (for long varchar, returns 4-byte integer)	If *c1* is a fixed-length c or char string, returns the length of *c*1 without trailing blanks. If *c*1 is a variable-length string, returns the number of characters actually in *c1*.
locate(*c1*,*c2*)	smallint	Returns the location of the first occurrence of *c2* in *c1*, including trailing blanks from *c2*. The location is in the range 1 to size(*c1*). If *c2* is not found, the function returns size(*c1*) + 1. The function size() is described below, in this table. If *c1* and *c2* are different string data types, *c2* is coerced into the *c1* data type.
lowercase(*c1*) or lower(*c1*)	any character or Unicode data type	Converts all upper case characters in *c1* to lower case.
pad(*c1*)	text or varchar	Returns *c1* with trailing blanks appended to *c1*; for instance, if *c1* is a varchar string that could hold fifty characters but only has two characters, then pad(*c1)* appends 48 trailing blanks to *c1* to form the result.

Name	Result Type	Description
right(*c1*,*len*)	any character data type	Returns the rightmost *len* characters of *c1*. Trailing blanks are not removed first. If *c1* is a fixed-length character string, the result is padded to the same length as *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
shift(*c1*,*nshift*)	any character data type	Shifts the string *nshift* places to the right if *nshift* > 0 and to the left if *nshift* < 0. If *c1* is a fixed-length character string, the result is padded with blanks to the length of *c1*. If *c1* is a variable-length character string, no padding occurs. The result format is the same as *c1*.
size(*c1*)	smallint	Returns the *declared* size of *c1* without removal of trailing blanks.
soundex(*c1*)	any character data type	Returns a *c1* four-character field that can be used to find similar sounding strings. For example, SMITH and SMYTHE produce the same soundex code. If there are less than three characters, the result is padded by trailing zero(s). If there are more than three characters, the result is achieved by dropping the rightmost digit(s). This function is useful for finding like-sounding strings quickly. A list of similar sounding strings can be shown in a search list rather than just the next strings in the index.
squeeze(*c1*)	text or varchar	Compresses white space. White space is defined as any sequence of blanks, null characters, newlines (line feeds), carriage returns, horizontal tabs and form feeds (vertical tabs). Trims white space from the beginning and end of the string, and replaces all other white space with single blanks.

Name	Result Type	Description
		This function is useful for comparisons. The value for *c1* must be a string of variable-length character string data type (not fixed-length character data type). The result is the same length as the argument.
substring(*c1* from *loc*[FOR l*en*])	varchar	Selects part of *c1* starting at the *loc* position and either extending to the end of the string or for the number of characters in the *len* operand. The result format is a varchar the size of *c1*.
substring(*c1* from n1 [for *n*2])	varchar or nvarchar	Returns a substring of parameter *c1* starting at offset *n1*. If *n2* is specified, the resulting string is min(*n2*, length(*c1*)-*n1*) in length. If *n1* is 0 or negative, the resulting substring starts with the 1st byte of *c1*. If *n1* > length(*c1*), the resulting string has length 0. If *n2* is negative, an error is returned.
trim(*c1*)	text or varchar	Returns *c1* without trailing blanks. The result has the same length as *c1*.
notrim(*c1*)	any character string variable	Retains trailing blanks when placing a value in a varchar column. This function can only be used in an embedded OpenSQL program. For more information, see the *Embedded SQL Companion Guide*.
uppercase(*c1*) or upper(*c1*)	any character or Unicode data type	Converts all lower case characters in *c1* to upper case.

String Concatenation Results

The following table shows the results of concatenating expressions of various character data types:

1st String	2nd String	Trim Blanks		Result Type
		from 1st?	from 2nd?	
c	c	Yes	--	c
c	text	Yes	--	c
c	char	Yes	--	c
c	varchar	Yes	--	c
c	long varchar	Yes	No	long varchar
text	c	No	--	c
char	c	Yes	--	c
varchar	c	No	--	c
long varchar	c	No	No	long varchar
text	text	No	No	text
text	char	No	Yes	text
text	varchar	No	No	text
text	long varchar	No	No	long varchar
char	text	Yes	No	text
varchar	text	No	No	text
long varchar	text	No	No	long varchar
char	char	No	--	char
char	varchar	No	--	char
char	long varchar	No	No	long varchar
varchar	char	No	--	char
long varchar	char	No	No	long varchar
varchar	varchar	No	No	varchar
long varchar	long varchar	No	No	long varchar

When concatenating more than two operands, expressions are evaluated from left to right. For example:

```
varchar + char + varchar
```

is evaluated as:

```
(varchar+char)+varchar
```

To control concatenation results for strings with trailing blanks, use the trim, notrim, and pad functions.

Date Functions

OpenSQL supports functions that derive values from absolute dates and from interval dates. These functions operate on columns that contain date values. An additional function, dow(), returns the day of the week (mon, tue, and so on) for a specified date. For a description of the dow() function, see Data Type Conversion Functions (see page 64).

Some date functions require you to specify a unit parameter; unit parameters must be specified using a quoted string. The following table lists valid unit parameters:

Date Portion	How Specified
Second	second, seconds, sec, secs
Minute	minute, minutes, min, mins
Hour	hour, hours, hr, hrs
Day	day, days
Week	week, weeks, wk, wks
ISO-Week	iso-week, iso-wk
Month	month, months, mo, mos
Quarter	quarter, quarters, qtr, qtrs
Year	year, years, yr, yrs

The following table lists the date functions:

Name	Format (Result)	Description
date_trunc(*unit*,*date*)	date	Returns a date value truncated to the specified *unit*.
date_part(*unit*,*date*)	integer	Returns an integer containing the specified (*unit*)

Name	Format (Result)	Description
		component of the input date.
date_gmt(*date*)	any character data type	Converts an absolute date into the Greenwich Mean Time character equivalent with the format *yyyy_mm_dd hh:mm:ss* GMT. If the absolute date does not include a time, the time portion of the result is returned as 00:00:00. For example, the query: select date_gmt('1-1-98 10:13 PM PST') returns the following value: 1998_01_01 06:13:00 GMT while the query: select date_gmt('1-1-1998') returns: 1998_01_01 00:00:00 GMT
gmt_timestamp(*s*)	any character data type	Returns a twenty-three-character string giving the date *s* seconds after January 1, 1970 GMT. The output format is '*yyyy_mm_dd hh:mm:ss* GMT'. For example, the query: select (gmt_timestamp (1234567890)) returns the following value: 2009_02_13 23:31:30 GMT while the query: (II_TIMEZONE_NAME = AUSTRALIA_ QUEENSLAND) select date(gmt_timestamp (1234567890)) returns: 14-feb-2009 09:31:30
interval (*unit*,*date_interval*)	float	Converts a date interval into a floating-point constant expressed in the unit of measurement specified by unit. The interval function assumes that there are 30.436875 days per month and 365.2425 days per year when using the mos, qtrs, and yrs specifications. For example, the query: select(interval('days', '5 years')) returns the following value: 1826.213

Name	Format (Result)	Description
		This function is not supported for the Oracle and MS SQL Enterprise Access products.
_date(*s*)	any character data type	Returns a nine-character string giving the date *s* seconds after January 1, 1970 GMT. The output format is *dd-mmm-yy*. For example, the query: select _date(123456) returns the following value: 2-jan-70
_date4(*s*)	any character data type	Returns an eleven-character string giving the date *s* seconds after January 1, 1970 GMT. The output format is controlled by the II_DATE_FORMAT setting. For example, with II_DATE_FORMAT set to US, the query: select _date4(123456) returns the following value: 02-jan-1970 while with II_DATE_FORMAT set to MULTINATIONAL, the query: select _date4(123456) returns this value: 02/01/1970
_time(*s*)	any character data type	Returns a five-character string giving the time *s* seconds after January 1, 1970 GMT. The output format is *hh:mm* (seconds are truncated). For example, the query: select _time(123456) returns the following value: 02:17

Date_trunc Function

Use the date_trunc function to group all the dates in the same month or year, and so forth. For example:

```
date_trunc('month',date('23-oct-1998 12:33'))
```

returns 1-oct-1998, and

```
date_trunc('year',date('23-oct-1998'))
```

returns 1-jan-1998.

Truncation takes place in terms of calendar years and quarters (1-jan, 1-apr, 1-jun, and 1-oct).

To truncate in terms of a fiscal year, offset the calendar date by the number of months between the beginning of your fiscal year and the beginning of the next calendar year (6 mos for a fiscal year beginning July 1, or 4 mos for a fiscal year beginning September 1):

```
date_trunc('year',date+'4 mos') - '4 mos'
```

Weeks start on Monday. The beginning of a week for an early January date may fall into the previous year.

Date_part Function

This function is useful in set functions and in assuring correct ordering in complex date manipulation. For example, if date_field contains the value 23-oct-1998, then:

```
date_part('month',date(date_field))
```

returns a value of 10 (representing October), and

```
date_part('day',date(date_field))
```

returns a value of 23.

Months are numbered 1 to 12, starting with January.

Hours are returned according to the 24-hour clock.

Quarters are numbered 1 through 4.

Week 1 begins on the first Monday of the year. Dates before the first Monday of the year are considered to be in week 0. However, if you specify ISO-Week, which is ISO 8601 compliant, the week begins on Monday, but the first week is the week that has the first Thursday. The weeks are numbered 1 through 53.

Therefore, if you are using Week and the date falls before the first Monday in the current year, date_part returns 0. If you are using ISO-Week and the date falls before the week containing the first Thursday of the year, that date is considered part of the last week of the previous year, and date_part returns either 52 or 53.

The following table illustrates the difference between Week and ISO-Week:

Date Column	Day of Week	Week	ISO-Week
02-jan-1998	Fri	0	1
04-jan-1998	Sun	0	1
02-jan-1999	Sat	0	53
04-jan-1999	Mon	1	1
02-jan-2000	Sun	0	52
04-jan-2000	Tue	1	1
02-jan-2001	Tue	1	1
04-jan-2001	Thu	1	1

Bit-wise Functions

Bit-wise functions operate from right to left, with shorter operands padded with hex zeroes to the left. Each result is a byte field the size of the longer operand, except BIT_NOT, which takes a single byte operand and returns the same-sized operand.

The external bit-wise functions are:

BIT_ADD

Returns the logical "add" of two byte operands; any overflow is disregarded.

BIT_AND

Returns the logical "and" of two byte operands. For example, if two bits are 1, the answer is 1, otherwise the answer is 0.

BIT_NOT

Returns the logical "not" of two byte operands.

BIT_OR

Returns the logical "or" of two byte operands. For example, if either or both bits are 1, the answer is 1.

BIT_XOR

Returns the logical "xor" of two byte operands. For example, if either bit is 1, the answer is 1.

INTEXTRACT(b1,n)

Returns the nth byte of b1 (byte type) as an integer. If n is less than 1 or larger than the number of bytes in b1, 0 is returned. For example, if b1 is x'080309040a05', the value of intextract(b1, 5) is 10 and the value of intextract(b1, 20) is 0.

Hash Function

This function is used to generate a four-byte numeric value from expressions of all data types except long data types. Note that the implicit size for the expression can affect the result. For example:

```
select hash(1), hash(int1(1)), hash(int2(1)), hash(int4(1))\g
```

returns the following single row:

Col1	Col2	Col3	Col4
-920527466	1526341860	-920527466	-1447292811

Note: Since the constant 1 is implicitly a short integer, only the return values for Hash(1) and Hash(int2(1)) match. For the remaining columns, the difference in the number of bytes holding the integer leads to a different hash value. Also note that the generated hash value is not guaranteed unique, even if the input values are unique. The hash function is not supported for Enterprise Access products.

Random Number Function

The random number function is used to generate random values. Use the following statement to set the beginning value for the random functions:

```
[exec sql] set random_seed [value]
```

There is a global seed value and local seed values. The global value is used until you issue "set random_seed," which changes the value of the local seed. Once changed, the local seed is used for the whole session. If you are using the global seed value, the seed is changed whenever a random function executes. This means that other users issuing random calls will enhance the "randomness" of the returned value. Note that the seed value can be any integer.

If you omit the value, then Ingres multiplies the process ID by the number of seconds past 1/1/1970 until now. This value generates a random starting point. You can use value to run a regression test from a static start and get identical results.

There are four random number functions:

- random() - Returns a random integer based on a seed value.
- randomf() - Returns a random float based on a seed value between 0 and 1. This is slower than random, but produces a more random number.
- random(l,h) - Returns a random integer in the specified range (that is, l >= x <= h).
- randomf(l,h) - Passing two integer values generates an integer result in the specified range; passing two floats generates a float in the specified range; passing an int and a float causes them to be coerced to an int and generates an integer result in the specified range (that is, l >= x <= h).

Aggregate Functions

Aggregate functions include the following:

- Unary
- Binary
- Count

Unary Aggregate Functions

A unary aggregate function returns a single value based on the contents of a column. Aggregate functions are also called *set* functions.

Note: For OpenROAD users, aggregate functions used within OpenROAD can only be coded inside SQL statements.

The following example uses the sum aggregate function to calculate the total of salaries for employees in department 23:

```
select sum (employee.salary)
    from employee
    where employee.dept = 23;
```

The following table lists SQL aggregate functions:

Name	Result Data Type	Description
any	integer	Returns 1 if any row in the table fulfills the where clause, or 0 if no rows fulfill the where clause.
avg	float, money, date (interval only)	Average (sum/count) The sum of the values must be within the range of the result data type.
count	integer	Count of non-null occurrences
max	same as argument	Maximum value
min	same as argument	Minimum value
sum	integer, float, money, date (interval only)	Column total
stddev_pop	float	Compute the population form of the standard deviation (square root of the population variance of the group).
stddev_samp	float	Computes the sample form of the standard deviation (square root of the sample variance of the group).
var_pop	float	Computes the population form of the variance (sum of the squares of the difference of each argument value in the group from the mean of the values, divided by the count of the

Name	Result Data Type	Description
		values).
var_samp	float	Computes the sample form of the variance (sum of the squares of the difference of each argument value in the group from the mean of the values, divided by the count of the values minus 1).

The general syntax of an aggregate function is as follows:

function_name ([**distinct | all**] *expr*)

where *function_name* denotes an aggregate function and *expr* denotes any expression that does not include an aggregate function reference (at any level of nesting).

To eliminate duplicate values, specify distinct. To retain duplicate values, specify all (this is the default.) Distinct is not meaningful with the functions min and max, because these functions return single values (and not a set of values).

Nulls are ignored by the aggregate functions, with the exception of count, as described in The Count Function and Nulls in this chapter.

Binary Aggregate Functions

Ingres supports a variety of binary aggregate functions that perform a variety of regression and correlation analysis.

For all of the binary aggregate functions, the first argument is the independent variable and the second argument is the dependent variable.

The following table lists binary aggregate functions:

Name	Result Data Type	Description
regr_count	integer	Count of rows with non-null values for both dependent and independent variables.
covar_pop	float	Population covariance (sum of the products of the difference of the independent variable from its mean, times the difference of the dependent variable from its mean, divided by the number of rows).

Name	Result Data Type	Description
covar_samp	float	Sample covariance (sum of the products of the difference of the independent variable from its mean, times the difference of the dependent variable from its mean, divided by the number of rows minus 1).
corr	float	Correlation coefficient (ratio of the population covariance divided by the product of the population standard deviation of the independent variable and the population standard deviation of the dependent variable).
regr_r2	float	Square of the correlation coefficient.
regr_slope	float	Slope of the least-squares-fit linear equation determined by the (independent variable, dependent variable) pairs.
regr_intercept	float	Y-intercept of the least-squares-fit linear equation determined by the (independent variable, dependent variable) pairs.
regr_sxx	float	Sum of the squares of the independent variable.
regr_syy	float	Sum of the squares of the dependent variable.
regr_sxy	float	Sum of the product of the independent variable and the dependent variable.
regr_avgx	float	Average of the independent variables.
regr_avgy	float	Average of the dependent variables.

Count(*) Function

Count can take the wildcard character, *, as an argument. This character is used to count the number of rows in a result table, including rows that contain nulls. For example, the statement:

```
select count(*)
    from employee
    where dept = 23;
```

counts the number of employees in department 23. The asterisk (*) argument cannot be qualified with all or distinct.

Because count(*) counts rows rather than columns, count(*) does not ignore nulls. Consider the following table:

Name	Exemptions
Smith	0
Jones	2
Tanghetti	4
Fong	Null
Stevens	Null

Running

```
count(exemptions)
```

returns the value of 3, whereas

```
count(*)
```

returns 5.

Except count, if the argument to an aggregate function evaluates to an empty set, the function returns a null. The count function returns a zero.

Aggregate Functions and Decimal Data

Given decimal arguments, aggregate functions (with the exception of count) return decimal results.

The following table explains how to determine the scale and precision of results returned for aggregates with decimal arguments:

Name	Precision of Result	Scale of Result
count	Not applicable	Not applicable
sum	39	Same as argument
avg	39	Scale of argument + 1 (to a maximum of 39)
max	Same as argument	Same as argument
min	Same as argument	Same as argument

Using Group By Clause with Aggregate Functions

The group by clause allows aggregate functions to be performed on subsets of the rows in the table. The subsets are defined by the group by clause. For example, the following statement selects rows from a table of political candidates, groups the rows by party, and returns the name of each party and the average funding for the candidates in that party.

```
select party, avg(funding)
    from candidates
    group by party;
```

Restrictions on the Use of Aggregate Functions

The following restrictions apply to the use of aggregate functions:

- Aggregate functions cannot be nested.
- Aggregate functions can only be used in select or having clauses.
- If a select or having clause contains an aggregate function, columns not specified in the aggregate must be specified in the group by clause. For example:

```
select dept, avg(emp_age)
from employee
    group by dept;
```

The above select statement specifies two columns, dept and emp_age, but only emp_age is referenced by the aggregate function, avg. The dept column is specified in the group by clause.

Ifnull Function

The ifnull function specifies a value other than a null that is returned to your application when a null is encountered. The ifnull function is specified as follows:

ifnull(*v1*,*v2*)

If the value of the first argument is not null, ifnull returns the value of the first argument. If the first argument evaluates to a null, ifnull returns the second argument.

For example, the sum, avg, max, and min aggregate functions return a null if the argument to the function evaluates to an empty set. To receive a value instead of a null when the function evaluates to an empty set, use the ifnull function, as in this example:

```
ifnull(sum(employee.salary)/25, -1)
```

Ifnull returns the value of the expression sum(employee.salary)/25 unless that expression is null. If the expression is null, the ifnull function returns -1.

Note that if an attempt is made to use the ifnull function with data types that are not nullable, such as system_maintained logical keys, a runtime error is returned.

Note: If II_DECIMAL is set to comma, be sure that when SQL syntax requires a comma (such as a list of table columns or SQL functions with several parameters), that the comma is followed by a space. For example:

```
select col1, ifnull(col2, 0), left(col4, 22) from t1:
```

Ifnull Result Data Type

If the arguments are of the same data type, the result is of that data type. If the two arguments are of different data types, they must be of comparable data types.

When the arguments are of different but comparable data types, the DBMS Server uses the following rules to determine the data type of the result:

- The result type is always the higher of the two data types; the order of precedence of the data types is as follows:

  date > money > float4 > float > decimal > integer > smallint > integer1

  and

  c > text > char > varchar > long varchar > byte > byte varying > long byte

- The result length is taken from the longest value. For example:

  ```
  ifnull (varchar (5), c10)
  ```

  results in c10.

The result is nullable if either argument is nullable. The first argument is not required to be nullable, though in most applications it is nullable.

IFNULL and Decimal Data

If both arguments to an IFNULL function are decimal, the data type of the result returned is decimal, and the precision (total number of digits) and scale (number of digits to the right of the decimal point) of the result is determined as follows:

- **Precision**—The largest number of digits to the left of the decimal point (precision - scale) plus largest scale (to a maximum of 39)
- **Scale**—The largest scale

Universal Unique Identifier (UUID)

A Universal Unique Identifier (UUID) is a 128 bit, unique identifier generated by the local system. It is unique across both space and time with respect to the space of all UUIDs.

A UUID can be used to tag records to ensure that the database records are uniquely identified regardless of which database they are stored in, for example, in a system where there are two separate physical databases containing accounting data from two different physical locations.

No centralized authority is responsible for assigning UUIDs. They can be generated on demand (10 million per second per machine if needed).

Purposes of a UUID

A UUID can be used for multiple purposes:

- Tagging objects that have a brief life
- Reliably identifying persistent objects across a network
- Assigning as unique values to transactions as transaction IDs in a distributed system

UUIDs are fixed-sized (128-bits), which is small relative to other alternatives. This fixed small size lends itself well to sorting, ordering, and hashing of all sorts, sorting in databases, simple allocation, and ease of programming.

UUID Format

The format of 128-bits (16 octets) UUID is:

Field	Data Type	Octet Number	Note
time_low	unsigned 32-bit integer	0-3	The low field of the timestamp
time_mid	unsigned 16-bit integer	4-5	Time middle field of the timestamp
time_hi_and_version	unsigned 16-bit integer	6-7	The high field of the timestamp multiplex with the release number
clock_seq_hi_and_reserved	unsigned 8-bit integer	8	The high field of the clock sequence multiplex with the variant
clock_seq_low	unsigned 8-bit integer	9	The low field of the clock sequence
node	unsigned 48-bit integer	10-15	The spatially unique node identifier

SQL Functions for UUID Implementation

Ingres implements the following SQL procedures to create, convert, and compare UUIDs:

- UUID_CREATE( )
- UUID_COMPARE(uuid1, uuid2)
- UUID_TO_CHAR(u)
- UUID_FROM_CHAR(c)

UUID_CREATE( ) Function

The UUID_CREATE() function creates a 128 bit UUID:

```
> createdb uuiddb
> sql uuiddb
* CREATE TABLE uuidtable (u1 BYTE (16), u2 BYTE(16));
* INSERT INTO uuidtable VALUES (UUID_CREATE(), UUID_CREATE());
//
// Verify the length in byte format
//
* SELECT LENGTH(u1) FROM uuidtable;
//
//Length returned equals 16 bytes
//
```

col1
16

UUID_COMPARE(uuid1, uuid2) Function

The UUID_COMPARE(uuid1, uuid2) function returns an integer value:

Return	Meaning
-1	uuid1 < uuid2
0	uuid1 == uuid2
+1	uuid1 > uuid2

```
//
// Determine if u1 is greater than, less than, or equal to u2
//
* SELECT UUID_COMPARE(u1, u2) FROM uuidtable;
```

col1
1

```
//
// u1 > u2
//
```

UUID_TO_CHAR(u) Function

The UUID_TO_CHAR(u) function converts a generated UUID into its character representation.

```
* SELECT UUID_TO_CHAR(u1) FROM uuidtable;
```

col1
2dd33cd2-b358-01d5-bf8d-00805fc13ce5

```
//
// Verify length of  UUID in  character format
//
* SELECT LENGTH(UUID_TO_CHAR(u1)) FROM uuidtable;
```

col1
36

```
//
//A UUID contains 36 characters
//
```

UUID_FROM_CHAR(c) Function

The UUID_FROM_CHAR(c) function converts a UUID from its character representation into byte representation:

```
//
// Insert a UUID in character format into a table
//

* CREATE TABLE uuidtochar
  AS
  SELECT UUID_TO_CHAR(u1) AS c1 FROM uuidtable;

* SELECT c1 FROM uuidtochar;
```

c1
f703c440-b35c-01d5-8637-00805fc13ce5

```
//
// convert UUID into byte representation
//

* SELECT UUID_FROM_CHAR(c1) FROM uuidtochar;
```

col1
œ\003Ä@ª\\001Ő\2067\221\0134¡\221\013

Expressions

Expressions are composed of various operators and operands that evaluate to either a single value or a set of values. Some expressions do not use operators. For example, a column name is an expression. Constants are expressions also. Expressions are used in many contexts, such as specifying values to be retrieved (in a select clause) or compared (in a where clause). For example:

```
select empname, empage from employee
     where salary     <String `75000'>  >
```

In this example, empname and empage are expressions representing the column values to be retrieved, salary is an expression representing a column value to be compared, and 75000 is an integer literal expression.

An expression can be enclosed in parentheses, such as ('J. J. Jones'), without affecting its meaning.

Predicates

Predicates are keywords that specify a relationship between two expressions:

expression_1 predicate expression_2

OpenSQL supports the following types of predicates:

- [not] like
- [not] between
- [not] in
- all | any | some
- exists
- is [not] null

The second expression can be a subquery. If the subquery does not return any rows, then the comparison evaluates to false. For details about subqueries, see Subqueries (see page 100).

Note: The is null predicate is the only predicate that can be used with long varchar and long byte data.

Like Predicate

The like predicate performs pattern matching for the character data types (character and varchar). The like predicate has the following syntax:

expression [NOT] LIKE *pattern* [ESCAPE *escape_character*]

The expression can be a column name or an expression involving string functions.

The *pattern* parameter must be a string literal. The pattern**-** matching (wild card) characters are the percent sign (%) to denote 0 or more arbitrary characters, and the underscore (_) to denote exactly one arbitrary character.

The like predicate does not handle trailing blanks. If matching a character data type or if the value has user-inserted trailing blanks, these trailing blanks must be included in your pattern. For example, if you are searching a character(10) column for any rows that have the name harold, use the following syntax for the like predicate:

```
name like 'harold    '
```

Four blanks are added to the pattern after the name in order to include the trailing blanks.

Because blanks are not significant when performing comparisons of c data types, the like predicate will return a correct result regardless of whether trailing blanks are included in the pattern.

If the escape clause is specified, the escape character suppresses any special meaning for the following character, allowing the character to be entered literally. The following characters can be escaped:

- The pattern matching characters % and _.
- The escape character itself. To enter the escape character literally, type it twice.
- Brackets [ ]. In escaped brackets ([and]), specify a series of individual characters or a range of characters separated by a dash (**-**).

The following examples illustrate some uses of the pattern matching capabilities of the like predicate:

To match any string starting with 'a':

```
name like 'a%'
```

To match any string starting with A through Z:

```
name like '\[A-Z\]%' escape '\'
```

To match any two characters followed by '25%':

```
name like '__25\%' escape '\'
```

To match a string starting with a backslash:

```
name like '\%'
```

Because there is no escape clause, the backslash is taken literally.

To match a string starting with a backslash and ending with a percent:

```
name like '\\%\%' escape '\'
```

To match any string starting with 0 through 4, followed by an uppercase letter, then a [, any two characters and a final]:

```
name like '\[01234\]\[A-Z\][__]' escape '\'
```

To detect names starting with 'S' and ending with 'h', disregarding any leading or trailing spaces:

```
trim(name) like 'S%h'
```

To detect a single quote, the quote must be repeated:

```
name like ''''
```

Between Predicate

The following table explains the operators between and not between:

Operator	Meaning
y between *x* and *z*	*x* < = *y* and *y* < = *z*
y not between *x* and *z*	not (y between *x* and *z)*

x, *y,* and *z* are expressions and cannot be subqueries.

In Operator

The following table explains the operators, in and not in:

Operator	Meaning
y in (*x*, ..., *z*)	*y* = *x* or ... or *y* = *z* (*x, ..., z*) represents a list of expressions, each of which evaluates to a single value. None of the expressions (*y, x,* or *z*) can be subqueries. The in predicate returns true if *y* is equal to one of the values in the list represented by (*x, ..., z*).
y not in (*x, ..., z*)	not (*y* in (*x*, ..., *z*)) (*x, ..., z)* represents a list of expressions, each of which evaluates to a single value. The not in predicate returns true if *y* is not equal to any value in the list represented by (*x, ..., z*). None of the expressions (*y, x,* or *z*) can be subqueries.
y in (*subquery*)	The subquery must be specified in parentheses and can refer to only one column in its select clause. The predicate returns true if y is equal to one of the values returned by the subquery.
y not in (*subquery*)	The subquery must be specified in parentheses and can refer to only one column in its select clause. The predicate returns true if *y* is not equal to any of the values returned by the subquery.

Any-or-All Predicate

An any-or-all predicate takes the form

any-or-all-operator **(***subquery***)**

The subquery must have exactly one element in the target list of its outermost subselect (so that it evaluates to a set of single values rather than a set of rows).

The any-or-all operator must be one of the following:

=any	=all
<>any	<>all
<any	<all
<=any	<=all
>any	>all
>=any	>=all

Let $ denote any one of the comparison operators =, <>, <, <=, >, >=. Then the predicate:

x $any (*subquery*)

evaluates to true if the comparison predicate:

x $ *y*

is true for at least one value *y* in the set of values represented by *subquery*. If the subquery is empty, the $any comparison fails (evaluates to false).

Likewise, the predicate:

x $all (*subquery*)

is true if the comparison predicate:

x $ *y*

is true for all values *y* in the set of values represented by *subquery*. If the subquery is empty, the $all comparison evaluates to true.

The operator =any is equivalent to the operator in. For example:

```
select ename
from employee
where dept = any
  (select dno
  from dept
  where floor = 3);
```

can be rewritten as:

```
select ename
from employee
where dept in
  (select dno
  from dept
  where floor = 3);
```

The operator some is a synonym for operator any. For example:

```
select ename
from employee
where dept = some
  (select dno from dept where floor = 3);
```

Exists Predicate

An exists predicate takes the form:

EXISTS (*subquery*)

An exists predicate expression evaluates to true if the set represented by subquery is non-empty. For example:

```
select ename
from   employee
where  exists
  (select *
  from   dept
  where  dno = employee.dept
  and  floor = 3);
```

It is typical, but not required, for the subquery argument to exists to be of the form select *.

Is Null Predicate

The is null predicate takes the form:

IS [NOT] NULL

For example:

```
x IS NULL
```

is true if *x* is a null. Because you cannot test for null using the "=" comparison operator, the null predicate must be used to determine whether an expression is null.

Search Conditions

Search conditions are used in where and having clauses to qualify the selection of data. Search conditions are composed of one or more predicates. Multiple predicates can be combined using parentheses and the logical operators (and, or, and not). The following examples illustrate possible combinations of search conditions:

Description	Example
Simple predicate	salary between 10000 and 20000
Predicate with not operator	edept not like 'eng_%'
Predicates combined using or operator	edept like 'eng_%' or edept like 'admin_%'
Predicates combined using and operator	salary between 10000 and 20000 and edept like 'eng_%'
Predicates combined using parentheses to specify evaluation	(salary between 10000 and 20000 and edept like 'eng_%') or edept like 'admin_%'

Predicates evaluate to true, false, or unknown. They evaluate to unknown if one or both operands are null (the is null predicate is the exception). When predicates are combined using logical operators (not, and, and or) to form a search condition, the search condition evaluates to true, false, or unknown as shown in the following tables:

and	**true**	**false**	**unknown**
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

or	**true**	**false**	**unknown**
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Not(true) is false, not(false) is true, not(unknown) is unknown.

After all search conditions are evaluated, the value of the where or having clause is determined. The where or having clause can be true or false only. Unknown values are considered false. For more information about predicates and logical operators, see Predicates (see page 92) and Logical Operators (see page 57) in this chapter.

Subqueries

Subqueries are select statements nested in other select statements. For example:

```
select ename
from employee
where dept in
  (select dno
  from dept
  where floor = 3);
```

Use subqueries in a where clause to qualify a specified column against a set of rows. In the previous example, the subquery returns the department numbers for departments on the third floor. The outer query then retrieves the names of employees who work in those departments.

Subqueries often take the place of expressions in predicates. Subqueries can be used in place of expressions only in the specific instances outlined in the descriptions of predicates earlier in this chapter. The select clause of a subquery must contain only one element.

A subquery can refer to correlation names defined (explicitly or implicitly) outside the subquery. For example:

```
select ename
from employee empx
where salary
  >  (select avg(salary)
  from employee empy
  where empy.dept = empx.dept);
```

The preceding subquery uses a correlation name (empx) defined in the outer query. The reference, empx.dept, must be explicitly qualified here, or it would be implicitly qualified by empy. The query is evaluated by assigning empx each of its values (that is, letting it range over the employee table), and evaluating the subquery for each value of empx. At least one of the correlation names must be specified in this example-either empx or empy, but not both, can be allowed to default to employee.

For more information about using correlation names in nested subqueries, see Correlation Names in the chapter “OpenSQL Data Types.”

Chapter 5: Embedded OpenSQL

This section contains the following topics:

Embedded OpenSQL

The term *embedded OpenSQL* refers to OpenSQL statements embedded in a host language such as C or Fortran. Embedded OpenSQL statements include most interactive OpenSQL statements, plus a number of statements that serve the specific needs of an embedded program. (In addition, forms statements can be used to develop forms-based applications. For details about forms statements, see the *Forms-based Application Development Tools User Guide.)*

How Embedded OpenSQL Programs Are Processed

Embedded OpenSQL programs must be processed by the embedded SQL preprocessor, which converts the statements into host language source code statements. The host language statements are primarily calls to a runtime library that provides the interface to the Enterprise Access product or server. (Non-SQL host language statements are not processed by the preprocessor.)

After the program has been preprocessed, compile and link it according to the requirements of the host language. For details about compiling and linking an embedded OpenSQL program, see the *Embedded SQL Companion Guide*.

The examples in this chapter use italics to indicate pseudocode that specifies the program statements that must be provided in the host language. All of the examples use the semicolon (;) as the statement terminator. However, in an actual program, the statement terminator is determined by the host language.

Syntax of an Embedded OpenSQL Statement

The syntax of an embedded OpenSQL statement is as follows:

```
[margin] EXEC SQL OpenSQL_statement [terminator]
```

When writing embedded OpenSQL statements, keep the following points in mind:

- The margin, consisting of spaces or tabs, is the margin that the host language compiler requires before the regular host code. Not all languages require margins. To determine if a margin is required, see the *Embedded SQL Companion Guide*.
- The keywords EXEC SQL must precede the OpenSQL statement; otherwise the statement is ignored. These words must appear together on a single line. They signal the preprocessor that the statement is an embedded OpenSQL statement.
- The statement terminator depends on the requirements of the host language. Different host languages require different terminators. Some host languages, such as Fortran, do not require a statement terminator.
- Embedded OpenSQL statements can be continued across multiple lines, according to the host language's rules for line continuation.
- Labels can precede the embedded statement if a host language statement in the same place can be preceded by a label. The label must be at the correct margin for labels and no syntactic element (including comments) can appear between it and the EXEC keyword.
- Host language comments must follow the rules for the host language.
- Some host languages allow you to place a line number in the margin.

For information about language-dependent syntax, see the *Embedded SQL Companion Guide*.

Structure of Embedded OpenSQL Programs

In general, OpenSQL statements can be embedded anywhere in a program that host language statements are allowed. The following example shows a simple embedded OpenSQL program that retrieves an employee's name and salary from the database and prints them on a standard output device. The statements that begin with the words EXEC SQL are embedded OpenSQL statements.

```
begin program

exec sql include sqlca;

exec sql begin declare section;
  name   character_string(15);
  salary float;
exec sql end declare section;

exec sql whenever sqlerror stop;

exec sql connect 'personnel/db2udb';

exec sql select ename, sal
into :name, :salary
from employee
where eno = 23;

print name, salary;

exec sql disconnect;

end program
```

The sequence of statements in the above example illustrates the typical structure of embedded OpenSQL programs. The first OpenSQL statement to appear is:

```
exec sql include sqlca;
```

This statement incorporates the OpenSQL error and status handling mechanism—the SQL Communications Area (SQLCA)—into the program. The SQLCA is required by the WHENEVER statement appearing later in the example.

Next is an OpenSQL declaration section. Host language variables to OpenSQL must be declared before using the variables in embedded OpenSQL statements.

The WHENEVER statement that follows uses information from the SQLCA to control program execution under error or exception conditions. An error handling mechanism should precede all executable embedded OpenSQL statements in a program. For details about error handling, see Error Handling in the chapter "OpenSQL Features."

Following the WHENEVER statement is a series of OpenSQL and host language statements. The first statement:

```
exec sql connect 'personnel/db2udb';
```

initiates access to the DB2 UDB personnel database through an Enterprise Access product. Your application must connect to a database before attempting to access the database. The slash (/) separates the database name from the server class. (The default server class is INGRES.) For details about server class, see your Enterprise Access product guide.

After connecting to the personnel database, the application issues the SELECT statement. The INTO clause specifies the host language variables into which the select statement retrieves values from the database. In the example, the variables are name and salary.

Following the SELECT statement is a host language statement that prints the values contained in the variables. Host language and embedded OpenSQL statements can be mixed in an application.

Finally, the application program disconnects from the database.

Host Language Variables

Embedded OpenSQL allows host language variables to be used for many elements of embedded OpenSQL statements. Host language variables can be used to transfer data from the database into the program and vice versa. Host language variables can also replace the search condition in a WHERE clause.

Host language variables can be used to specify:

- **Database expressions** - Variables can generally be used wherever expressions are allowed in embedded OpenSQL statements, such as in target lists and predicates. Variables must contain constant values and cannot represent names of database columns or include any operators.
- **Search conditions** - A WHERE clause can be specified in a variable. The entire WHERE clause must be contained in the variable. For example, to retrieve all columns for employees who earn more than the average salary:

```
wherevar = 'salary>(select avg(salary)
                    from employee)'
        exec sql select ename
        into :name
        from employee
        where :wherevar
```

- **Receiving variables** - A host variable can be used to specify the objects of the INTO clause of the SELECT and FETCH statements. The INTO clause is the means by which values retrieved from the database are transferred to host language variables.
- **Other statement arguments** - The statement descriptions in this guide note which arguments can be specified using host language variables.

A host language variable can be a single variable or a structure.

All host language variables must be declared to embedded OpenSQL before you can use them in embedded OpenSQL statements. The names of these variables cannot be keywords reserved by Ingres.

The following sections describe how to use host language variables. For language-specific details, see the *Embedded SQL Companion Guide*.

Variable Declarations

Host language variables must be declared to OpenSQL before using them in any embedded OpenSQL statements. Host language variables are declared to OpenSQL in a *declaration section* that has the following syntax:

```
EXEC SQL BEGIN DECLARE SECTION;
        host variable declarations
EXEC SQL END DECLARE SECTION;
```

A program can contain multiple declaration sections. The preprocessor treats variables that are declared in each declaration section as global to the embedded OpenSQL program from the point of declaration forward.

The variable declarations are identical to any variable declarations in the host language. The data types of the declared variables must belong to a subset of host language data types that are compatible with embedded OpenSQL data types. OpenSQL converts between host language data types and OpenSQL data types.

For a list of valid embedded OpenSQL data types and a discussion of data type conversion, see the *Embedded SQL Companion Guide*.

The embedded OpenSQL preprocessor is concerned only with host language variables that are declared to OpenSQL. Host language variables that are not declared to OpenSQL are invisible to the preprocessor and therefore can include data types that the preprocessor does not understand.

The Include Statement

The embedded OpenSQL INCLUDE statement lets you include external files in your source code. The syntax of the INCLUDE statement is as follows:

```
EXEC SQL INCLUDE filename;
```

This statement is commonly used to include an external file containing variable declarations. For example, assuming you have a file, myvars.dec, that contains a group of variable declarations, you can use the INCLUDE statement in the following manner:

```
exec sql begin declare section;
exec sql include 'myvars.dec';
exec sql end declare section;
```

This is the functional equivalent of listing all the declarations in the myvars.dec file in the declaration section itself.

For details about the INCLUDE statement, see Include in the chapter "OpenSQL Statements."

Variable Usage

After host language variables are declared, they can be used in embedded statements. Host language variables must be preceded by a colon. For example:

```
exec sql select ename, sal
into :name, :salary
from employee
where eno = :empnum;
```

The INTO clause contains two host language variables, name and salary and the WHERE clause contains one, empnum.

A host variable can have the same name as a database object, such as a column. The preceding colon distinguishes the variable from a database object of the same name.

If the application issues a query intended to retrieve values from a table into a host variable and the query returns no value (for example, no row in the table fulfilled the query), the contents of the variable are not modified.

Variable Structures

To simplify the transfer of data between database tables and embedded programs, variable structures can be used in the select, fetch, and insert statements. Variable structures are specified, like single variables, according to the rules of the host language and must be declared in an embedded OpenSQL declare section. The number, data type, and ordering of the structure's elements must correspond to the number, data type, and ordering of the result columns associated with a SELECT, FETCH, or INSERT statement.

For example, for a database table, employee, with the columns ename (data type character(20)) and eno (integer), declare the variable structure:

```
emprec
  ename character_string(20),
  eno integer;
```

and issue the SELECT statement

```
exec sql select *
 into :emprec.ename, :emprec.eno
 from employee
 where eno = 23;
```

Rather than specifying individual variables, you can specify the structure name in the SELECT statement. To specify the preceding example using a structure name, use the following SELECT statement:

```
exec sql select *
 into :emprec
 from employee
 where eno = 23;
```

The embedded OpenSQL preprocessor expands the structure name into the names of the individual members. Therefore, placing a structure name in the INTO clause is equivalent to enumerating all members of the structure in the order in which they were declared.

You can also use a structure to insert values in the database table. For example:

```
exec sql insert into employee (ename, eno)
 values (:emprec);
```

For details on the declaration and use of variable structures, see the *Embedded* SQL *Companion Guide*.

The Dclgen Utility—Generate Structure

The dclgen (Declaration Generator utility) is a structure-generating utility that maps the columns of a database table into a structure that can be included in a variable declaration. Invoke dclgen from the operating system level with the following command:

```
DCLGEN language dbname tablename filename structurename
```

where:

language

Specifies the host language (for example, "C").

dbname

Specifies the name of the database containing the table.

tablename

Specifies the name of the database table.

filename

Names the output file generated by dclgen that contains the structure declaration.

structurename

Specifies the name of the generated host language structure.

Dclgen creates the declaration file, *filename*, containing a structure corresponding to the database table. The file also includes a DECLARE TABLE statement that identifies the database table and columns from which the structure was generated. After the file has been generated, an embedded OpenSQL INCLUDE statement can be used to incorporate the file into the variable declaration section.

For details on the dclgen utility, see the *Embedded SQL Companion Guide*.

Indicator Variables

An *indicator variable* is a two-byte integer variable associated with a host language variable in an embedded OpenSQL statement. Indicator variables enable the application to:

- Detect when a null has been retrieved into a host variable. (When used to detect or assign a null, indicator variables are referred to as *null indicator variables*.)
- Assign a null to a table column.
- Detect character string truncation (when retrieving from a table into a host variable).

Indicator variable must be declared to embedded OpenSQL in a declare section.

In an embedded OpenSQL statement, the indicator variable is specified immediately after the host variable, with a colon separating the two:

host_variable:indicator_variable

The optional keyword indicator can be used in the syntax:

host_variable indicator:*indicator_variable*

Indicator variables can be associated with host language variables that contain the value of a database column or a constant database expression. For example, the following statement associates null indicators with variables that contain values retrieved from table columns:

```
exec sql select ename, esal
into :name:name_null, :salary:sal_null
from employee;
```

Null Indicators and Data Retrieval

When OpenSQL retrieves a null for a host variable that has an associated indicator variable, it sets the indicator variable to -1 and does not change the value of the host variable. If the value retrieved is not a null, then the indicator variable is set to 0 and the value is assigned to the host variable.

If the value retrieved is null and the program does not supply a null indicator, an error results.

Null indicator variables can be associated with the following:

- SELECT INTO and FETCH INTO result variables
- Data handlers for long varchar and long byte values

The following example illustrates the use of a null indicator when retrieving data from a database. This program retrieves employee information, then updates a roster. If a null phone number is detected (using the indicator, variable phone_null), the program places the string, N/A, in the roster's phone column.

```
exec sql fetch emp_cursor into :name,
        :phone:phone_null,  :id;
if (phone_null = -1) then
  update_roster(name, 'N/A', id);
else
  update_roster(name, phone, id);
end if;
```

Using Null Indicators to Assign Nulls

An indicator variable can be used with a host variable to assign a null value to a table column. When OpenSQL performs the assignment, it checks the value of the host variable's associated indicator variable. If the indicator variable's value is -1, then OpenSQL assigns a null to the column and ignores the value of the host variable. If the indicator variable does not contain -1, OpenSQL assigns the value of the host variable to the column. If the indicator value is -1 and the column is not nullable, then OpenSQL returns an error.

The following example demonstrates the use of an indicator variable and the null constant with the INSERT statement:

```
read name, phone number, and id from terminal;
if (phone = ' ') then
  phone_null = -1;
else
  phone_null = 0;
end if;
exec sql insert into newemp (name, phone, id,
        comment) values (:name, :phone:phone_null,
        :id, null);
```

This second example retrieves data from a form and updates the data in the database:

```
exec frs getform empform (:name:name_null = name,  :id:id_null = id);
exec sql update employee
set name = :name:name_null, id = :id:id_null
where current of emp_cursor;
```

Use null indicators to assign nulls in:

- The insert values list
- The update set list
- Constant expressions in select target lists used in embedded SELECT statements or subselect clauses

All constant expressions in the above list can include the keyword NULL. Specifying the word NULL is equivalent to specifying a null indicator with the value -1.

Indicator Variables and Character Data Retrieval

If OpenSQL retrieves a character string into a host variable that is too small to hold the string, the data is truncated to fit. (If the data was retrieved from the database, OpenSQL sets the sqlwarn1 field to "W".) If the host variable has an associated indicator variable, the indicator is set to the original length of the data. For example, the following statement sets the variable, char_ind, to 6 because it is attempting to retrieve a 6-character string into a 3-byte host variable, char_3:

```
exec sql select 'abcdef' into :char_3:char_ind;
```

Note: If a long varchar or long byte column is truncated into a host language variable, the indicator variable is set to 0. The maximum size of a long varchar or long byte column (2 GB) is too large to fit in an indicator variable.

Null Indicator Arrays and Host Structures

Use host structures to hold the data to be retrieved or written by SELECT, FETCH, and INSERT statements. In combination with host structures, an indicator array can be used to detect whether a particular member of the host structure contains a null.

An indicator array is an array of 2-byte integers that is associated with a host variable structure. Generally, indicator arrays are declared in the same declare section as their associated host variable structure. For example, the following code declares a host variable structure, emprec, and its associated indicator array, empind:

```
emprec
  ename    character(20),
  eid      integer,
  esal     float;
empind  array(3) of short_integer;
```

The preceding structure and indicator array might be used as follows:

```
exec sql select name, id, sal
into :emprec:empind
from employee
where number = 12;
```

A particular element of the indicator array is associated with the corresponding ordered member of the host structure: you do not need to specify each array element separately. The embedded OpenSQL preprocessor enumerates the elements in the array when expanding the structure into its members.

Data Manipulation with Cursors

Cursors enable embedded OpenSQL programs to process the result rows returned by a SELECT statement, one at a time. After a cursor has been opened, it can be advanced through the result rows. When the cursor is positioned to a row, the data in the row can be transferred to host language variables and processed according to the requirements of the application. The row to which the cursor is positioned is referred to as the current *row*.

A typical cursor application uses OpenSQL statements to perform the following steps:

1. Declare a cursor that will select a set of rows for processing.
2. Open the cursor, thereby selecting the data.
3. Fetch each row from the result table and move the data from the row into host language variables.
4. Optionally update or delete the current row.
5. Close the cursor and terminate processing.

An Example of Cursor Processing

This simple example of cursor processing prints the names and salaries of all the employees in the employee table and updates the salary of employees earning less than $10,000.

```
exec sql include sqlca;

exec sql begin declare section;
name     character_string(15);
salary   float;
exec sql end declare section;

exec sql whenever sqlerror stop;

exec sql connect personnel/rdb;

exec sql declare c1 cursor for
select ename, sal
from employee
for update of sal;

exec sql open c1;

exec sql whenever not found goto closec1;

loop while more rows

/* The WHENEVER NOT FOUND statement causes the loop
**   to be broken as soon as a row is not fetched.
*/

exec sql fetch c1 into :name, :salary;

print name, salary;

if salary < 10000 then
    exec sql update employee
      set salary = 10000
      where current of c1;

end if;
end loop;

closec1:

exec sql close c1;

exec sql disconnect;
```

Cursor Declaration

Before a cursor can be used in an application, it must be declared. The syntax for declaring a cursor is:

```
EXEC SQL DECLARE cursor_name CURSOR FOR
        select_statement;
```

The DECLARE CURSOR statement assigns a name to the cursor and associates the cursor with a SELECT statement to use to retrieve data. A cursor is always associated with a particular SELECT statement. The select is executed when the cursor is opened.

Updates can be performed only if the cursor's SELECT statement does not include any of the following elements:

- Aggregates
- Union clause
- Group by clause
- Having clause
- Distinct

These elements can be present in subselects within the SELECT statement, but must not occur in the outermost SELECT statement.

The *cursor_name* can be specified using a string literal or a host language string variable. Cursor names can be assigned dynamically. For details, see Example of Dynamically Specified Cursor Names (see page 122).

Opening a Cursor

Opening a cursor executes the associated SELECT statement and positions the cursor before the first row in the result table. To open a cursor, use the OPEN statement:

```
EXEC SQL OPEN cursor_name [FOR READONLY];
```

To specify that you intend to read the table without updating it, include the FOR READONLY clause. This clause may improve the performance of the cursor retrieval. If FOR READONLY is specified, updates cannot be performed on the data. FOR READONLY can be specified even if the cursor was declared for update.

Open Cursors and Transaction Processing

OpenSQL treats a multi-query transaction as a single statement (logically). Cursors cannot remain open across transactions. The commit statement closes all open cursors, even if a close cursor statement was not issued.

If an error occurs while a cursor is open, the Enterprise Access product or DBMS may roll back the entire transaction and close the cursor.

Fetch Statement—Fetch the Data

The FETCH statement advances the position of the cursor through the result rows returned by the select. Using the FETCH statement, your application can process the rows one at a time.

The syntax of the FETCH statement is:

```
EXEC SQL FETCH cursor_name
        INTO variable {, variable};
```

The FETCH statement advances the cursor to the first or next row in the result table and loads the values into host language variables.

To illustrate, the example of cursor processing shown previously contains the following DECLARE CURSOR statement:

```
exec sql declare c1 cursor for
select ename, sal
from employee
for update of sal;

open c1;
```

Later in the program, the following FETCH statement appears:

```
exec sql fetch c1 into :name, :salary;
```

This FETCH statement puts the values from the ename and sal columns of the current row into the host language variables name and salary.

Since the FETCH statement operates on a single row at a time, it is ordinarily placed inside a host language loop.

There are two ways to detect when the last row in the result table has been fetched:

- The sqlcode variable in the SQLCA is set to 100 if an attempt to fetch past the last row of the result table is made. (The SQL Communications Area (SQLCA) is a group of variables used by OpenSQL to provide error and status information to applications. After the last row is retrieved, succeeding fetches do not affect the contents of the host language variables specified in the INTO clause of the FETCH statement.
- The WHENEVER NOT FOUND statement specifies an action to be performed when the cursor moves past the last row. For details about the WHENEVER statement, see Trapping Errors Using the Whenever Statement in the chapter "OpenSQL Features."

Cursors can only move forward through a set of rows. To fetch a row again, a cursor must be closed and reopened.

Fetching Rows Inserted by Other Queries

While a cursor is open, the application can append rows using non-cursor insert statements. If rows are inserted after the current cursor position, the rows may or may not be visible to the cursor, depending on the following criteria:

- **Updatable cursors** - The newly inserted rows are visible to the cursor. Updatable cursors reference a single base table or updatable view.
- **Non-updatable cursors** - If the cursor SELECT statement retrieves rows directly from the base table, the newly inserted rows are visible to the cursor. If the SELECT statement manipulates the retrieved rows (for example, includes an ORDER BY clause), the cursor retrieves rows from an intermediate buffer, and cannot detect the newly inserted rows.

Using Cursors to Update Data

To use a cursor to update data, specify the FOR UPDATE clause in the cursor's declaration:

```
EXEC SQL DECLARE cursor_name CURSOR FOR
         select_statement
         FOR UPDATE OF column {, column};
```

The FOR UPDATE clause must list any columns in the selected database table that may require updating. Columns that have not been declared cannot be updated. If you are deleting rows, you do not need to declare the cursor for update.

The syntaxes for the CLOSE and FETCH statements are no different for cursors opened for update. However, the UPDATE statement has an extended version for cursors:

```
EXEC SQL UPDATE tablename
         SET column = expression {, column = expression}
         WHERE CURRENT OF cursor_name;
```

The WHERE clause of the cursor version specifies the row to which the cursor currently points, and the update affects only data in that row. Each column specified in the SET clause must have been declared for updating in the DECLARE CURSOR statement.

Be sure that the cursor is pointing to a row (a fetch has been executed) before performing a cursor update. The UPDATE statement does not advance the cursor. A fetch is still required to move the cursor forward one row. Two cursor updates not separated by a fetch will cause the same row to be updated twice or generate an error on the second update, depending on the underlying DBMS.

Using Cursors to Delete Data

The cursor version of the DELETE statement has the following syntax:

```
EXEC SQL DELETE FROM tablename
         WHERE CURRENT OF cursor_name;
```

The DELETE statement deletes the current row. The cursor must be positioned on a row (as the result of a FETCH statement) before a cursor delete can be performed. After the row is deleted, the cursor points to the position after the row (and before the next row) in the set. To advance the cursor to the next row, issue the FETCH statement.

You do not have to declare a cursor for update to perform a cursor delete.

Example of Updating and Deleting with Cursors

This example illustrates updating and deleting with a cursor:

```
exec sql include sqlca;

exec sql begin declare section;
name    character_string(15);
salary  float;
exec sql end declare section;

exec sql whenever sqlerror stop;

exec sql connect personnel/rdb;

exec sql declare c1 cursor for
select ename, sal
from employee
for update of sal;

exec sql open c1;

exec sql whenever not found goto closec1;

loop while more rows

exec sql fetch c1 into :name, :salary;
  print name, salary;

/* Increase salaries of all employees earning less
  than 60,000. */

if salary < 60,000 then

print 'Updating ', name;
  exec sql update employee
    set sal = sal * 1.1
    where current of c1;

/* Fire all employees earning more than 300,000. */

else if salary > 300,000 then

print 'Terminating ', name;
  exec sql delete from employee
    where current of c1;

end if;

end loop;

closec1:

exec sql close c1;

exec sql disconnect;
```

Closing Cursors

The final action in cursor processing is to close the cursor. Once the cursor is closed, no more processing can be performed with it unless another OPEN statement is issued.

The syntax for closing the cursor is as follows:

```
EXEC SQL CLOSE cursor_name;
```

The same cursor can be opened and closed any number of times in a single program, but it must be closed before reopening it. If a cursor is closed and reopened, the associated select statement is executed again and the cursor is positioned before the start of the result rows.

Summary of Cursor Positioning

The following table summarizes the effects of cursor statements on cursor positioning:

Statement	Effect on Cursor Position
open	Cursor positioned before first row in set.
fetch	Cursor moves to next row in set. If it is already on the last row, the cursor moves beyond the set and its position becomes undefined.
update(*cursor)*	Cursor remains on current row.
delete(*cursor)*	Cursor moves to a position after the deleted row (but before the following row).
close	Cursor and set of rows become undefined.

For extended examples of the use of cursors in embedded OpenSQL, see the *Embedded SQL Companion Guide*.

Example of Dynamically Specified Cursor Names

A dynamically specified cursor name (a cursor name specified using a host string variable) can be used to scan a table that contains rows that are related hierarchically, such as a table of employees and managers. In a relational database, this structure must be represented as a relationship between two columns. In an employee table, typically employees are assigned an ID number. One of the columns in the employee table contains the ID number of each employee's manager. This column establishes the relationships between employees and managers.

To use dynamically specified cursor names to scan this kind of table:

- Write a routine that uses a cursor to retrieve all the employees that work for a manager.
- Create a loop that calls this routine for each row that is retrieved and dynamically specifies the name of the cursor to be used by the routine.

The following example retrieves rows from the employee table that has the following format:

```
exec sql declare employee table
(ename   varchar(32),
 title   varchar(20),
 manager varchar(20));
```

This program scans the employee table and prints out all employees and the employees that they manage:

```
/* This program will print out, starting with
** the top manager,
** each manager and who they manage for the entire
** company. */

exec sql include sqlca;

/* main program */
exec sql begin declare section;
  topmanager character string(21)
exec sql end declare section;

exec sql connect enterprise/db2udb;

exec sql whenever not found goto closedb;
exec sql whenever sqlerror call sqlprint;

/* Retrieve top manager */
exec sql select ename into :topmanager from employeewhere title = 'President';

print 'President', topmanager
call printorg(1, topmanager);
/* start with top manager */
```

```
/* closedb */
closedb:
exec sql disconnect;

/* This subroutine retrieves and displays employees
** who report to a given manager. This subroutine is
** called recursively to determine if a given
** employee is also a manager and if so,
** it will display who reports to them.
*/

subroutine printorg(level, manager)
level integer

exec sql begin declare section;
  manager character string(21)
  ename character string(33)
  title character string(21);
  cname character string(4);
exec sql end declare section;

/* set cursor name to 'c1', 'c2', ... */
cname = 'c' + level

exec sql declare :cname cursor for
select ename, title, manager from employee
  where manager = :manager
  order by ename;

exec sql whenever not found goto closec;

exec sql open :cname;

loop
  exec sql fetch :cname into :ename, :title,
        :manager;

/* Print employee's name and title */
  print title, ename
/* Find out who (if anyone) reports to this employee*/
  printorg(level+1, ename);

end loop

closec:
exec sql close :cname;
return;
```

Data Handlers for Large Objects

To read and write long varchar and long byte columns (referred to as large objects), create routines called *data handlers*. Data handlers use GET DATA and PUT DATA statements to read and write segments of large object data. To invoke a data handler, specify the DATAHANDLER clause in an INSERT, UPDATE, FETCH, or SELECT statement. When the query is executed, the data handler routine is invoked to read or write the column.

In embedded SQL programs, use the DATAHANDLER clause in place of a variable or expression. For example, you can specify a data handler in a WHERE clause. The syntax of the DATAHANDLER clause is as follows:

```
datahandler(handler_routine([handler_arg]))[:indicator_var]
```

The following table lists the parameters for the DATAHANDLER clause:

Parameter	Description
handler_routine	Pointer to the data handler routine. Must be a valid pointer. An invalid pointer results in a runtime error.
handler_arg	Optional pointer to an argument to be passed to the data handler routine. The argument does not have to be declared in the declare section of the program.
indicator_var	Optional indicator variable (see page 110). For DATAHANDLER clauses in INSERT and UPDATE statements and WHERE clauses, if this variable is set to a negative value, the data handler routine is not called. If the data returned by a SELECT or FETCH statement is null, the indicator variable is set to -1 and the data handler routine is not called.

For example, the following SELECT statement returns the column, bookname, using the normal SQL method and the long varchar column, booktext, using a data handler:

```
exec sql select bookname, booktext into
     :booknamevar, datahandler(get_text())
     from booktable where bookauthor = 'Melville';
```

Separate data handler routines can be created to process different columns.

In select loops, data handlers are called once for each row returned.

Errors in Data Handlers

Errors from PUT DATA and GET DATA statements are raised immediately, and abort the SQL statement that invoked the data handler. If an error handler is in effect (as the result of a SET_SQL(ERRORHANDLER) statement), the error handling routine is called.

The data handler read routines (routines that issue get data statements) must issue the ENDDATA statement before exiting. If a data handler routine attempts to exit without issuing the ENDDATA statement, a runtime error is issued.

To determine the name of the column for which the data handler was invoked, use the INQUIRE_SQL(columnname) statement. To determine the data type of the column, use the INQUIRE_SQL(columntype) statement. The INIQUIRE_SQL(columntype) statement returns an integer code corresponding to the column data type (see page 139). These INQUIRE_SQL statements are valid only within a data handler routine. Outside of a data handler, these statements return empty strings.

Restrictions on Data Handlers

Data handlers are subject to the following restrictions:

- The DATAHANDLER clause is not valid in interactive SQL.
- The DATAHANDLER clause cannot be specified in a dynamic SQL statement.
- The DATAHANDLER clause cannot be specified in an execute procedure statement.
- The DATAHANDLER clause cannot be specified in a declare section.
- A data handler routine must not issue a database query. The following statements are valid in data handlers:
  - PUT DATA and GET DATA
  - ENDDATA (for read data handlers only)
  - INQUIRE_SQL and SET_SQL
  - Host language statements

Large Objects in Dynamic SQL

The following sections contain considerations and restrictions for using large object data in dynamic SQL programs.

Length Considerations

The sqllen field of the SQLDA is a 2-byte integer in which the DBMS Server returns the length of a column. If a long varchar or long byte column that is longer than the maximum value possible for sqllen (32,768) is described, a 0 is returned in sqllen.

Long varchar and long byte columns can contain a maximum of 2 GB of data. To prevent data truncation, be sure that the receiving variable to which the SQLDA sqldata field points is large enough to accommodate the data in the large object columns your program is reading. If data is truncated to fit in the receiving variable, the sqlwarn member of the sqlca structure is set to indicate truncation.

Data Handlers in Dynamic SQL

To specify a data handler routine to be called by a dynamic query that reads or writes a large object column, prepare the SQLDA fields for the large object column as follows:

- Set the sqltype field to IISQL_HDLR_TYPE. This value is defined when using the include sqlda statement to define an SQLDA structure in your program.
- Declare a sqlhdlr structure in your program. For details, see the *Embedded SQL Companion Guide*. Load the sqlhdlr field of this structure with a pointer to your data handler routine. If a variable is to be passed to the data handler, load the sqlarg field with a pointer to the variable. If no argument is to be passed, set the sqlarg field to 0.

If the value of the large object column is null (sqlind field of the SQLDA set to -1) the data handler is not invoked.

Example: PUT DATA Handler

The following example illustrates the use of the PUT DATA statement; the data handler routine writes a chapter from a text file to the book table. The data handler is called when the INSERT statement is executed on a table with the following structure.

```
exec sql create table book
     (chapter_name char(50),
      chapter_text long varchar);
For example:
exec sql begin declare section;
          char chapter_namebuf(50);
exec sql end declare section;

int put_handler();/* not necessary to
                    declare to embedded SQL */
...
copy chapter text into chapter_namebuf

exec sql insert into book
     (chapter_name, chapter_text)
      values (:chapter_namebuf,
      datahandler(put_handler()));
...

put_handler()

exec sql begin declare section;
          char         chap_segment[3000];
          int          chap_length;
          int          segment_length;
          int          error;

exec sql end declare section;

int               local_count = 0;

                ...
exec sql whenever sqlerror goto err;

chap_length = byte count of file

open file for reading

loop while (local_count < chap_length)

          read segment from file into chap_segment

          segment_length = number of bytes read

          exec sql put data
          (segment = :chap_segment,
           segmentlength = :segment_length)

     local_count = local_count + segment_length

end loop
```

```
exec sql put data (dataend = 1);  /* required by embedded SQL */

...

err:

exec sql inquire_sql(:error = errorno);

if (error <> 0)
     print error
     close file
```

Example: GET DATA Handler

The following example illustrates the use of the GET DATA statement in a data handler. This routine retrieves a chapter titled, "One Dark and Stormy Night," from the book table which has the following structure.

```
exec sql create table book
     (chapter_name char(50),
      chapter_text long varchar);
The data handler routine is called when the select statement is executed:
exec sql begin declare section;

          char     chapter_namebuf(50);

exec sql end declare section;

          int      get_handler()

...

Copy the string "One Dark and Stormy Night" into the chapter_namebuf variable.
exec sql select chapter_name, chapter_text
     into :chapter_namebuf, datahandler(get_handler())
     from book where chapter_name = :chapter_namebuf
     exec sql begin
          /* get_handler will be invoked
          once for each row */
     exec sql end;
...

get_handler()

exec sql begin declare section;
     char         chap_segment[1000];
     int          segment_length;
     int          data_end;
     int          error;
exec sql end declare section;

          ...

exec sql whenever sqlerror goto err;

data_end = 0

open file for writing

/* retrieve 1000 bytes at a time and write to text file. on last segment, less
than 1000 bytes may be  returned, so segment_length is used
 for actual number of bytes to write to file. */

while (data_end != 1)

     exec sql get data (:chap_segment = segment,
          :segment_length = segmentlength,
          :data_end = dataend)
          with maxlength = 1000;
```

```
     write segment_length number of bytes from
     "chap_segment" to text file

end while

...

err:

exec sql inquire_ingres(:error = errorno);

if (error != 0)

     print error
     close file
```

Example: Dynamic SQL Data Handler

The following example illustrates the use of data handlers in a dynamic SQL program. The sample table, big_table, was created with the following CREATE TABLE statement.

```
create table big_table
     (object_id integer, big_col long varchar);
```

The dynamic program retrieves data from big_table.

The data handler routine, userdatahandler, accepts a structure composed of a (long varchar) character string and an integer (which represents an object ID). The data handler writes the object ID followed by the text of the large object to a file.

The logic for the data handler is shown in the following pseudocode:

```
userdatahandler(info)

hdlr_param         pointer to info structure

{exec sql begin declare section;

          char         segbuf[1000];
          int          seglen;
          int          data_end;

exec sql end declare section;

data_end = 0

open file for writing

set arg_str field of info structure to filename
 /* to pass back to main program */

write arg_int field to file     /* id passed in
                                   from main program */

loop while (data_end != 1)
     exec sql get data
               (:segbuf = segment, :dataend = dFataend)
               with maxlength = 1000;

     write segment to file

end loop

close file

}
```

The structures required for using data handlers in dynamic SQL programs are declared in the eqsqlda.h source file, which is included in your program by an INCLUDE SQLDA statement. The following (C-style) definitions pertain to the use of data handlers:

```
# define   IISQ_LVCH_TYPE    22
# define   IISQ_HDLR_TYPE    46

typedef struct sqlhdlr_
{
           char     *sqlarg;
           int     (*sqlhdlr)();
} IISQLHDLR;
```

The following definitions must be provided by the application program. In this example the header file, mydecls.h, contains the required definitions.

```
/* Define structure hdlr_param, which will be used to pass information to and
receive information from the data handler. The data handler argument is a pointer
to a structure of this type, which is declared in the main program.*/

typedef  struct  hdlr_arg_struct

{
          char    arg_str[100];
          int     arg_int;

} hdlr_param;
```

The following code illustrates the main program, which uses dynamic SQL to read the long varchar data from the sample table. This sample program sets up the SQLDA to handle the retrieval of two columns, one integer column and one long varchar column. The long varchar column is processed using a user-defined data handler.

```
exec sql include 'mydecls.h';

main()

{

/*  declare the sqlda */

exec sql include sqlda;

declare host SQLDA: _sqlda

declare sqlda as pointer to host SQLDA _sqlda

exec sql begin declare section;

     character          stmt_buf[100];
     short integer      indicator1;
     short integer      indicator2;

exec sql end declare section;
```

```
integer          userdatahandler()

integer          i

/* Set the iisqhdlr structure; the data handler "userdatahandler" is invoked with
a pointer to "hdlr_arg" */

iisqlhdlr        data_handler;

/* Declare parameter to be passed to datahandler -- in this example a pointer to a
hdlr_param -- a struct with one character string field and one integer field as
defined in "mydecls.h". */

declare hdlr_param          hdlr_arg

set the SQLDA's sqln field to 2

copy "select object_id,big_col from big_table2" to the host language variable
stmt_buf

i = 0

exec sql connect 'mydatabase';


set the sqlhdlr field to point to the userdatahandler routine

set the sqlarg field to point to arguments (hdlr_arg)

/* Set the first sqlvar structure to retrieve column "object_id".Because this
column appears before the large object column in the target list, it IS retrieved
prior to the large object column, and can be put into the hdlr_arg that is passed
to the data handler. */

sqlvar[0].sqltype = IISQ_INT_TYPE

sqlvar[0].sqldata points to hdlr_arg.arg_int

sqlvar[0].sqlind points to indicator1

/* Set the second sqlvar structure to invoke a datahandler.the "sqltype" field
must be set to iisq_hdlr_type.the "sqldata" field must be pointer to iisqlhdlr
type. */

sqlvar[1].sqltype = IISQ_HDLR_TYPE

sqlvar[1].sqldata points to data_handler

sqlvar[1].sqlind points to indicator2

/* The data handler is called when the large object is retrieved. The data handler
writes the object_id and large object to a file and returns the file name to the
main program in the hdlr_arg struct. */

exec sql execute immediate :stmt_buf
     using descriptor sqlda;

exec sql begin;
```

```
/* process the file created in the data handler */

call processfile(hdlr_arg)

exec sql end;

}
```

Chapter 6: Dynamic OpenSQL

This section contains the following topics:

Note: Depending on your host language, some of the statements discussed in this chapter may vary in syntax or may not be supported. For information about dynamic programming that is specific to your host language, see the *Embedded SQL* Companion *Guide*.

Dynamic Programming

Dynamic programming enables embedded OpenSQL programs to specify a variety of program elements (such as queries and OpenSQL statements) at runtime. In applications where table names or column names are not known until runtime, or where complete queries must be built based on the application's runtime environment, the hard-coded OpenSQL statement is not sufficient. For example, an application might include an expert mode in which the runtime user can type in select queries and browse the results at the terminal. To support applications such as these, OpenSQL provides dynamic OpenSQL.

Dynamic OpenSQL provides the ability to specify table and column names and build queries at runtime. Using dynamic OpenSQL, you can:

- Execute a statement that is stored in a buffer (EXECUTE IMMEDIATE)
- Encode a statement stored in a buffer and execute it many times (PREPARE and EXECUTE)
- Obtain information about a table at runtime (PREPARE and DESCRIBE)

For details about the EXECUTE IMMEDIATE, PREPARE, EXECUTE and DESCRIBE statements, see Dynamic OpenSQL Statements (see page 140).

To support dynamic SELECT statements, the cursor statements (for example, DECLARE and OPEN) have dynamic versions. For details, see How to Execute a Dynamic Select Statement (see page 146) and Retrieve the Results Using a Cursor (see page 155).

The SQL Descriptor Area (SQLDA)

The OpenSQL Descriptor Area (SQLDA) is an integral part of dynamic programming. The SQLDA is a host language structure used by dynamic OpenSQL as a storage space for information. When used with the describe statement, this information includes the name, data type, and length of the result columns, the form's fields, or the table field's columns. When the SQLDA is used with other dynamic forms statements, the information includes the data type, length, and addresses of the variables that either store values from the table or form or contain values to be placed in the table or form.

Dynamic OpenSQL uses the SQLDA to store information about each result column of the SELECT statement. Dynamic Forms Runtime System (FRS) uses the SQLDA to hold descriptive information about the fields of a described form or columns of a described table field. Both dynamic OpenSQL and dynamic FRS use the SQLDA to store descriptive information about program variables. The SQLDA must be used when executing a DESCRIBE statement. The SQLDA can optionally be used when executing a FETCH, OPEN, PREPARE, EXECUTE, or EXECUTE IMMEDIATE statement.

Structure of the SQLDA

Storage for the SQLDA structure is typically allocated at runtime. If a program allows several dynamically defined cursors to be opened at one time, the program can allocate several SQLDA structures, one for each SELECT statement, and assign each structure a different name.

Each host language has different considerations for the SQLDA structure. Before writing a program that uses the SQLDA, see the *Embedded SQL Companion Guide* on dynamic OpenSQL statements.

The layout of the SQLDA is:

sqldabc

8-byte character array assigned a blank-padded value, "SQLDA."

sqldabc

4-byte integer assigned the size of the SQLDA.

sqln

2-byte integer indicating the number of allocated sqlvar elements. This value must be set by the program before describing a statement. The value must be greater than or equal to zero.

sqld

2-byte integer indicating the number of result columns associated with the DESCRIBE statement. This number specifies how many of the allocated sqlvar elements were used to describe the statement. If sqld is greater than sqln, then the program must reallocate the SQLDA to provide more storage buffers and reissue the DESCRIBE statement.

To use the SQLDA to place values in a table or form, the program must set sqld to the proper number before the SQLDA is used in a statement.

When describing a dynamic OpenSQL statement, if the value in sqld is zero, the described statement is not a SELECT statement.

sqlvar

An sqln-size array of:

sqltype

2-byte integer containing a code number indicating the data type of the column or variable. For a list of the codes and corresponding types, see Data Type Codes (see page 139).

sqllen

2-byte integer indicating the length of the column, variable, or field.

sqldata

Pointer to the variable described by the type and length.

sqlind

Pointer to indicator variable associated with the host variable.

sqlname

String containing the result column name (if a SELECT statement is being described). Maximum length is 32 bytes.

Including the SQLDA in a Program

To define the SQLDA, your application must issue the following INCLUDE statement:

```
EXEC SQL INCLUDE SQLDA;
```

Do not place this statement in a declaration section.

In most languages, this statement incorporates a set of type definitions that can be used to define the SQLDA structure. In some languages, it actually declares the structure. If the structure is declared directly (instead of using the INCLUDE statement), you can specify any name for the structure. For information about how your language handles this statement, see the *Embedded SQL Companion Guide*.

A program can have more than one SQLDA-type structure. A dynamic FRS DESCRIBE statement and a dynamic OpenSQL statement can use the same SQLDA structure if the described fields or table field columns have the same names, lengths, and data types as the columns of the database table specified in the dynamic OpenSQL statement.

Describe Statement and the SQLDA

Dynamic OpenSQL uses the DESCRIBE statement to return information about the result columns of a SELECT statement. Describing a select tells the program the data types, lengths, and names of the columns retrieved by the select. If you describe a statement other than SELECT, the only information returned is a 0 in the sqld field. For a complete discussion of how to use DESCRIBE in a dynamic OpenSQL application, see Preparing and Describing the Select Statement (see page 150).

Data Type Codes

The DESCRIBE statement returns a code indicating the data type of a field or column. This code is returned in sqltype, one of the fields in an sqlvar element.

The following table lists the data type codes. If the column, variable, or field described by the sqlvar element is nullable, the type code is returned as a negative value.

Data Type Name	Data Type Code	Nullable
integer	30	No
	-30	Yes
float	31	No
	-31	Yes
decimal	10	No
	-10	Yes
character	20	No
	-20	Yes
varchar	21	No
	-21	Yes
date	3	No
	-3	Yes
money	5	No
	-5	Yes

The Using Clause

The USING clause is an optional clause that provides certain OpenSQL statements with dynamic capabilities. The USING clause directs OpenSQL to use the variables pointed to by the sqlvar elements of the SQLDA (or other host variables) when executing the statement.

The USING clause has the following syntax:

```
USING DESCRIPTOR descriptor_name
```

Note: The keyword DESCRIPTOR is optional in some statements that accept the USING clause.

The following statements accept the USING clause:

- DESCRIBE
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE

For details about the EXECUTE statement, see How to Execute a Dynamic Non-select Statement (see page 143). For details about the EXECUTE IMMEDIATE statement, see How to Execute a Dynamic Select Statement (see page 146).

Dynamic OpenSQL Statements

This section describes the (non-cursor) dynamic OpenSQL statements. Dynamic OpenSQL has four statements that are exclusively used in a dynamic program:

- EXECUTE IMMEDIATE
- PREPARE
- EXECUTE
- DESCRIBE

In addition, all statements that support cursors (DECLARE, OPEN, FETCH, UPDATE, DELETE) have dynamic versions to support dynamically executed SELECT statements.

Execute Immediate Statement

The EXECUTE IMMEDIATE statement executes an OpenSQL statement specified as a string literal or using a host variable. The EXECUTE IMMEDIATE is most useful when the program intends to execute a statement only once, or when using a select loop with a dynamic SELECT statement.

The EXECUTE IMMEDIATE statement can be used to execute all OpenSQL statements except for the following statements:

CALL	FETCH
CLOSE	INCLUDE
CONNECT	INQUIRE_SQL
DECLARE	OPEN
DESCRIBE	PREPARE
DISCONNECT	SET_SQL
EXECUTE	WHENEVER

The EXECUTE IMMEDIATE statement has the following syntax:

```
EXEC SQL EXECUTE IMMEDIATE statement_string
       [INTO variable {, variable} | USING [DESCRIPTOR]
                descriptor_name
         [EXEC SQK BEGIN;
                program_code
          EXEC SQL END;]];
```

The contents of the *statement_string* must not include the keywords, EXEC SQL, or a statement terminator. The optional INTO/USING clause and BEGIN/END statement block can only be used when you are executing a dynamic SELECT statement.

Prepare and Execute Statements

The PREPARE statement tells OpenSQL to encode the dynamically built statement and assign it the specified name. After a statement is prepared, the program can execute the statement one or more times in a transaction by issuing the EXECUTE statement and specifying the statement name.

If your program executes the same statement many times in a transaction, the prepare and execute method can improve the performance of the statement. Committing a transaction discards any statements that were prepared during the transaction.

The following OpenSQL statements cannot be prepared:

CALL	EXECUTE
CLOSE	FETCH
CONNECT	INCLUDE
DECLARE	INQUIRE_SQL
DESCRIBE	OPEN
DISCONNECT	SET
EXECUTE IMMEDIATE	WHENEVER

The PREPARE statement has the following syntax:

```
EXEC SQL PREPARE statement_name
        [INTO descriptor_name|USING DESCRIPTOR descriptor_name]
        FROM host_string_variable | string_literal;
```

The *statement_name* can be a string literal or variable. The contents of the host string variable or the string literal cannot include EXEC SQL or the statement terminator.

If the INTO clause is included in the PREPARE statement, the PREPARE statement also describes the statement string into the specified descriptor area and it is not necessary to describe the statement string separately.

The syntax of the EXECUTE statement is as follows:

```
EXEC SQL EXECUTE statement_name
        [USING host_variable {, host_variable}
         | USING DESCRIPTOR descriptor_name];
```

A prepared statement can be fully specified, or some portions can be specified by question marks (?). The portions specified using question marks must be filled in by the USING clause when the statement is executed.

Describe Statement

The DESCRIBE statement describes a prepared OpenSQL statement into a program descriptor (SQLDA) to allow the program to interact with the dynamic statement as though it was hard coded in the program. This statement is used primarily with dynamic SELECT statements.

The DESCRIBE statement has the following syntax:

```
EXEC SQL DESCRIBE statement_name INTO|USING descriptor_name;
```

For more information about the DESCRIBE statement, see The SQL Descriptor Area (SQLDA (see page 136)), and Preparing and Describing the Select Statement (see page 150).

How to Execute a Dynamic Nonselect Statement

To execute a dynamic non-select statement, use either the EXECUTE IMMEDIATE statement or the PREPARE and EXECUTE statements. EXECUTE IMMEDIATE is most useful if the program executes the statement only once within a transaction. If the program executes the statement many times within a transaction, for example, within a program loop, use the PREPARE and EXECUTE combination: prepare the statement once, then execute as many times as necessary.

If the program does not know whether the statement is a SELECT statement, the program can prepare and describe the statement. The results returned by the DESCRIBE statement will indicate whether the statement was a select. For more information and a sample of the conditional coding to handle such situations, see Executing the Select with Execute Immediate (see page 154).

Preparing and Executing a Non-select Statement

The PREPARE and EXECUTE statements can also be used to execute dynamic non-select statements. These two statements, working together, allow your program to save a statement string and execute it as many times as necessary. However, a prepared statement is discarded when the transaction in which it was prepared is rolled back or committed. Also, if a statement is prepared with the same name as an existing statement, the new statement supersedes the old statement.

The following example demonstrates how a runtime user can prepare (save) a dynamically specified OpenSQL statement and execute it a specific number of times:

```
read OpenSQL statement from terminal into buffer;
exec sql prepare s1 from :buffer;
read number in N
loop N times
  exec sql execute s1;
end loop;
```

The next example illustrates a dynamically prepared query. This example creates a table whose name is the same as the user's name, and inserts into the table a set of rows with fixed-typed parameters (the user's children):

```
get user name from terminal;
buffer = 'create table ' + user_name + '(child
character(15), age integer)';
exec sql execute immediate :buffer;

buffer = 'insert into ' + user_name + '(child, age)
values (?, ?)';
exec sql prepare s1 from :buffer;

read child's name and age from terminal;
loop until no more children
exec sql execute s1 using :child, :age;

read child's name and age from terminal;
end loop;
```

Some statements cannot be executed using PREPARE and EXECUTE, as described in Dynamic OpenSQL Statements (see page 140).

Executing a Non-select Statement Using Execute Immediate

EXECUTE IMMEDIATE executes an OpenSQL statement specified using a string literal or host variable. The EXECUTE IMMEDIATE statement can be used to execute all but a few of the OpenSQL statements. For a list of statements that you cannot execute with the EXECUTE IMMEDIATE statement, see Execute Immediate Statement (see page 141).

When the EXECUTE IMMEDIATE statement is used to execute a statement that is not a select, its syntax is as follows:

```
EXEC SQL EXECUTE IMMEDIATE statement_string;
```

For example, the following statement executes a DROP statement specified as a string literal:

```
/*
** Statement specification included
** in string literal. The string literal does
** NOT include 'exec sql' or ';'
*/
exec sql execute immediate 'drop employee';
```

As another example, the following code reads OpenSQL statements from a file into a host string variable named buffer, and executes the contents of the variable. If the variable includes a statement that cannot be executed by EXECUTE IMMEDIATE, or if another error occurs, the loop is broken.

```
exec sql begin declare section;
    character buffer(100);
exec sql end declare section;
 open file;
loop while not end of file and not error

read statement from file into buffer;
exec sql execute immediate :buffer;

end loop;
close file;
```

If only a statement's parameters, such as an employee name or number, change at runtime, then you do not need to use EXECUTE IMMEDIATE; you can replace a value with a host variable. For example, the following fragment increases the salaries of all employees with a specific employee number (read out of a file into variable, number):

```
loop while not end of file and not error

read number from file;
exec sql update employee
  set sal = sal * 1.1
  where eno = :number;

end loop;
```

How to Execute a Dynamic Select Statement

If you know the data types of the result columns, use the EXECUTE IMMEDIATE statement with the INTO clause. For details, see When the Result Column Data Types are Known (see page 148).

If you do not know the data types of the result columns, the SELECT statement must be prepared and described first, then the program can either:

- Use the EXECUTE IMMEDIATE statement with the USING clause to execute the select.
- Declare a cursor for the prepared SELECT statement and use the cursor to retrieve the results.

For more information, see When the Result Column Data Types are Unknown (see page 149).

The EXECUTE IMMEDIATE option defines a select loop to process the results of the select. Select loops minimize disk and communications I/O but do not allow the program to issue any other OpenSQL statements in the loop. If the program must access the database while processing rows, use the cursor option.

If the program does not know whether the statement is a select, the PREPARE and DESCRIBE statements can be used to determine whether the statement is a select. The following example demonstrates the program logic required to accept OpenSQL statements from a user, execute the statements, and print the results. If the statement is a select, the program uses a cursor to execute the query.

```
statement_buffer = ' ';
loop while reading statement_buffer from terminal

exec sql prepare s1 from :statement_buffer;
exec sql describe s1 into :result_descriptor;

if (sqlda.sqld = 0) then

exec sql execute s1;

else

/* This is a SELECT */
  exec sql declare c1 cursor for s1;
  exec sql open c1;

allocate result variables using result_descriptor;

loop while there are more rows in the cursor

exec sql fetch c1 using descriptor
                :result_descriptor;
    if (sqlca.sqlcode not equal 100) then
      print the row using result_descriptor;
    end if;

end loop;

free result variables from result_descriptor;

exec sql close c1;

end if;

process sqlca for status;

end loop;
```

When the Result Column Data Types Are Known

If the program knows the data types of the resulting columns and of the result variables used to store the column values, the program can execute the SELECT statement using the EXECUTE IMMEDIATE statement with the INTO clause.

In the following example, a database contains several password tables, each having one column and one row and containing a password value. An application connected to this database requires a user to correctly enter two passwords before continuing. The first password entered is actually the name of a password table and the second is the password value in that table.

The following code uses the EXECUTE IMMEDIATE statement to execute the dynamically defined select built by the application to check these passwords:

```
...
exec frs prompt noecho ('First Password: ',
 :table_password);
exec frs prompt noecho ('Second Password: ',
 :value_password);

select_stmt = 'select column1 from ' +
        table_password;
exec sql execute immediate :select_stmt
  into :result_password;
if (sqlcode < 0) or (value_password <>
        result_password)
then
  exec frs message 'Password authorization failure';
endif
...
```

Because the application's developer knows the data type of the column in the password table (although not which password table will be selected), the developer can execute the dynamic select with the EXECUTE IMMEDIATE statement and the INTO clause.

The syntax of EXECUTE IMMEDIATE in this context is shown here:

```
EXEC SQL EXECUTE IMMEDIATE select_statement
        INTO variable {, variable};
[EXEC SQL BEGIN;
        host_code
 EXEC SQL END;]
```

This syntax retrieves the results of the select into the specified host variables. The BEGIN and END statements define a select loop that processes each row returned by the SELECT statement and terminates when there are no more rows to process. If a select loop is used, your program cannot issue any other OpenSQL statements for the duration of the loop.

If the select loop is not included in the statement, OpenSQL assumes that the SELECT statement is a singleton select returning only one row and, if more than one row is returned, issues an error.

When the Result Column Data Types Are Unknown

In most instances, when executing a dynamically defined SELECT statement, the program does not know the number or types of result columns. To provide this information to the program, first prepare and then describe the SELECT statement. The DESCRIBE statement returns to the program the type description of the result columns of a prepared SELECT statement. After the select is described, the program must dynamically allocate (or reference) the correct number of result storage areas of the correct size and type to receive the results of the select.

If the statement is not a SELECT statement, describe returns a zero to the sqld and no sqlvar elements are used.

After the statement has been prepared and described and the result variables allocated, the program has two choices regarding the execution of the SELECT statement:

- The program can associate the statement name with a cursor name, open the cursor, fetch the results into the allocated result storage area (one row at a time), and close the cursor.
- The program can use EXECUTE IMMEDIATE, which allows you to define a select loop to process the returned rows. If the select will return only one row, then it is not necessary to define the select loop.

Preparing and Describing the Select Statement

If the program has no advance knowledge of the resulting columns, the first step in executing a dynamic SELECT statement is to prepare and describe the statement. Preparing the statement encodes and saves the statement and assigns it a name. For information about the syntax and use of PREPARE, see Prepare and Execute Statements (see page 142).

The DESCRIBE statement returns descriptive information about a prepared statement into a program descriptor, that is, an SQLDA structure. This statement is primarily used to return information about the result columns of a SELECT statement to the program, but other statements can be described. When describing a non-select statement, the only information returned to the program is that the statement was not a SELECT statement.

The DESCRIBE statement has the following syntax:

EXEC SQL DESCRIBE *statement_name* INTO|USING *descriptor_name*;

When a SELECT statement is described, OpenSQL returns the information about each result column to a sqlvar element (described in Structure of the SQLDA (see page 137)). This is a one-to-one correspondence: the information in one sqlvar element corresponds to one result column. Before issuing the DESCRIBE statement, the program must allocate sufficient sqlvar elements and set the SQLDA sqln field to the number of allocated sqlvars. The program must set sqln before the DESCRIBE statement is issued.

After issuing the DESCRIBE statement, the program must check the value of sqld, which contains the number of sqlvar elements actually used to describe the statement. If sqld is zero, the prepared statement was not a SELECT statement. If sqld is greater than sqln, the SQLDA does not have enough sqlvar elements: more storage must be allocated and the statement must be redescribed.

The following fragment shows a typical DESCRIBE statement and the surrounding host program code. The program assumes that 20 sqlvar elements will be sufficient:

```
sqlda.sqln = 20;
exec sql describe s1 into sqlda;

if (sqlda.sqld = 0) then
  statement is not a select statement;

else if (sqlda.sqld > sqlda.sqln) then

  save sqld;
  free current sqlda;
  allocate new sqlda using sqld as the size;
  sqlda.sqln = sqld;
  exec sql describe s1 into sqlda;

end if;
```

Analyzing the Sqlvar Elements

After describing a statement, the program must analyze the contents of the sqlvar array. Each element of the sqlvar array describes one result column of the SELECT statement. Together, all the sqlvar elements describe one complete row of the result table.

The DESCRIBE statement sets the data type, length, and name of the result column (sqltype, sqllen and sqlname), and the program must use that information to supply the address of the result variable and result indicator variable (sqldata and sqlind).

For example, assuming the table, object, was created as follows:

```
exec sq  create table object
(o_id    integer not null,
 o_desc  character(100) not null,
 o_price float not null,
 o_sold  date);
```

and the following dynamic query was described as follows:

```
exec sql prepare s1 from 'select * from object';
exec sql describe s1 into sqlda;
```

The SQLDA descriptor results would be:

sqld	4 (columns)
sqlvar(1)	sqltype = 30 (integer)
	sqllen = 4
	sqlname = 'o_id'
sqlvar(2)	sqltype = 20 (character)

	sqllen = 100
	sqlname = 'o_desc'
sqlvar(3)	sqltype = 31 (float)
	sqllen = 8
	sqlname = 'o_price'
sqlvar(4)	sqltype = -3 (date)
	sqllen = 0
	sqlname = 'o_sold'

The DESCRIBE statement sets the value of sqllen to the length of the result column. For character data types, sqllen is set to the maximum length of the character string. For numeric data types, sqllen is set to the size of the numeric field as declared when created. For the date data type, sqllen is set to 0, but the program should use a 25-byte character string to retrieve or set date data. Note that, for nullable columns, a negative value is returned.

After the statement is described, your program must analyze the values of sqltype and sqllen in each sqlvar element. If sqltype and sqllen do not correspond exactly with the types of variables used by the program to process the SELECT statement, then sqltype and sqllen must be modified to be consistent with the program variables. After describing a SELECT statement, there will be one sqlvar element for each expression in the select target list.

After processing the values of sqltype and sqllen, allocate storage for the variables that will contain the values in the result columns of the SELECT statement, by pointing sqldata at a host language variable that will contain the result data. If the value of sqltype is negative, which indicates a nullable result column data type, allocate an indicator variable for the particular result column and set sqlind to point to the indicator variable. If sqltype is positive, indicating that the result column data type is not nullable, an indicator variable is not required. In this case, set sqlind to zero.

To omit the null indicator for a nullable result column (sqltype is negative), set sqltype to its positive value and sqlind to zero. If sqltype is positive and an indicator variable is allocated, set sqltype to its negative value, and set sqlind to point to the indicator variable.

In the previous example, after the program analyzes the results as described, the date type is changed to character and sqlind and sqldata are set to appropriate values. The values in the resulting sqlvar elements are:

```
sqlvar(1)    sqltype   =   30 (integer),
             sqllen    =   4,
             sqldata   =   Address of 4-byte integer,
             sqlind    =   0,
             sqlname   =   'o_id'
sqlvar(2)    sqltype   =   20 (character),
             sqllen    =   100,
             sqldata   =   Address of 100-byte character string,
             sqlind    =   0,
             sqlname   =   'o_desc'
sqlvar(3)    sqltype   =   31 (float),
             sqllen    =   8
             sqldata   =   Address of 8-byte floating point,
             sqlind    =   0,
             sqlname   =   'o_price'
sqlvar(4)    sqltype   =   -30 (Nullable character, was date),
             sqllen    =   25, (was 0)
             sqldata   =   Address of 25-byte character string,
             sqlind    =   Address of 2-byte indicator variable,
             sqlname   =   'o_sold'
```

Executing the Select with Execute Immediate

You can execute a dynamic SELECT statement that has been prepared and described with an EXECUTE IMMEDIATE statement that includes the USING clause. The USING clause tells OpenSQL to place the values returned by the select into the variables pointed to by the elements of the SQLDA sqlvar array. If the select will return more than one row, you can also define a select loop to process each row before another is returned.

The syntax of EXECUTE IMMEDIATE in this context is as follows:

```
EXEC SQL EXECUTE IMMEDIATE select_statement
 USING [DESCRIPTOR] descriptor_name;
[EXEC SQL BEGIN;
 host_code
 EXEC SQL END;]
```

Within a select loop, no OpenSQL statements other than an ENDSELECT can be issued. For selects without select loops, OpenSQL issues an error if more than one row is returned.

To illustrate this option, the following example contains a dynamic SELECT. The results of the SELECT statement are used to generate a report.

```
...
allocate an sqlda
read the dynamic select from the terminal into a
stmt_buffer

exec sql prepare s1 from :stmt_buffer;
exec sql describe s1 into :sqlda;
if (sqlca.sqlcode < 0) or (sqlda.sqld = 0) then
  print ('Error or statement is not a select');
  return;
else if (sqlda.sqld > sqlda.sqln) then
 allocate a new sqlda;
  exec sql describe s1 into :sqlda;
endif;

analyze the results and allocate variables

exec sql execute immediate :stmt_buffer
 using descriptor :sqlda;
exec sql begin;
process results, generating report
if error occurs, then
  exec sql endselect;
endif
exec sql end;
...
```

Retrieve the Results Using a Cursor

To give your program the ability to access the database or issue other database statements while processing rows retrieved as the result of the select, use a cursor to retrieve those rows.

To use cursors, after the SQLDA has been analyzed and result variables have been allocated and pointed at, the program must declare and open a cursor in order to fetch the result rows.

The syntax of the cursor declaration for a dynamically defined SELECT statement is as follows:

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
```

This statement associates the SELECT statement represented by *statement_name* with the specified cursor. *Statement_name* is the name assigned to the statement when the statement was prepared. As with non-dynamic cursor declarations, the SELECT statement is not evaluated until the cursor is actually opened. After opening the cursor, the program retrieves the result rows using the FETCH statement with the SQLDA instead of the list of output variables.

The syntax for a cursor fetch statement is as follows:

```
EXEC SQL FETCH cursor_name USING DESCRIPTOR descriptor_name;
```

Before the FETCH statement, the program has filled the result descriptor with the addresses of the result storage areas. When executing the FETCH statement, OpenSQL copies the result columns into the result areas referenced by the descriptor.

The following program fragment elaborates on an earlier example in this section. The program reads a statement from the terminal. If the statement is quit, the program ends. Otherwise, the program prepares the statement. If the statement is not a SELECT, then it is executed. If the statement is a SELECT statement, then it is described, a cursor is opened, and the result rows are fetched. Error handling is not shown.

```
exec sql include sqlca;
exec sql include sqlda;

allocate an sqlda with 400 sqlvar elements;
sqlda.sqln = 400;

read statement_buffer from terminal;

loop while (statement_buffer <> 'quit')

exec sql prepare s1 from :statement_buffer;
exec sql describe s1 into sqlda;

if (sqlda.sqld = 0) then  /* This is not a select */
  exec sql execute s1;
 else    /* This is a select */
  exec sql declare c1 cursor for s1;
  exec sql open c1;

  print column headers from the sqlname fields;
  analyze the SQLDA, inspecting types and lengths;
  allocate result variables for a cursor result row;
  set sqlvar fields sqldata and sqlind;

  loop while (sqlca.sqlcode = 0)
   exec sql fetch c1 using descriptor sqlda;
   if (sqlca.sqlcode = 0) then
      print the row using sqldata and sqlind
      pointed at by the sqlvar array;
   end if;
  end loop;

  free result variables from the sqlvar elements;

  exec sql close c1;

end if;

process sqlca and print the status;
 read statement_buffer from terminal;

end loop;
```

Chapter 7: OpenSQL Features

This section contains the following topics:

Transactions

A *transaction* is one or more OpenSQL statements processed as a single, indivisible database action. A transaction that consists of a single OpenSQL statement is sometimes called a single query transaction (SQT). If the transaction contains multiple statements, it is often called a multiple query transaction (MQT). By default, all transactions are multiple query transactions.

How Transactions Work

The transaction begins with the first OpenSQL statement following a CONNECT, COMMIT, or ROLLBACK statement, which can be issued by you, the program, or in some instances, by the DBMS.

The transaction continues until there is an explicit COMMIT or ROLLBACK statement or until the session terminates. (Terminating the session or disconnecting from the database normally issues an implicit COMMIT statement. If the session or connection terminates abnormally, the results depend on the host DBMS.)

In Enterprise Access products, transactions are managed by the underlying DBMS. Transaction handling may vary depending on the DBMS to which your session is connected.

For example, some DBMSs begin a transaction immediately following the CONNECT or ROLLBACK statements, rather than awaiting the next OpenSQL statement. For details, see the documentation for the host DBMS.

How Consistency Is Maintained During Transactions

None of the database changes made by a transaction are visible to other sessions until the transaction is committed. In a multi-user environment, where many transactions may be executing simultaneously, this behavior maintains database consistency. For example, if two transactions are updating the same information in a table, the DBMS must ensure that one transaction's updates are complete before allowing the other to proceed.

How Transactions Are Controlled

The COMMIT and ROLLBACK statements allow control of the effects of a transaction on the database:

- The COMMIT statement makes the changes permanent.
- The ROLLBACK statement undoes the changes made by the transaction.

When a COMMIT statement is issued, the DBMS makes all changes resulting from the transaction permanent, terminates the transaction, and drops any locks held during the transaction. When a ROLLBACK statement is issued, the DBMS undoes any database changes made by the transaction, terminates the transaction, and releases any locks held during the transaction.

How Transactions Are Committed

Transactions are composed of one or more OpenSQL statements. In general, a transaction begins with the first statement after connection to the database or the first statement following a commit or rollback. The precise starting point of a transaction depends on the DBMS to which you are connected. Subsequent statements are part of the transaction until a commit or rollback is executed. By default, an explicit commit or rollback must be issued to close a transaction.

To direct the DBMS to commit each database statement individually, use the SET AUTOCOMMIT ON statement. (This statement cannot be issued in an open transaction.) When autocommit is set on, a commit occurs automatically after every statement, except PREPARE and DESCRIBE.

If autocommit is on and a cursor is opened, the server or Enterprise Access product does not issue a commit until the CLOSE CURSOR statement is executed, because cursors are logically a single statement. A ROLLBACK statement can be issued when a cursor is open. To restore the default behavior (and enable multiquery transactions), issue the SET AUTOCOMMIT OFF statement.

To determine whether you are in a transaction, use the INQUIRE_SQL statement. For information about INQUIRE_SQL, see Status Information (see page 160) and Inquire_sql in the "OpenSQL Statements" chapter. To find out if autocommit is on or off, use dbmsinfo. For information about dbmsinfo, see The Dbmsinfo Function (see page 161).

Abort Policy for Statements and Transactions

Transactions and statements can be aborted by an application or by the DBMS. Applications can abort transactions or statements as a result of:

- ROLLBACK statement
- Timeout (if available and set)

The DBMS aborts statements and transactions as a result of these conditions:

- Deadlock
- Error while executing a database statement

Effects of Aborting Transactions

When a statement or transaction is aborted:

- Rolling back a single statement does not cause the DBMS to release any locks held by the transaction. Locks are released when the transaction ends.
- If cursors are open, the entire transaction is always aborted.
- When an entire transaction is aborted, all open cursors are closed, and all prepared statements are invalidated.

Interrupting Transactions

The effect of a keyboard interrupt (Ctrl+C or Ctrl+Y) on a transaction depends on the Enterprise Access product and underlying DBMS. For details, see the DBMS documentation.

Status Information

The following features enable your application program to obtain status information:

- Dbmsinfo - Returns information about the current session
- Inquire_sql - Returns information about the last database statement that was executed
- Inquire_frs - Returns information about the forms system
- SQLCA (SQL Communications Area) - Returns status and error information about the last OpenSQL statement that was executed

The Dbmsinfo Function—Retrieve Information on Current Session

Dbmsinfo is a function that returns a string containing information about the current session. This statement can be used in the Terminal Monitor or in an embedded OpenSQL application.

The dbmsinfo statement has the following syntax:

```
SELECT DBMSINFO ('request_name') [AS result_column]
```

For example, to determine which version of the Enterprise Access product or server you are using, enter:

```
select dbmsinfo('_version');
```

In OpenSQL, only one dbmsinfo request is allowed per SELECT statement. In addition, when issuing a SELECT DBMSINFO statement, you cannot specify other SELECT statement clauses (such as FROM or WHERE).

The following table lists valid *request_names*:

Request Name	Response Description
autocommit_state	Returns 1 if autocommit is on; 0 if autocommit is off.
_bintim	Returns the current time and date in an internal format, represented as the number of seconds since January 1, 1970 00:00:00 GMT.
database	Returns the name of the database to which the session is connected.
dba	Returns the DBMS username of the database owner.
_et_sec	Returns the elapsed time for session, in seconds.
query_language	Returns query language in use (“SQL”).
server_class	Returns the class of DBMS server, for example “db2”.
terminal	Returns the terminal address for local connections.
transaction_state	Returns 1 if presently in a transaction, 0 if not.
unicode_normalization	Returns blank if the database does not support Unicode or does not perform normalization. Returns NFC if the database supports the NFC normalization form. Returns NFD if the database supports the NFD normalization form.

Request Name	Response Description
username	Returns the DBMS user name of the current session's user (like user).
_version	Returns the DBMS version number.

The Inquire_sql Statement—Retrieve Runtime Information

The INQUIRE_SQL statement returns information about the results of the last OpenSQL database statement issued by a session. Using INQUIRE_SQL you can obtain a variety of information, including:

- Error number and text (if the last statement resulted in an error)
- Type of error being returned (described in detail in Local and Generic Errors (see page 166))
- Whether a transaction is open
- Session identifier (in multiple-session applications)

Note: The INQUIRE_SQL statement does not return status information about forms statements. Use the INQUIRE_FRS statement, described in the *Forms-based Application Development Tools User Guid*e, to obtain information about forms statements.

The SQL Communications Area (SQLCA)

The SQL Communications Area (SQLCA) consists of a number of variables that contain error and status information accessible by the program. This information reflects only the status of executed embedded OpenSQL database statements. Forms statements do not affect these variables. Because each embedded OpenSQL statement has the potential to change values in the SQLCA, the application must perform any checking and consequent processing required to handle a status condition immediately after the statement in question. Otherwise, the next executed OpenSQL statement might change the status information in the variables.

Each host language implements the SQLCA structure differently. For instructions on how to include the SQLCA in your applications, see the *Embedded SQL* Companion *Guide*.

The following list describes the variables that compose the SQLCA (not all of the variables are currently used):

sqlcaid — An 8-byte character string variable initialized to "SQLCA." This value does not change.

sqlcaid — An 8-byte character string variable initialized to "SQLCA." This value does not change.

sqlcabc — A 4-byte integer variable initialized to the length in bytes of the SQLCA, 136. This value also does not change.

sqlcode — A 4-byte integer variable indicating the OpenSQL return code. Its value falls into one of three categories:

= 0	The statement executed successfully (though there may have been warning messages - see sqlwarn0).
< 0	An error occurred. The value of sqlcode is the negative value of the error number returned to errorno. (For a discussion of errorno, see Error Handling (see page 164).) A negative value sets the sqlerror condition of the WHENEVER statement.
>0	The statement executed successfully but an exceptional condition occurred. The value +100 indicates that no rows were processed by a DELETE, FETCH, INSERT, SELECT, UPDATE, MODIFY, COPY, CREATE INDEX, or CREATE...AS SELECT statement. This value (+100) sets the not found condition of the WHENEVER statement.

sqlerrm — A varying-length character string variable composed of an initial 2-byte count and a 70-byte long buffer. This variable is used for error messages. When an error occurs for a database statement, the leading 70 characters of the error message are assigned to this variable. If the message contained within the variable is less than 70 characters, the variable contains the complete error message. Otherwise, the variable contains a truncated error message.

To retrieve the full error message, use the INQUIRE_SQL statement with the errortext object. If no errors occur, sqlerrm will contain blanks. For some languages, this variable is divided into two other variables: sqlerrml, a 2-byte integer count indicating how many characters are in the buffer, and sqlerrmc, a 70-byte fixed-length character string buffer.

sqlerrp — An 8-byte character string variable, currently unused.

sqlerrd — An array of six 4-byte integers. Currently only sqlerrd(1) and sqlerrd(3) are in use. sqlerrd(1) is used to store error numbers returned by the server. For more information about sqlerrd(1), see Local and Generic Errors (see page 166).

sqlerrd (3) indicates the number of rows processed by a DELETE, FETCH, INSERT, SELECT, UPDATE, MODIFY, COPY, CREATE INDEX, or CREATE...AS SELECT statement. All other database statements reset this variable to zero. Some host languages start array subscripts at 0. In these languages (C, BASIC), use

sqlcaid — An 8-byte character string variable initialized to "SQLCA." This value does not change.

the subscript 2 to select the third array variable.

qlwarn0 through sqlwarn 7 — A set of eight 1-byte character variables that denote warnings when set to "W." The default values are blanks.

- sqlwarn0 — If set to "W," at least one other sqlwarn contains a "W." When "W" is set, the sqlwarning condition of the WHENEVER statement is set.
- sqlwarn1 — Set to "W" on truncation of a character string assignment from the database into a host variable. If an indicator variable is associated with the host variable, the indicator variable is set to the original length of the character string.
- sqlwarn2 — Set to "W" on elimination of nulls from aggregates.
- sqlwarn3 — Set to "W" when mismatching number of result columns and result host variables in a FETCH or SELECT statement.
- sqlwarn4 — Set to "W" when preparing (prepare) an UPDATE or DELETE statement without a WHERE clause.
- sqlwarn5 — Currently unused.
- sqlwarn6 — Set to "W" when the error returned in sqlcode caused the abnormal termination of an open transaction.
- sqlwarn7 — Currently unused.

sqlext — An 8-byte character string variable not currently in use.

Error Handling

The following sections describe the types of errors returned to OpenSQL sessions, and the methods used to handle errors.

The SQLSTATE Variable

The SQLSTATE variable is a 5-character string in which OpenSQL returns the status of the last SQL statement executed. The values returned in SQLSTATE are specified in the ANSI/ISO Entry SQL-92 standard. For details about the requirements for declaring the SQLSTATE variable in embedded programs, see the *Embedded SQL Companion Guide*.

If queries are executed while connected to a DBMS that does not support SQLSTATE, SQLSTATE is set to '5000K' (meaning “SQLSTATE not available”). This result does not necessarily mean that an error occurred. To check the results of the query, one of the other error-checking methods must be used. SQLSTATE is not available within database procedures. However, a routine that directly executes a database procedure can check SQLSTATE to determine the result of the procedure call.

The following is a brief example illustrating the use of SQLSTATE in an embedded program:

```
exec sql begin declare section;

 character     SQLSTATE(5)

exec sql end declare section;

exec sql connect mydatabase;

if SQLSTATE <> "00000" print 'Error on connection!'
```

Local and Generic Errors

A local error is a specific error issued by a specific server, such as Ingres or the Enterprise Access to IBM DB2 UDB product. All server-specific local errors are also mapped into generic errors, enabling applications to handle errors returned from a variety of servers in a consistent way.

For example, the Ingres DBMS returns the local error number 4702 for a timeout error, but other database management systems may return different error numbers for a timeout error. To handle errors consistently, OpenSQL maps the different local timeout error numbers to the same generic error number.

By default, Enterprise Access servers return errors as follows:

- Generic errors
  - Returned to sqlcode (an SQLCA variable) as a negative value
  - Returned when your application issues the INQUIRE_SQL(ERRORNO) statement
- Local errors
  - Returned in sqlerrd(1), the first element of the SQLCA's sqlerrd array
  - Returned when your application issues the INQUIRE_SQL(DBMSERROR) statement

To reverse this arrangement (so that local error numbers are returned to errorno and sqlcode and generic errors to dbmserror and sqlerrd(1)), use the SET_SQL(ERRORTYPE) statement. To obtain the text of error messages, use the INQUIRE_SQL(ERRORTEXT) statement or check the SQLCA variable sqlerrm.

Error Message Format

Every generic error message consists of an error code and the accompanying error message text.

All generic error codes begin with E_, followed by one or two letters plus a 4-digit hexadecimal number, and, optionally, descriptive text or the decimal equivalent of the hex error code. For example:

```
E_GEC2EC_SERIALIZATION
```

indicates a serialization failure (deadlock).

The content and format of local error messages are determined by the local DBMS.

Display of Error Messages

If you are working in one of the forms-based user interfaces, such as the Terminal Monitor, error messages display on a single line across the bottom of the terminal screen. The text appears first, followed by the error code. If the text is longer than one line, press the Help key to display the rest of the message. To clear the error message from the screen, press the Return key.

If you are not working in a forms-based user interface, OpenSQL displays the error code followed by the entire message text.

If you have included an SQLCA, embedded OpenSQL applications do not automatically display error messages. You must provide program code to do so.

Error Handling in Embedded Applications

OpenSQL provides a variety of tools for trapping and handling errors in embedded applications, including:

- The SQLCA
- The WHENEVER statement
- Handler routines
- INQUIRE statements
- The IIseterr function

Error Information from SQLCA

The SQL Communications Area (SQLCA) is a collection of host language variables whose values provide status and error information about embedded OpenSQL database statements. (The status of forms statements is not reflected in SQLCA variables.) If your application does not have an SQLCA, the default is to display errors and continue with the next statement if possible.

Two variables in the SQLCA contain error information: sqlcode and sqlerrm. The value in sqlcode indicates one of three conditions:

- **Success** - Sqlcode contains zero.
- **Error** - Sqlcode contains the error number as a negative value.
- **Warning** - (Set when the statement executed successfully but an exceptional condition occurred.) Sqlcode contains +100, indicating that no rows were processed by a DELETE, FETCH, INSERT, UPDATE, MODIFY, COPY, or CREATE TABLE...AS statement.

The sqlerrm variable is a varying length character string variable that contains the text of the error message. The maximum length of sqlerrm is 70 bytes. If the error message exceeds that length, OpenSQL truncates the message when it assigns it to sqlerrm. To retrieve the full message, use the INQUIRE_SQL statement (see page 162). In some host languages, this variable has two parts: sqlerrml, a 2-byte integer indicating how many characters are in the buffer, and sqlerrmc, a 70-byte fixed length character string buffer.

The SQLCA also contains eight 1-byte character variables, sqlwarn0 - sqlwarn7, that are used to indicate warnings. These variables are described in The SQL Communications Area (SQLCA) (see page 162).

The SQLCA is often used in conjunction with the WHENEVER statement, which defines a condition and an action to take whenever that condition is true. The conditions are set to true by values in the sqlcode variable. For example, if sqlcode contains a negative error number, then the sqlerror condition of the WHENEVER statement is true, and any action specified for that condition is performed.

You can also access the SQLCA variables directly. For information about implementing the SQLCA in an application and using its variables, see the *Embedded SQL* Companion *Guide*.

Error Trapping Using Whenever Statement

The WHENEVER statement specifies a particular action to be performed whenever a particular condition is true. Since conditions are set to true by values in the SQLCA sqlcode, the WHENEVER statement responds only to errors generated by embedded OpenSQL database statements. Forms statements do not set sqlcode.

The conditions that indicate errors or warnings are as follows:

sqlwarning

Indicates that the executed OpenSQL database statement produced a warning condition. Sqlwarning becomes true when the SQLCA sqlwarn0 variable is set to "W."

sqlerror

Indicates that an error occurred in the execution of the database statement. Sqlerror becomes true when the SQLCA sqlcode variable contains a negative number.

There are two other conditions that are more closely related to status conditions rather than error conditions. For a complete list of the conditions, see Whenever in the chapter "OpenSQL Statements."

The actions that can be specified for these conditions are listed in the following table:

Action	Description
CONTINUE	Execution continues with the next statement.
STOP	Prints an error message and terminates the program's execution. Pending updates are not committed.
GOTO *label*	Performs a host language "go to."
CALL *procedure*	Calls the specified host language procedure. If call sqlprint is specified, the sqlprint procedure prints the error or warning message and continues with the next statement.

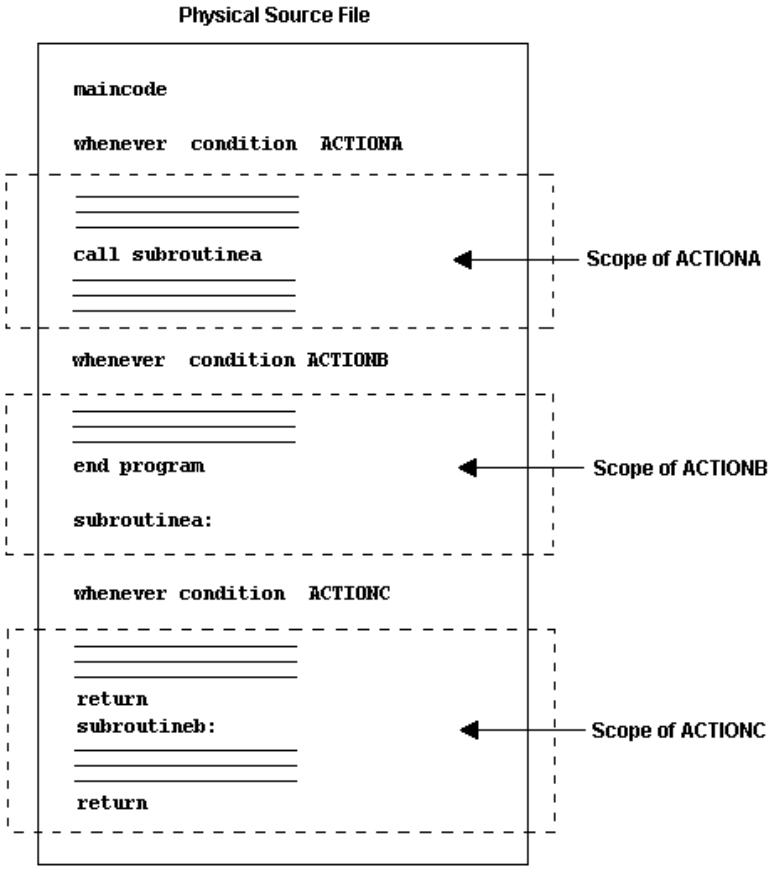
In an application program, a WHENEVER statement is in effect until the next WHENEVER statement (or the end of the program). For example, if you put the following statement in your program:

```
exec sql whenever sqlerror call myhandler;
```

OpenSQL traps errors for all database statements in your program that physically follow the WHENEVER statement to the procedure, myhandler. A WHENEVER statement does not affect the statements that physically precede it.

If your program includes an SQLCA, OpenSQL will not display error messages unless your application issues a WHENEVER ... SQLPRINT statement, or you set II_EMBED_SET to sqlprint.

Scope of the Whenever Statement



How You Define an Error Handler

An error handling function can be defined to be called when OpenSQL errors occur. To do this, you must:

- Write the error handling routine and link it into your embedded OpenSQL application.
- In the application, issue the following SET statement:

  ```
  exec sql set_sql(errorhandler = error_routine)
  ```

where:

error_routine is the name of the error-handling routine you created. Do not declare *error_routine* in an OpenSQL declare section, and do not precede *error_routine* with a colon (:). The *error_routine* argument must be a function pointer.

When this form of error-trapping is enabled, all OpenSQL errors are trapped to your routine until you disable error-trapping (or until the application terminates). Forms errors are not trapped.

To disable the trapping of errors to your routine, your application must issue the following set statement:

```
exec sql set_sql(errorhandler = 0 | :error_var)
```

where *error_var* is a host integer variable having a value of 0.

Your error-handling routine must not issue any database statements in the same session in which the error occurred. If it is necessary to issue database statements in an error handler, open a session or switch to another session.

To obtain error information, your error-handling routine should issue the INQUIRE_SQL statement.

Error Checking Using Inquire Statements

There are two inquire statements that can perform error checking: INQUIRE_SQL and INQUIRE_FRS. Both statements return error numbers and the associated error text using the constants errorno and errortext. INQUIRE_SQL returns the error number and text for the last executed OpenSQL database statement. INQUIRE_FRS returns information about the last executed forms statement.

Unlike the WHENEVER statement, an inquire statement must be executed immediately after the database or forms statement in question. The INQUIRE_SQL returns a generic error number in errorno by default. OpenSQL can be directed to return a local error number in errorno. For more information, see Local and Generic Errors (see page 166).

Neither of the inquire statements suppress the display of error messages. Both of the inquire statements return a wide variety of information in addition to error numbers and text.

For a complete list of the information returned by INQUIRE_SQL, see the chapter "OpenSQL Statements." For details about INQUIRE_FRS, see the *Forms-based Application Development Tools User Guide.*

Error Message Suppression

The IIseterr function is a feature that allows the display of error messages to be suppressed. Although IIseterr is intended for use in Ingres 4GL applications, it can also be used to suppress error messages generated by forms statements. For details, see the *4GL Reference Guide.*

If IIseterr is used in an embedded OpenSQL program that makes use of the SQLCA, errors returned by embedded OpenSQL database statements do not interact with IIseterr. If your program does not use the SQLCA, errors resulting from both forms statements and embedded OpenSQL database statements are passed through IIseterr, if it is present. For ease of use and implementation, it is recommended that you use the SQLCA and WHENEVER statements to handle embedded OpenSQL database statement errors.

Program Termination When Errors Occur

The SET_SQL(PROGRAMQUIT) statement specifies how an embedded OpenSQL application handles the following types of errors:

- Attempting to execute a query when not connected to a database
- Enterprise Access product or DBMS server failure
- Communications service failure

By default, when these types of errors occur, OpenSQL issues an error but lets the program continue.

To force an application to abort when one of these errors occur, issue the following SET_SQL statement:

```
exec sql set_sql (programquit = 1);
```

If an application aborts as the result of one of the previously listed errors, OpenSQL issues an error, then rolls back open transactions and disconnects all open sessions. (To disable aborting and restore the OpenSQL default behavior, specify programquit = 0.)

Errors affected by the programquit setting belong to the generic error class GE_COMM_ERROR, which is returned to errorno as 37000, and to sqlcode (in the SQLCA) as -37000. An application can check for these errors and, when detected, must disconnect from the current session. After disconnecting from the current session, the application can attempt another connection, switch to another session (if using multiple sessions), or perform clean-up operations and quit.

You can also specify programquit using II_EMBED_SET.

To determine the current setting for this behavior, use the INQUIRE_SQL statement:

```
exec sql inquire_sql (int_variable = programquit);
```

This returns a 0 if programquit is not set (OpenSQL continues on any of the errors) or 1 if programquit is set (OpenSQL exits the application on these errors).

Handling Deadlock

Deadlock occurs when two transactions are each waiting for the other to release a part of the database to enable it to complete its update. Transactions that handle deadlocks in conjunction with other errors can be difficult to code and test, especially if cursors are involved.

To facilitate the proper coding and testing for these situations, you can use the following three template programs as guides in your resolution of similar error situations. Deadlock conditions are identified by the generic error code value of E_GEC2EC_SERIALIZATION.

The following templates assume the default OpenSQL transaction behavior (SET AUTOCOMMIT is OFF).

Noncursor Template for Handling Deadlock

This template assumes your transactions do not contain a cursor.

```
exec sql whenever not found continue;
exec sql whenever sqlwarning continue;
exec sql whenever sqlerror goto err;
/* branch on error */

start:
exec sql insert into ...
exec sql update ...
exec sql select ...

exec sql commit;
goto end;
err:
exec sql whenever sqlerror call sqlprint;
  if (sqlca.sqlcode = E_GEC2EC_SERIALIZATION) then
   goto start;
  else if (sqlca.sqlcode < 0) then
   exec sql inquire_sql (:err_msg = errortext);
   exec sql rollback;
   print 'Error', err_msg;
  end if;

end:
```

Single Cursor Template for Handling Deadlock

This template is similar to the first, but with a single cursor added.

```
exec sql whenever not found continue;
exec sql whenever sqlwarning continue;
exec sql whenever sqlerror goto err;
/* branch on error */

exec sql declare c1 cursor for ...

start:
exec sql open c1;
while more rows loop
  exec sql fetch c1 into ...
  if (sqlca.sqlcode = E_GE0064_NO_MORE_DATA) then
   exec sql close c1;
   exec sql commit;
   goto end;
  end if;

  exec sql insert into ...
  exec sql update ...
  exec sql select ...

end loop;
goto end

err:
exec sql whenever sqlerror call sqlprint;
if (sqlca.sqlcode = E_GEC2EC_SERIALIZATION) then
  goto start;
else if (sqlca.sqlcode \ 0) then
  exec sql inquire_sql (:err_msg = errortext);
  exec sql rollback;
  print 'Error', err_msg;
end if;

end:
```

Master/Detail Template for Handling Deadlock

This template is like the previous one, but includes two cursors with a master/detail relationship.

```
exec sql whenever not found continue;
exec sql whenever sqlwarning continue;
exec sql whenever sqlerror goto err;
/* branch on error */

exec sql declare master cursor for ...
exec sql declare detail cursor for ...

start:
exec sql open master;
while more master rows loop
   exec sql fetch master into ...
  if (sqlca.sqlcode = E_GE0064_NO_MORE_DATA) then
   exec sql close master;
   exec sql commit;
   goto end;
  end if;

/* ...queries using master data... */
  exec sql insert into ...
  exec sql update ...
  exec sql select ...

  exec sql open detail;
  while more detail rows loop
   exec sql fetch detail into ...
   if (sqlca.sqlcode = E_GE0064_NO_MORE_DATA) then
    exec sql close detail;
    end loop; /* drops out of detail fetch loop */
   end if;

/* ...queries using detail & master data... */
   exec sql insert into ...
   exec sql update ...
   exec sql select ...

  end loop;  /* end of detail fetch loop */
```

```
/* ...more queries using master data... */
  exec sql insert into ...
  exec sql update ...
  exec sql select ...

end loop;  /* end of master fetch loop */
goto end

err:
exec sql whenever sqlerror call sqlprint;
if (sqlca.sqlcode = E_GEC2EC_SERIALIZATION) then
  goto start;
else if (sqlca.sqlcode < 0) then
  exec sql inquire_sql (:err_msg = errortext);
  exec sql rollback;
  print 'Error', err_msg;
end if;
end:
```

Multiple Session Connections

OpenSQL provides embedded applications with the ability to maintain multiple sessions. An application can open an initial session and, with subsequent CONNECT statements, open additional sessions with the same or different types of servers or Enterprise Access products.

Session Identifier—Connect to Multiple Sessions

Individual sessions in a multiple session application are identified by a session identifier that is specified when the CONNECT statement for each session is issued. Each CONNECT statement in a multiple session application, including the first CONNECT statement, must specify a session identifier.

It is possible to open new sessions with previously unconnected databases or with databases already associated with an open session. New sessions can be opened under different user names (for Enterprise Access products that support the CONNECT statement's IDENTIFIED BY clause) and can be entered using different option flags. For a description of syntax and optional flags for connect, see the *SQL Reference* Guide.

Once an application issues a CONNECT statement, the session initiated by the statement is the current session and all subsequent embedded OpenSQL statements apply to the database associated with that statement until another CONNECT statement or a SET_SQL statement (to switch sessions) is issued.

If an error occurs during a connection attempt, the program is no longer connected to any session after the failure, whether or not it was connected before the attempted connection. After the failure of an attempt to connect, the program must either attempt to connect again or switch to a previously established session before continuing.

Session Switching

To switch from one open session to another, use the SET_SQL statement. To open a new session, issue the CONNECT statement. To determine the session identifier for the current session, use the INQUIRE_SQL statement.

Applications can switch sessions freely. Note that session switching is supported under the following circumstances:

- In a transaction.
- While cursors are open.
- In OpenSQL statement blocks (such as select loops).

  The code for the nested session must be inside a host language subroutine. If it is not, the SQL preprocessor will issue an error.
- In subroutines called by a WHENEVER statement.

After an application switches sessions, the error information obtained from the SQLCA or the INQUIRE_SQL statement is not updated until an OpenSQL statement has completed in the new session.

Session Termination

To terminate the current session, the application issues the DISCONNECT statement. An optional session identifier parameter exists to identify the current session specifically if desired.

When an application terminates one of many open sessions, it is not automatically placed in another session. The application must issue either a CONNECT or SET_SQL statement to establish the current session. If the application fails to do this, OpenSQL returns an error when the next OpenSQL statement is issued.

To terminate a specific session, specify the session identifier. To obtain the session identifier for the current session, issue the INQUIRE_SQL(:*session_id*=SESSION) statement. To disconnect all sessions, issue the DISCONNECT ALL statement.

Multiple Sessions and the SQLCA

The SQL Communications Area (SQLCA) is a data area in which OpenSQL passes query status information to your application program. Although an application can sustain multiple sessions, there is only one SQLCA per application. By contrast, the values returned by INQUIRE_SQL(ERRORCODE) and INQUIRE_SQL(ERRORTEXT) are specific to a session.

If you switch sessions in a select loop (for example, by calling a routine that switches sessions) and execute database statements in the alternate session, the values in the SQLCA will be reset. When you return to the original session, the SQLCA will not reflect the results of the select loop.

When switching between sessions, the values in the SQLCA fields are not updated until after the first OpenSQL statement in the current session has completed. If you switch sessions, the values in the SQLCA will apply to the previous session until an OpenSQL statement in the current session resets them. In contrast, the error information returned by INQUIRE_SQL(ERRORTEXT and ERRORNO) always apply to the current session.

When an application switches sessions within a select loop or other block statement, the SQLCA field values are updated to reflect the status of the statements executed inside the nested session. After the application switches back to the session with the loop, the SQLCA field values reflect the status of the last statement in the nested session. Sqlcode and sqlwarn are not updated until the statement immediately following the loop completes. (The information obtained by INQUIRE_SQL is not valid until the statement following a loop completes.) For this reason, the application should reset the sqlcode and sqlwarn fields before continuing the loop.

Multiple Sessions and the DBMS

Each session in a multiple-session application requires an independent connection to the Enterprise Access products or DBMS server. When creating multiple-session applications, keep the following points in mind:

- In a multi-session application, an application can encounter deadlock against itself. For example, one session may attempt to update a table that was locked by another session.
- An application can also lock itself out in an undetectable manner. For example, if a table is updated in a transaction in one session and then selected from in another transaction in a second session, the second session waits indefinitely.
- For sessions connected to Ingres databases, be sure that the server parameter connect_limit is large enough to accommodate the number of sessions required by the application.

Multiple Session Examples

This section presents two examples of multiple sessions. The first example, illustrates session switching using two open sessions in a forms-based application. These sessions gather project information for updating the projects database using the personnel database to verify employee identification numbers.

```
exec sql begin declare section;
   empid integer;
   found integer;
   ...
exec sql end declare section;

/* Set up two database connections */
exec sql connect 'projects/rdb' session 1;
exec sql connect 'unixbox::personnel/db2udb' session 2;

/* Set 'projects' database to be current session */
exec sql set_sql (session = 1);

exec frs display projectform;
exec frs activate field empid;
exec frs begin;

/* Validate user-entered employee id against master
** list of employees in personnel database. */

found = 0;
exec sql getform (:empid = empid);

/* Switch to 'personnel' database session */
exec sql set_sql (session = 2);
exec sql repeated select 1 into :found from employee
  where empid = :empid;

/* Switch back to 'project' database session */
exec sql set_sql (session = 1);
if (found !=1) then
 exec frs message 'Invalid employee identification';
 exec frs sleep 2;
else
  exec frs resume next;
endif;

exec frs end;
program code to validate other fields in
        'projectform'
exec frs activate menuitem 'Save';
exec frs begin;
get project information and update 'projectinfo'
         table
exec frs end;
...

exec sql disconnect;
exec sql set_sql (session = 2);
exec sql disconnect;
```

The second example illustrates session switching inside a select loop and the resetting of status fields. The main program processes sales orders and calls the subroutine, new_customer, for every new customer.

The main program:

```
...
exec sql include sqlca;
exec sql begin declare section;

/* Include output of DCLGEN for declaration
** of record order_rec

*/
exec sql include 'decls';
exec sql end declare section;
exec sql connect 'customers/alb' session 1;
exec sql connect sales session 2;
...

exec sql select * into :order_rec from orders;
exec sql begin;
if (order_rec.new_customer = 1) then
  call new_customer(order_rec);
endif
process order;
exec sql end;
...

exec sql disconnect;

exec sql set_sql(session = 1);
exec sql disconnect;
```

The subroutine, new_customer, from the select loop, containing the session switch:

```
subroutine new_customer(record order_rec)
begin;

exec sql set_sql(session = 1);
exec sql insert into accounts values (:order_rec);

process any errors;
exec sql set_sql(session = 2);

sqlca.sqlcode = 0;
sqlca.sqlwarn.sqlwarn0 = ' ';

end subroutine;
```

Database Procedures

A *database procedure* is a named routine that is stored in the host DBMS or linked to an Enterprise Access or EDBC product.

Database procedures must be created, declared, and executed.

How Database Procedures Are Created

Database procedures can be created using the following methods:

- **Ingres Database Procedures** - The Ingres DBMS (and some Enterprise Access and EDBC products) allows you to create database procedures using the CREATE PROCEDURE statement.
- **Host DBMS Procedures** - Enterprise Access and EDBC provide access to procedures located in the host DBMS. (These procedures are created and maintained in the host DBMS.) The procedure must be declared to Enterprise Access or EDBC by issuing the REGISTER PROCEDURE statement. For details regarding support of host DBMS procedures, see your Enterprise Access or EDBC product guide. For details about creating and managing the host DBMS procedures, see your host DBMS documentation.
- **Enterprise Access and EDBC Procedures** - Enterprise Access and EDBC to host DBMSs that do not support database procedures provide an alternate mechanism for database procedures: object code modules for the routine are linked into the Enterprise Access or EDBC executable program. These routines must be declared to Enterprise Access or EDBC by issuing the REGISTER PROCEDURE statement. For details on the creating and registering this kind of procedure, see your Enterprise Access or EDBC product guide.
- **Ingres Star** - Using Ingres Star, you can execute database procedures that are located in a remote database, Enterprise Access, or EDBC.

Note: OpenSQL does not control the transaction behavior that occurs when executing a database procedure. Transaction behavior is determined by the host DBMS.

Register Procedure Statement—Register Database Procedure

The REGISTER PROCEDURE statement defines the interface between an application and a database procedure when support for the CREATE PROCEDURE statement is not available. Creation and maintenance of the database procedure is dependent on the host DBMS, Enterprise Access, or EDBC.

The REGISTER PROCEDURE statement defines the procedure name, its parameters and their types, and the host DBMS, Enterprise Access, or EDBC information required to access the procedure.

Non-row returning REGISTER PROCEDURE syntax:

```
REGISTER PROCEDURE procedure_name
        [(parameter_definition {,parameter_definition})]
        AS IMPORT FROM '...'
        WITH [with_clause]
```

Note: The BYREF keyword can be used for parameters.

Row-returning REGISTER PROCEDURE syntax:

```
REGISTER PROCEDURE procedure_name
      [(parameter_definition {,parameter_definition})]
      AS IMPORT FROM '...'
      RESULT ROW (return_type_list);
```

Note: The BYREF keyword cannot be used for parameters.

procedure_name

Specifies the procedure name to be used in the OpenSQL EXECUTE PROCEDURE or REMOVE PROCEDURE statements.

as import from

Specifies host DBMS, Enterprise Access, or EDBC information required to identify the procedure being registered.

with *with_clause*

Specifies additional information that may be required by an Enterprise Access or EDBC product.

return_type_list

Lists data types.

The *parameter_definition* is specified as:

parameter_name datatype [NOT | WITH NULL]
[NOTE DEFAULT | [WITH] DEFAULT [*default_value*]]
[BYREF]
[IS *host_info*]

Parameters are nullable unless you specify NOT NULL.

The *default_value* can be a numeric or character literal or one of the following constants: NULL, USER, CURRENT_DATE, or CURRENT_TIME. If the default value is omitted, a system-generated default is assigned. If the DEFAULT clause is omitted, DEFAULT NULL is assumed.

The BYREF keyword specifies that the parameter is passed by reference, enabling the procedure to return a value in the parameter. The BYREF keyword must also be used in the EXECUTE PROCEDURE statement to obtain the returned value. BYREF cannot be used if the procedure returns rows.

The IS clause specifies additional information about the parameter, as required by the host DBMS, Enterprise Access, or EDBC product. The host information must be enclosed in single quotes.

Remove Procedure Statement—Delete a Procedure Registration

To delete a procedure registration, use the REMOVE PROCEDURE statement. After deleting the registration, the procedure cannot be executed (unless you register the procedure again).

The REMOVE PROCEDURE syntax is:

```
REMOVE PROCEDURE procedure_name
```

For details about using the REGISTER PROCEDURE or REMOVE PROCEDURE statements, see your Enterprise Access or EDBC product guide.

Guidelines for Executing Database Procedures

To execute a database procedure, issue the EXECUTE PROCEDURE statement.

To ensure portability of your application code and consistency of the transactions in your application, observe the following guidelines for executing database procedures:

- Do not issue COMMIT or ROLLBACK statements within a database procedure, because these statements or their equivalents may not be supported in all host database management systems.
- Issue a COMMIT or ROLLBACK statement before and after executing a database procedure.
- If an error occurs while a database procedure is being executed, the current transaction may be rolled back by the host DBMS. While this is permitted by OpenSQL, it is not required. After executing a database procedure, your application should check for errors and, if necessary, roll back the transaction.

DBMS Extensions

OpenSQL statements work with all Enterprise Access, EDBC, and DBMS servers. However, the underlying DBMS typically supports additional SQL statements and extensions. (For example, Ingres SQL includes statements that support rules and database events, and the DB2 UDB SQL CREATE TABLE statement includes extensions governing "editprocs.")

OpenSQL provides the following methods for issuing DBMS-specific statements from an OpenSQL application:

- Ingres SQL extensions

  Extensions can be coded directly in an application. The embedded SQL or 4GL preprocessors recognize the extension and execute it properly. However, extended statements will fail if issued against a non-Ingres DBMS.

- DIRECT EXECUTE IMMEDIATE

  The DIRECT EXECUTE IMMEDIATE statement passes a statement to the underlying DBMS. OpenSQL does not attempt to process or translate the statement. The DIRECT EXECUTE IMMEDIATE statement can be used with any SQL statement that can be executed dynamically (statements that can be issued with a DBMS EXECUTE IMMEDIATE). Statements that return rows (for example, SELECT or FETCH) cannot be issued.

- Enterprise Access and EDBC with clause

  Many Enterprise Access and EDBC products support WITH clauses that provide the ability to access DBMS extensions to database connection and Data Definition Language (DDL) SQL statements.

Enterprise Access and EDBC With Clause

The Enterprise Access and EDBC WITH clause enables DBMS-specific options to be specified in an OpenSQL statement. EDBC and Enterprise Access servers process only the options directed at them, and ignore the rest. Valid options depend on the specific Enterprise Access or EDBC product, and DBMS. For information on valid WITH clause parameters, see your Enterprise Access or EDBC product guide. The DBMS, Enterprise Access, or EDBC is responsible for performing the specified action or translating the WITH clause to the syntax required by the underlying DBMS.

The following OpenSQL statements support the Enterprise Access and EDBC WITH clause:

- CONNECT
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- DROP INDEX
- DROP TABLE
- DROP VIEW

With Clause Syntax

The WITH clause can contain options intended for different Enterprise Access or EDBC products. Enterprise Access, EDBC, and Ingres options can be specified in a WITH clause.

The WITH clause has the following syntax:

```
WITH [db_id_]option_name [= option_value]
        {, [db_id_]option_name [= option_value]}
```

db_id_

Specifies the server class of the Enterprise Access or EDBC product for which the option is specified. The trailing underscore is required, and the option_name parameter must be appended with no intervening space.

If this parameter is specified, only the specified Enterprise Access or EDBC product will process the option. If this parameter is omitted, all Enterprise Access, EDBC, or database management systems will attempt to process the option. Enterprise Access and EDBC will ignore options they cannot process. Valid values are:

DB2_ DB2 or DB2 UDB

IMS_ IMS

RDB_ Rdb/VMS

RMS_ RMS

SQL_ All Ingres relational or SQL-based Enterprise Access or EDBC products

option_name

Specifies the name of the option. If this is an Enterprise Access-specific or EDBC-specific option, option_name must be preceded by the *db_id*. For details about the product-specific options, see your Enterprise Access or EDBC product guide.

option_value

Specifies the value of the option (if required). This value can be specified using a quoted or unquoted character string, numeric literal, or variable.

All values must be specified using simple data formats, such as integers, numerics, names, or strings. If a complex value is required, it must be encoded in a quoted string.

To specify a list of values, use a comma-separated list enclosed in parentheses. For example:

```
with myoption=(value1, value2, value3)
```

If an option is specified using a string variable, (for example with :stringvar,) and no value is to be provided, the variable must contain the string "NULL." Enterprise Access or EDBC will ignore the option.

WITH Clause Examples

1. Connect to a DB2 subsystem DB2T and set the default database for table creation to mydb.

   ```
   connect 'mvs1::db2t/db2' with db2_ct_option =
   'mydb';
   ```

   EDBC for DB2 receiving the preceding connect request will issue the DB2 statements required to connect to the DB2T DB2 subsystem with the indicated default database.

2. Create a database table and specify DBMS-specific extensions for DB2.

   ```
   create table newtab (col1 integer, col2 integer not null)
   with db2_ct_option = 'audit all', alb_type = private;
   ```

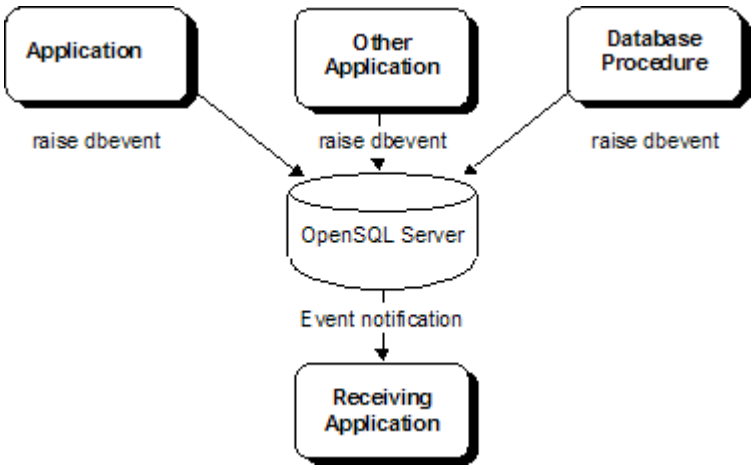
Database Events

Database events are an extended feature of OpenSQL. This means that not all OpenSQL servers support database events. If an OpenSQL server supports database events, there will be a row in iidbcapabilites with a cap_capability of DBEVENTS and a cap_value of Y. To see if a particular OpenSQL server supports database events or for any restrictions, refer to the documentation for that server.

Database events enable an application or the DBMS to notify other applications that a specific event has occurred. An *event* is any occurrence that your application program is designed to handle. Note that support for database events is optional for OpenSQL servers.

The following diagram illustrates a typical use of database events: various applications raise database events, and the DBMS notifies a monitor (receiving) application that is registered to receive the database events.

The monitor application responds to the database events by performing the actions the application designer specified when writing the monitor application.



Note: In the diagram above, OpenSQL Server refers only to OpenSQL servers that support database events.

Database events can be raised by an application that issues the RAISE DBEVENT statement.

Database Event Statements

The SQL statements required to define and use database events are as follows:

- CREATE DBEVENT
- RAISE DBEVENT
- REGISTER DBEVENT
- GET DBEVENT
- REMOVE DBEVENT
- DROP DBEVENT
- INQUIRE_SQL
- SET_SQL

Creating a Database Event

To create a database event, use the CREATE DBEVENT statement:

```
CREATE DBEVENT event_name
```

where *event_name* is a unique database event name and a valid object name.

Database events can be raised by all applications connected to the database, and received by all applications connected to the database and registered to receive the database event.

Raising a Database Event

To raise a database event, use the RAISE DBEVENT statement:

```
RAISE DBEVENT event_name [event_text]
```

The RAISE DBEVENT statement can be issued from interactive or embedded SQL applications. When the RAISE DBEVENT statement is issued, the DBMS sends a database event message to all applications that are registered to receive *event_name*. If no applications are registered to receive a database event, raising the database event has no effect.

The optional *event_text* parameter is a string (maximum 256 characters) that can be used to pass information to receiving applications. For example, you can use *event_text* to pass the name of the application that raised the database event, or to pass diagnostic information.

Registering to Receive a Database Event

To register an application to receive database events, use the REGISTER DBEVENT statement:

```
REGISTER DBEVENT event_name
```

where *event_name* is an existing database event. Sessions must register for each database event to be received. For each database event, the registration is in effect until the session issues the REMOVE DBEVENT statement or disconnects from the database.

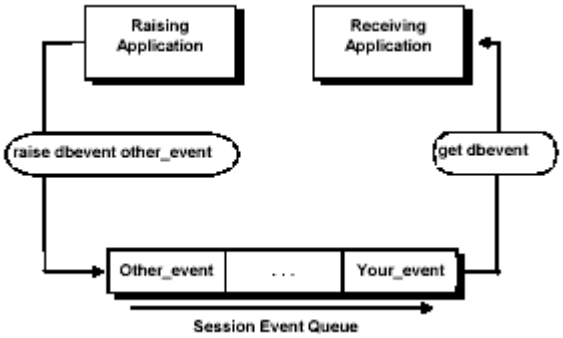
The DBMS issues an error if a session attempts to register for a database event for which the session does not have register privilege.

The REGISTER DBEVENT statement can be issued from interactive or embedded SQL program.

How a Database Event is Received

To receive a database event and its associated information, an application must perform two steps:

1. Remove the next database event from the session's database event queue (using GET DBEVENT or, implicitly, using WHENEVER DBEVENT or SET_SQL DBEVENTHANDLER).
2. Inquire for database event information (using INQUIRE_SQL). The GET DBEVENT statement gets the next database event, if any, from the queue of database events that have been raised and for which the application session has registered, as shown in the following illustration:



GET DBEVENT returns database events for the current session only; if an application runs multiple sessions, each session must register to receive the desired database events, and the application must switch sessions to receive database events queued for each session.

The optional WITH clause specifies whether your application waits for a database event to arrive in the queue. If GET DBEVENT WITH WAIT is specified, the application waits indefinitely for a database event to arrive. If GET DBEVENT WITH WAIT=*wait_value* is specified, the application waits the specified number of seconds for a database event to arrive. If no database event arrives in the specified time, the GET DBEVENT statement times out, and no database event is returned. If GET DBEVENT WITH NOWAIT is specified, the DBMS checks for a database event and returns immediately. The default is NOWAIT.

The WITH WAIT clause cannot be specified if the GET DBEVENT statement is issued in a select loop or user-defined error handler.

To obtain database event information, your application must issue the INQUIRE_SQL statement, and specify one or more of the following parameters:

dbeventname

The name of the database event (in lowercase letters). If there are no database events in the database event queue, the DBMS returns an empty string (or a string containing blanks, if your host language uses blank padded strings).

dbeventowner

The username of the user that created the database event.

dbeventdatabase

The database in which the database event was raised; returned in lowercase letters.

dbeventtime

The date and time the database event was raised, in date format. The receiving host variable must be a string (minimum length of 25 characters).

dbeventtext

The text, if any, specified in the optional *event_text* parameter by the application that raised the database event. The receiving variable must be a 256-character string. If the receiving variable is too small, the text is truncated.

Methods for Processing Database Events

Three methods can be used to process database events:

- Use the GET DBEVENT statement to explicitly consume each database event from the database event queue of the session. Typically, a loop will be constructed that polls for database events and calls routines that appropriately handle different database events. *Get* dbevent is a low overhead statement: it polls the application's database event queue and not the server.
- Trap database events using the WHENEVER DBEVENT bevent statement. To display database events and remove them from the database event queue, specify WHENEVER DBEVENT SQLPRINT. To continue program execution without removing database events from the database event queue, specify WHENEVER DBEVENT CONTINUE. To transfer control to a database event handling routine, specify WHENEVER DBEVENT GOTO or WHENEVER DBEVENT CALL. To obtain the database event information, the routine must issue the INQUIRE_SQL statement.
- Trap database events to a handler routine, using SET_SQL DBEVENTHANDLER. To obtain the database event information, the routine must issue the INQUIRE_SQL statement.

Note: If your application terminates a select loop using the ENDSELECT statement, unread database events may be purged. Note that dbevents are received only during communication between the application and the DBMS server while performing SQL query statements. When notification is received, the application programmer should ensure that all database events in the database events queue are processed by using the GET DBEVENT loop, which is described below.

Using GET DBEVENT

To get a database event registration, use the GET DBEVENT statement:

```
EXEC SQL GET DBEVENT [WITH NOWAIT | WAIT [= wait_value]];
```

To specify whether the GET DBEVENT statement waits for database events or checks the queue and returns immediately, specify the WITH [NO]WAIT clause. By default, GET DBEVENT checks and returns immediately.

If WITH WAIT is specified, GET DBEVENT waits indefinitely for the next database event to arrive. If WITH WAIT = *wait_value* is specified, GET DBEVENT returns when a database event arrives or when *wait_value* seconds have passed, whichever occurs first.

The following example shows a loop that processes all database events in the database event queue. The loop terminates when there are no more database events in the queue.

```
loop
exec sql get dbevent;
exec sql inquire_sql (:event_name =

dbeventname);
if event_name = 'event_1'
process event 1
else
if event_name = 'event_2'
process event 2
else
...
endif
until event_name = ''
```

Using WHENEVER DBEVENT

To specify an action to occur whenever a DBEvent is raised, use the WHENEVER statement:

```
EXEC SQL WHENEVER DBEVENT action;
```

The *action* can be one of the following: CONTINUE, STOP, or GO TO *label*.

To use the WHENEVER DBEVENT statement, your application must include an SQLCA. When a database event is added to the database event queue, the sqlcode variable in the SQLCA is set to 710 (as will the standalone SQLCODE variable; SQLSTATE is not affected). However, if a query results in an error that resets sqlcode, the WHENEVER statement will not trap the database event. The database event will still be queued, and your error-handling code can use the GET DBEVENT statement to check for queued database events. To avoid inadvertently (and recursively) triggering the WHENEVER mechanism from within a routine called as the result of a WHENEVER DBEVENT statement, your database event-handling routine should turn off trapping:

```
main program:
exec sql whenever dbevent call event_handler;
...
event_handler:
/* turn off the whenever event trapping */
exec sql whenever dbevent continue;
exec sql inquire_sql(:evname=dbeventname...);
process events
return
```

Using User-defined Database Event Handlers

To define your own database event-handling routine, use the EXEC SQL SET_SQL(DBEVENTHANDLER) statement. This method traps database events as soon as they are added to the database event queue; the WHENEVER method must wait for queries to complete before it can trap database events. For more information, see the Set_sql section in the chapter "SQL Statements."

Removing a Database Event Registration

To remove a database event registration, use the REMOVE DBEVENT statement:

```
REMOVE DBEVENT event_name
```

where *event_name* specifies a database event for which the application has previously registered. After a database event registration is removed, the DBMS will not notify the application when the specified database event is raised. (Pending database event messages are not removed from the database event queue.)

When attempting to remove a registration for a database event that was not registered, the DBMS issues an error.

Dropping a Database Event

To drop a database event, use the DROP DBEVENT statement:

```
DROP DBEVENT event_name
```

where *event_name* is a valid and existing database event name. Only the user that created a database event can drop it. After a database event is dropped, it cannot be raised, and applications cannot register to receive the database event. (Pending database event messages are not removed from the database event queue.) If a database event is dropped while applications are registered to receive it, the database event registrations are not dropped from the DBMS until the application disconnects from the database or removes its registration for the dropped database event. If the database event is recreated (with the same name), it can again be received by registered applications.

To enable or disable the display of database events as they are received by an application, use the following statement:

```
EXEC SQL SET_SQL(DBEVENTDISPLAY = 1 | 0)
```

Specify a value of 1 to enable the display of received database events, or 0 to disable the display of received database events. This feature can also be enabled by using II_EMBED_SET. For details about II_EMBED_SET, see your *System Administrator Guide*.

A routine can be created that will trap all database events returned to an embedded SQL application. To enable or disable a database event-handling routine or function, your embedded SQL application must issue the following SET_SQL statement:

```
EXEC SQL SET_SQL(DBEVENTHANDLER = event_routine | 0)
```

To trap database events to your database event-handling routine, specify event_routine as a pointer to your error-handling function. For information about specifying pointers to functions, see your host language companion guide. Before using the SET_SQL statement to redirect database event handling, create the database event-handling routine, declare it, and link it with your application.

Chapter 8: OpenSQL Statements

This section contains the following topics:

Whenever (see page 320)

OpenSQL Version

This chapter describes the version of OpenSQL indicated by the following values in the iidbcapabilities catalog:

CAP_CAPABILITIES	CAP_VALUE
OPEN/SQL_LEVEL	00602

Context for SQL Statements

For each statement description in this chapter, the following notation is used to indicate the contexts in which the statement can be used:

SQL Interactive session

ESQL Embedded SQL

DBProc Database procedure

OpenAPI OpenAPI

ODBC ODBC

JDBC JDBC

.NET .NET Data Provider

For example:

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

This chapter does not describe Ingres Forms statements. For information about these statements, see the Forms-based Application Development Tools User Guide.

Extended Statements

If the iidbcapabilities catalog contains this row:

CAP_CAPABILITIES	CAP_VALUE
SQL92_COMPLIANCE	ENTRY

these additional statements and features can be used:

- GRANT
- REVOKE
- CREATE SCHEMA
- CREATE TABLE constraints and defaults enhancements

If the iidbcapabilities catalog contains this row:

CAP_CAPABILITIES	CAP_VALUE
SQL92_COMPLIANCE	NONE

these statements and extensions are not supported.

Begin Declare

Valid in: ESQL

The BEGIN DECLARE statement begins a program section that declares host language variables to embedded SQL.

Syntax

The BEGIN DECLARE statement has the following format:

```
EXEC SQL BEGIN DECLARE SECTION
```

Description

All variables used in embedded SQL statements must be declared. A single program can have multiple declaration sections.

The statements that can appear inside a declaration section are:

- Legal host language variable declarations
- An INCLUDE statement that includes a file containing host language variable declarations. (This must be an SQL INCLUDE statement, not a host language include statement.)
- A DECLARE TABLE statement (normally generated by dclgen in an included file)

The END DECLARE section statement marks the end of the declaration section.

Example: Begin Declare

The following example shows the typical structure of a declaration statement:

```
EXEC SQL BEGIN DECLARE SECTION;
  buffer character_string(2000);
  number integer;
  precision float;
EXEC SQL END DECLARE SECTION;
```

Call

Valid in: ESQL

The CALL statement calls the operating system or an Ingres tool.

Syntax

The CALL statement has the following format:

To call the operating system:

```
EXEC SQL CALL SYSTEM (COMMAND = command_string)
```

To call an Ingres tool:

```
EXEC SQL CALL subsystem (DATABASE = dbname {, parameter = value})
```

where:

command_string

Specifies the command to be executed at the operating system level when the operating system is called. If *command_string* is a null, empty, or blank string, the statement transfers the user to the operating system and the user can execute any operating system command. Exiting or logging out of the operating system returns the user to the application.

subsystem

Specifies the name of the Ingres tool.

dbname

Specifies the name of the current database.

parameter

Specifies one or more parameters specific to the called subsystem.

value

Specifies the value assigned to the specified parameter.

The *command_string* can invoke an Ingres tool. For example:

```
EXEC SQL CALL SYSTEM (COMMAND = 'qbf personnel');
```

However, it is more efficient to call the subsystem directly:

```
EXEC SQL CALL qbf (DATABASE = 'personnel');
```

When a subsystem is called directly, the database argument must identify the database to which the session is connected.

The CALL statement is not sent to the database; therefore, it cannot appear in a dynamic SQL statement string. When calling an Ingres tool, an application cannot rely on the dynamic scope of open transactions, open cursors, prepared queries, or repeated queries. The application must consider each subsystem call as an individual DBMS Server session. The Ingres tool commits any open transaction when it starts. For this reason, it is a good practice to commit before calling the subsystem.

Call Description

The CALL statement allows an embedded SQL application to call the operating system or an Ingres tool (such as QBF or Report-Writer).

When used to call the operating system, this statement executes the specified *command_string* as if the user typed it at the operating system command line. After the *command_string* is executed, control returns to the application at the statement following the CALL statement.

If the CALL statement is being used to call an Ingres tool, it is more efficient to call the tool directly, rather than calling the operating system and, from there, calling the tool.

Examples: Call

The following are CALL statement examples:

1. Run a default report on the employee table in the column mode.

   ```
   EXEC SQL COMMIT;
   EXEC SQL CALL report (DATABASE='personnel',
        NAME='employee', MODE='column');
   ```

2. Run QBF in the append mode with the QBF name expenses, suppressing verbose messages.

   ```
   EXEC SQL COMMIT;
   EXEC SQL CALL qbf (DATABASE='personnel',
        QBFNAME='expenses', FLAGS='-mappend -s');
   ```

Close

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The CLOSE statement closes an open cursor.

Syntax

The CLOSE statement has the following format:

```
EXEC SQL CLOSE cursor_name
```

cursor_name

Specifies the cursor name using a quoted or unquoted string literal or a host language string variable. If cursor_name is a reserved word, it must be specified in quotes.

Description

The *cursor_name* must have been previously defined in your source file by a DECLARE CURSOR statement. Once closed, the cursor cannot be used for further processing unless reopened with a second OPEN statement. A COMMIT, ROLLBACK, or DISCONNECT statement closes all open cursors.

A string constant or host language variable can be used to specify the cursor name.

Embedded Usage

In an embedded CLOSE statement, a string constant or host language variable can be used to specify the cursor name.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, Close cannot be directly issued on a cursor name, but can take handles or objects that perform the same task.

Example: Close

The following example illustrates cursor processing from cursor declaration to closing:

```
EXEC SQL DECLARE c1 CURSOR FOR
SELECT ename, jobid
FROM employee
WHERE jobid = 1000;
...
EXEC OPEN c1;
LOOP UNTIL NO MORE ROWS;
EXEC SQL FETCH c1
     INTO :name, :jobid;
PRINT NAME, jobid;
END LOOP;

EXEC SQL CLOSE c1;
```

Commit

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The COMMIT statement commits the current transaction.

Syntax

The COMMIT statement has the following format:

```
[EXEC SQL] COMMIT [WORK]
```

Note: The optional keyword WORK is included for compliance with the ISO and ANSI standards for SQL.

Description

The COMMIT statement terminates the current transaction. Once committed, the transaction cannot be aborted, and all changes it made become visible to all users through any statement that manipulates that data.

Note: If READLOCK=NOLOCK is set, the effect of the transaction is visible before it is committed. This is also true when the transaction isolation level is set to read uncommitted.

The COMMIT statement can be used inside a database procedure if the procedure is executed directly, using the execute procedure statement. However, database procedures that are invoked by a rule cannot issue a COMMIT statement: the commit prematurely terminates the transaction that fired the rule. If a database procedure invoked by a rule issues a COMMIT statement, the DBMS Server returns a runtime error. Similarly a database procedure called from another database procedure must not issue a COMMIT because that leaves the calling procedure outside the scope of a transaction.

Embedded Usage

In addition to terminating the current transaction, an embedded COMMIT statement:

- Closes all open cursors.
- Discards all statements prepared (with the PREPARE statement) during the current transaction.

When a program issues the DISCONNECT statement, an implicit COMMIT is also issued. Any pending updates are submitted. To roll back pending updates before terminating the program, issue a ROLLBACK statement.

Usage in OpenAPI, ODBC, JDBC, .NET

While applications can send a COMMIT query to the DBMS, we recommend that they instead use the interface-specific mechanism for commit in OpenAPI, ODBC, JDBC, and .NET.

Example: Commit

The following embedded example issues two updates, each in its own transaction:

```
exec sql connect 'mvs1::personnel/db2';

exec sql update employee set salary = salary * 1.1
      where rating = 'Good';

exec sql commit;

exec sql update employee set salary = salary * 0.9
where rating = 'Bad';

exec sql disconnect;
/* Implicit commit issued on disconnect */
```

Connect

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The CONNECT statement connects the application to a database and, optionally, to a distributed transaction.

Syntax

The CONNECT statement has the following format:

```
EXEC SQL CONNECT dbname
               [SESSION session_number]
               [IDENTIFIED BY username]
               [OPTIONS = flag {, flag}]
               [WITH options]
```

dbname

Specifies the database to which the session connects. *Dbname* can be a quoted or unquoted string literal or a host string variable. If the name includes any name extensions (such as a system or node name), string literals must be quoted.

SESSION *session_number*

Specifies a positive integer literal or variable whose value must be unique among existing session numbers in the application. A value of 0 is equivalent to omitting the SESSION clause.

IDENTIFIED BY *username*

Specifies the user identifier under which this session runs. *Username* can be specified using a quoted or unquoted string literal or string variable.

Note: Some Enterprise Access products do not support the IDENTIFIED BY clause.

OPTIONS = *flag*

Specifies runtime options for the connection. Valid flags are those accepted by the SQL command. Flags specific to the Terminal Monitor are not valid. For more information about these flags, see the *System Administrator Guide*.

The flags can be specified using quoted or unquoted character string literals or string variables.

The maximum number of flags is twelve.

WITH *options*

Specifies Enterprise Access product specific connection parameters. The command line +c flag provides access to the CONNECT statement's Enterprise Access product WITH clause. For a discussion of the Enterprise Access product with clause, see DBMS Extensions in the chapter "OpenSQL Feature."

For a list of the valid WITH clause options for a specific Enterprise Access product, see your Enterprise Access product guide.

Description

The embedded SQL CONNECT statement connects an application to a database, similar to the operating-system-level SQL and ISQL commands. The CONNECT statement must precede all statements that access the database. The CONNECT statement cannot be issued in a dynamic OpenSQL statement.

Use the SESSION clause if your application includes multiple open sessions (see page 177). The SESSION clause uniquely identifies each session, by associating each session with the specified session_identifier. The session identifier must be a positive integer.

Multiple-session applications require the SESSION clause on each CONNECT statement including the first. If this clause is not present on the first connect in the application, OpenSQL assumes that the application does not use multiple open sessions, and subsequent attempts to open other sessions generate an error.

To switch from one existing session to another existing session, use the SET_SQL statement. The CONNECT statement with the SESSION clause is used only to establish new sessions. You can, however, open more than one session with the same database.

The identified by clause allows the session to run as the specified user, like the -u flag of the SQL command. To determine whether your Enterprise Access product supports the -u flag (and, therefore, the identified by clause), see your Enterprise Access product guide.

The OPTIONS clause allows up to twelve flags to be specified that control session behavior. For details about these flags, see the description of the SQL command in the Command Reference Guide. Not all flags are supported by all Enterprise Access products.

The WITH clause (see page 188) enables Enterprise Access product-specific connection parameters to be specified. To determine the options supported by a specific Enterprise Access product, see your Enterprise Access product guide.

Permissions

This statement is available to all users.

To use the IDENTIFIED BY clause, you must be one of the following:

- The DBA of the specified database
- A user with the SECURITY privilege
- A user that has been granted the DB_ADMIN privilege for the database

Examples: Connect

The following are examples of CONNECT:

1. Connect to the DB2UDB database mydb on virtual node UNIXBOX.

```
exec sql connect 'unixbox::master/db2'
with db2udb = 'audit all';
```

2. Connect to two databases: the Ingres database named personnel, which is located in London, and the local Rdb/VMS database called sales. Set the current session to the personnel database.

```
exec sql connect 'london::personnel' session 1;
exec sql connect 'sales/rdb' session 2;
exec sql set_sql (session = 1);
```

Create Dbevent

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE DBEVENT statement defines a database event.

Syntax

The CREATE DBEVENT statement has the following format:

```
[EXEC SQL] CREATE DBEVENT [schema.]event_name
```

event_name

Identifies the event. The event_name must be a valid object name.

Description

The CREATE DBEVENT statement creates the specified database event. Database events enable an application to pass status information to other applications.

Database events can be registered or raised by any session, provided that the owner has granted the required permission (raise or register) to the session's user, group, or role identifier, or to public. Only the user, group, or role that created a database event can drop that database event.

Embedded Usage

In an embedded CREATE DBEVENT statement, *event_name* cannot be specified using a host language variable. *Event_name* can be specified as the target of a dynamic SQL statement string.

Usage in OpenAPI, ODBC, JDBC, .NET

In ODBC, JDBC, and .NET, events can be created in query statements, but other interfaces must be used to catch the event being created.

Create Index

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE INDEX statement creates an index on an existing table.

Syntax

The CREATE INDEX statement has the following format:

```
[EXEC SQL] CREATE [UNIQUE] INDEX [schema.]index_name
              ON table_name
              (column_name {, column_name})[UNIQUE]
              [WITH with-clause]
```

index_name

Defines the name of the index. This must be a valid object name.

table_name

Specifies the table on which the index is to be created.

column_name

Specifies a list of columns from the table to be included in the index.

UNIQUE

Prevents the index from accepting duplicate values in key fields.

WITH *with-clause*

Specifies Enterprise Access product-specific options. For details,see your Enterprise Access product guide.For an overview of the Enterprise Access product WITH clause, see the chapter "OpenSQL Features."

Description

The CREATE INDEX statement creates an index on an existing base table. The index contains the columns specified and is keyed on those columns, in the order they are specified.

Indexes can improve query processing. If data is retrieved from a table based on an indexed column, the DBMS uses indexes, if available, to accelerate query processing. To obtain the greatest benefit, create indexes that contain all of the columns that are generally queried and keyed on some subset of those columns.

Any number of indexes can be created for a table, but, for portability, each index can contain no more than 16 columns.

To prevent the index from accepting duplicate values in key fields, specify the UNIQUE option. If the base table on which the index is being created has duplicate values for the index's key fields, then the create index statement will fail. Similarly, if you attempt an insert or update that violates the uniqueness constraint of an index created on the table, then the insert or update will fail. This is true for an UPDATE statement that updates multiple rows: the UPDATE statement will fail as soon as it attempts to write a row that update violates the uniqueness constraint.

Particular Enterprise Access products may support extensions to the CREATE INDEX statement (using the WITH clause).

To ensure application portability, follow each CREATE INDEX statement with a COMMIT statement.

An index cannot be updated directly. When a table is changed, the DBMS updates indexes as required. To destroy an index, use the DROP statement. All indexes on a table are destroyed when the table is dropped.

Embedded Usage

The preprocessor does not validate the WITH clause syntax.

Example: Create Index

Create an index called, x, for the columns, ename and age, on employee table.

```
create index x on employee (ename, age);
```

Create Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE TABLE statement creates a base table.

Syntax

The CREATE TABLE statement has the following format:

```
[EXEC SQL] CREATE TABLE table_name
              (column_specification {, column_specification }
              [WITH with_clause]
```

The CREATE TABLE...AS SELECT statement (which creates a table and load rows from another table) has the following format:

```
[EXEC SQL] CREATE TABLE table_name
              [(column_name {, column_name})] AS
                            subselect
                            {UNION [ALL]
                            subselect}
              [WITH with_clause]
```

table_name

Defines the name of the new table, which must be a valid object name (see page 24).

column_specification

Defines the characteristics of the column, as described in Column Specification (see page 217).

AS

Causes the table that you create to be defined and populated by the subselect.

subselect

Specifies a SELECT clause, as described in Using Create Table...As Select (see page 218).

WITH *with_clause*

Specifies Enterprise Access product-specific options. For details,see your Enterprise Access product guide.For an overview of the Enterprise Access product WITH clause, see the chapter "OpenSQL Features."

Description

The CREATE TABLE statement creates a new base table owned by the user who issues the statement. If you use the CREATE TABLE...AS syntax, then the table that you create is a subset of the columns and values in existing tables defined by the subselect.

To ensure application portability, follow every CREATE TABLE statement with a COMMIT statement.

Column Specification—Describe Column Characteristics

The column_specification in a CREATE TABLE statement describes the characteristics of the column.

This statement has the following format:

```
column_name datatype [WITH NULL | NOT NULL]
```

where:

column_name

Assigns a name to the column. It can be any valid OpenSQL name.

datatype

Assigns a valid OpenSQL data type and length to the column. If CREATE TABLE...AS SELECT is specified, the new table takes its column names and formats from the results of the SELECT clause of the *subselect* specified in the AS clause (unless different column names are specified).

WITH NULL | NOT NULL

Specifies whether the column accept nulls during an insert, update, or copy operation. The options are:

WITH NULL

Accepts nulls. The DBMS Server inserts null as the default value if no value is supplied by the user.

This is the default if the clause is omitted.

NOT NULL

Does not accept nulls. The DBMS does not supply a default value. The user must supply a non-null value. (The column is mandatory.)

Using Create Table...As Select

The CREATE TABLE...AS SELECT syntax creates a table from another table or tables. The new table is populated with the set of rows resulting from execution of the specified SELECT statement.

When the CREATE TABLE statement includes an AS clause, specifying column names is optional unless two or more columns of the table would otherwise have the same name.

The column format cannot be specified when using CREATE TABLE...AS SELECT. The formats are copied from the source table columns specified in the subselect clause. The nullability attribute of a column in the new table is the same as the corresponding column in the source table.

Examples: Create Table

1. Create the employee table with columns eno, ename, age, job, salary, and dept.

```
create table employee
  (eno     smallint,
   ename   varchar(20) not null,
   age     integer,
   job     smallint,
   salary  float,
   dept    smallint
   started date);
```

2. Create a table listing employee numbers for employees who make more than the average salary.

```
create table highincome as
  select eno
  from employee
  where salary
   (select avg (salary)
    from employee);
```

Create View

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE VIEW statement defines a virtual table.

Syntax

The CREATE VIEW statement has the following format:

```
[EXEC SQL] CREATE VIEW view_name
               [(column_name {, column_name})]
               AS subselect {UNION [ALL] subselect}
              [WITH with_clause]
```

view_name

Defines the name of the view. It must be a valid object name.

subselect

Specifies a SELECT clause, as described in the Select statement description in this chapter.

WITH *with_clause*

Specifies Enterprise Access product-specific options. For details,see your Enterprise Access product guide.For an overview of the Enterprise Access product WITH clause, see the chapter "OpenSQL Features."

Enterprise Access products that do not support the WITH CHECK OPTION will ignore it when creating the specified view.

Description

The syntax of the CREATE VIEW statement is similar to that of CREATE TABLE...AS SELECT. However, data is not retrieved when a view is created. Instead, the view definition is stored and, when the view_name is later used in an SQL statement, the statement operates on the tables that are used to define the view. (The tables or views used to define a view are called its base tables.)

All selects on views are fully supported. Simply use a view_name in place of a tablename in any SQL retrieval. However, updates, inserts, and deletes on views are subject to several rules:

- The view was created from more than one table.
- The view was created from a non-updatable view.
- Any columns in the view are derived from an expression or aggregate (set) function.
- Additionally, inserts are not allowed if:
  - The view definition contains a where clause and specifies the with check option (if supported by the Enterprise Access product).
  - If any column in the underlying table that was declared as not null not default is not present in the view.
- The ability to update a view depends in part on whether the with check option is specified.

When a view is created WITH CHECK OPTION, columns that are part of the view's qualification cannot be updated.

If the WITH CHECK OPTION is not specified, any row in the view can be updated, even if the update results in a row that is no longer a part of the view.

For example, consider the following two statements:

```
create view v
as select *
from t
where c = 10;
update v
set c = 5;
```

Because the WITH CHECK OPTION is not specified in the view's definition, you are allowed to update column c. If the view had been created WITH CHECK OPTION, the update would not be allowed.

By default, WITH CHECK OPTION is not set.

When a table used in the definition of a view is dropped, the view is also dropped.

Note: Particular Enterprise Access products may support extensions to the CREATE VIEW statement, using the WITH clause.

To ensure application portability, follow every CREATE VIEW statement with a COMMIT statement.

Embedded Usage

In an embedded program, constant expressions can be expressed in the *select_stmt* with host language string variables. If the *select_stmt* includes a WHERE clause, a host language string variable can be used to specify the entire WHERE clause qualification.

Example: Create View

Define a view of employee data including names, salaries, and name of manager.

```
create view empdpt (ename, sal, dname)
  as select employee.name, employee.salary, dept.name
  from employee, dept
  where employee.mgr = dept.mgr;
```

Declare Cursor

Valid in: ESQL, OpenAPI

The DECLARE CURSOR statement associates a cursor name with a SELECT statement.

Syntax

The DECLARE CURSOR statement has the following format:

```
EXEC SQL DECLARE cursor_name CURSOR
              FOR SELECT [ALL | DISTINCT] result_expression {, result_expression}
              FROM table {table}
              [WHERE search_condition]
              [GROUP BY column {, column}]
              [HAVING search_condition]
              [UNION [all] full_select]
              [ORDER BY ordering-expression [ASC | DESC]
                           {, ordering-expression [ASC | DESC]}]
              [FOR UPDATE OF column {, column}]
```

Dynamic SQL form:

```
EXEC SQL DECLARE cursor_name CURSOR
              FOR statement_name;
```

cursor_name

Assigns a name to the cursor. The name can be specified using a quoted or unquoted string literal or a host language string variable. If *cursor_name* is a reserved word, it must be specified in quotes.

Limits: The cursor name cannot exceed 24 characters.

Description

DECLARE CURSOR is a compile-time statement and must appear before the first statement that references the cursor. Despite its declarative nature, a DECLARE CURSOR statement must not be located in a host language variable declaration section. A cursor cannot be declared for repeated select.

A typical cursor-based program performs the following steps:

1. Issue a DECLARE CURSOR statement to associate a cursor with a SELECT statement.
2. Open the cursor. When the cursor is opened, the DBMS Server executes the SELECT statement that was specified in the DECLARE CURSOR statement.
3. Process rows one at a time. The FETCH statement returns one row from the results of the SELECT statement that was executed when the cursor was opened.
4. Close the cursor by issuing the CLOSE statement.

A source file can have multiple cursors, but the same cursor cannot be declared twice. If you want to declare several cursors using the same host language variable to represent cursor_name, it is only necessary to declare the cursor once, since declare cursor is a compile-time statement. Multiple declarations of the same cursor_name will cause a preprocessor error, even if its actual value is to be changed between declarations. For example, the following statements cause a preprocessor error:

```
exec sql declare :cname[i] cursor for s1;
i = i + 1
/* The following statement causes preprocessor error */
exec sql declare :cname[i] cursor for s2;
```

Instead, declare the cursor once. The value assigned to the host language variable cursor_name is not determined until the open cursor statement is executed. For example:

```
exec sql declare :cname[i] cursor for :sname[i];
loop incrementing i
exec sql open :cname[i];
end loop;
```

If a cursor is declared using a host language variable, all subsequent references to that cursor must use the same host language variable. At runtime, a dynamically specified cursor name, that is, a cursor declared using a variable, must be unique among all dynamically specified cursor names in an application. In a similar manner, any cursors referenced in a dynamic statement, for example a dynamic UPDATE or DELETE CURSOR statement, must be unique among all open cursors within the current transaction.

A cursor name declared in one source file cannot be referred to in another file, since the scope of a cursor declaration is the source file. If the cursor is redeclared in another file with the same associated query, it will still not identify the same cursor, not even at runtime. For example, if a cursor c1 is declared in source file, file1, then all references to c1 must be made within file1. Failure to follow this rule results in runtime errors. For example, if you declare cursor c1 in an include file, open it in one file and fetch from it in another file, at runtime the DBMS returns an error indicating that the cursor c1 is not open on the fetch.

This rule applies equally to dynamically specified cursor names. If a dynamic UPDATE or DELETE CURSOR statement is executed, the cursor referenced in the statement must be declared in the same file in which the update or delete statement appears.

The embedded SQL preprocessor does not generate any code for the DECLARE CURSOR statement. Therefore, in a language that does not allow empty control blocks, (for example, COBOL, which does not allow empty IF blocks), the DECLARE CURSOR statement should not be the only statement in the block.

The FOR UPDATE clause must be included if there is any possibility that the cursor will be used to update rows. List any column that might be updated. If you only intend to delete rows, then the FOR UPDATE clause is not required. The actual updating or deleting takes place with the cursor version of the UPDATE or DELETE statement, respectively.

A cursor cannot be declared for updating if its SELECT statement:

- Refers to more than one table.

  For example, the following cursor declaration causes a compile-time error, and is illegal because two tables are used in the SELECT statement:

```
/* illegal join on different tables for update */
exec sql declare c1 cursor for
       select employee.id, accounts.sal
       from employee, accounts
       where employee.salno = accounts.accno
       for update of sal;
```

- Refers to a non-updatable view.

  In the following example, if empdept is a read-only view, the following code generates a runtime error when the OPEN statement is executed. No preprocessor error is generated, because the preprocessor does not know that empdept is a view.

  ```
  /* empdept is a read-only view */
  exec sql declare c2 cursor for
         select name, deptinfo
         from empdept
         for update of deptinfo;

  exec sql open c2;
  ```

- Includes a DISTINCT, GROUP BY, HAVING, ORDER BY, or UNION clause.
- Includes a column that is a constant or is based on a calculation.

  For example, the following cursor declaration causes an error when attempting to update the column named constant:

  ```
  /* "constant" cannot be declared for update */
  exec sql declare c3 cursor for
         select constant = 123, ename
         from employee
         for update of constant;
  ```

  If an updatable column has been assigned a result column name using the syntax:

  column_name AS *result_name*

  the column referred to in the FOR UPDATE list must see the table column name, and not the result column name.

Updates associated with a cursor take effect on the underlying table when the statement is executed. The effects of the updates can be seen by the program before the cursor is closed. The actual committal of the changes does not override or interfere with COMMIT or ROLLBACK statements that may be executed subsequently in the program. Because changes take effect immediately, avoid updating keys that cause the current row to move "forward" with respect to the current position of the cursor, because this may result in fetching the same row more than once.

If the FOR UPDATE clause is specified, the cursor can still be opened for reading only. The OPEN statement accepts the optional FOR READONLY clause, which specifies that, though the cursor may have been declared for update, the cursor is not being opened for update. Including this clause in the OPEN statement can improve the performance of the cursor retrieval.

Not all database management systems allow the use of a cursor to update a row more than once. For details, see your Enterprise Access product and DBMS-specific documentation for details.

The UNION form of the SELECT statement can be used in a cursor declaration. To select all columns, use SELECT *. Each column does not need to be listed individually.

When the ORDER BY clause is specified, the ordering is performed according to SQL comparison rules. Each column specified in the ordering must specify either a column name, which identifies a column of the result table, or an integer, which identifies a numbered column of the result table. A named result column can be identified by an ordering name or a number. An unnamed result column must be identified by an ordering number.

Host language variables can be used in the SELECT statement of a declare cursor to substitute for expressions in the SELECT clause or in the search condition. When the search condition is specified within a single string variable (as when the query is constructed using the form system query mode) then all the following clauses, such as the ORDER BY or UPDATE clause, can be included within the variable. These variables must be valid at the time of the cursor's open statement, because that is when the select is actually evaluated—they need not have defined values at the time of the DECLARE CURSOR statement. Host language variables cannot substitute for any table or column names.

You can also use the dynamic OpenSQL syntax and specify a prepared statement name instead of a SELECT statement. The statement name must identify a SELECT statement that has been prepared previously. The statement name must not be the same as another prepared statement name that is associated with a currently open cursor.

Usage in OpenAPI

In OpenAPI, Declare Cursor functionality is achieved through query parameters to the IIapi_query() function.

Examples: Declare Cursor

The following are DECLARE CURSOR statement examples:

1. Declare a cursor for a retrieval of employees from the shoe department, ordered by name (ascending) and salary (descending). (This can also be specified as a select loop.)

```
EXEC SQL DECLARE cursor1 CURSOR FOR
    SELECT ename, sal
    FROM employee
    WHERE dept = 'shoes'
    ORDER BY 1 ASC, 2 DESC;
```

2. Declare a cursor for updating the salaries and departments of employees currently in the shoe department.

```
EXEC SQL DECLARE cursor2 CURSOR FOR
    SELECT ename, sal
    FROM employee
    WHERE dept = 'shoes'
    FOR UPDATE OF sal, dept;
```

3. Declare a cursor for updating the salaries of employees whose last names are alphabetically like a given pattern.

```
searchpattern = 'a%';
EXEC SQL DECLARE cursor3 CURSOR FOR
    SELECT ename, sal
    FROM employee
    WHERE ename LIKE :searchpattern
    FOR UPDATE OF sal;
...
EXEC SQL OPEN cursor3;
```

   In the above example, the variable, searchpattern, must be a valid declaration in the host language at the time the statement OPEN CURSOR3 is executed. It also must be a valid embedded SQL declaration at the point where the cursor is declared.

4. Declare a cursor to print the results of a retrieval for runtime viewing and salary changes.

```
EXEC SQL DECLARE cursor4 CURSOR FOR
    SELECT ename, age, eno, sal
    FROM employee
    FOR DIRECT UPDATE OF sal;

EXEC SQL WHENEVER sqlerror stop;
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL OPEN cursor4;

loop /* loop is broken when NOT FOUND becomes true. */
EXEC SQL FETCH cursor4
        INTO :name, :age, :idno, :salary;
    PRINT name, age, idno, salary;
    PRINT 'New salary';
    READ newsal;
    IF (newsal > 0 AND newsal <> salary) THEN
        EXEC SQL UPDATE employee
            SET sal = :newsal
            WHERE CURRENT OF cursor4;
    END IF;
END LOOP;
close_cursor:
  EXEC SQL CLOSE cursor4;
```

5. Declare a cursor for retrieval of specific data. The FOR UPDATE clause refers to column name, sal, and not, res.

```
EXEC SQL DECLARE cursor5 CURSOR FOR
    SELECT ename, sal AS res
    FROM employee
    WHERE eno BETWEEN :eno_low AND :eno_high
    FOR UPDATE OF sal;
.  .  .

loop while more input
    READ eno_low, eno_high;

EXEC SQL OPEN cursor5;

print and process rows;
END LOOP;
```

6. Declare two cursors for the department and employee tables, and open them in a master-detail fashion.

```
EXEC SQL DECLARE master_cursor CURSOR FOR
    SELECT * FROM dept
    ORDER BY dno;

EXEC SQL DECLARE detail_cursor CURSOR FOR
    SELECT * FROM employee
    WHERE edept = :dno
    ORDER BY ename;

EXEC SQL OPEN master_cursor;

loop while more department

EXEC SQL FETCH master_cursor
    INTO :dname, :dno, :dfloor, :dsales;

if not found break loop;

/*
    ** For each department retrieve all the
    ** employees and display the department
    ** and employee data.
*/

EXEC SQL OPEN detail_cursor;

loop while more employees

EXEC SQL FETCH detail_cursor
    INTO :name, :age, :idno, :salary, :edept;
    /*
    ** For each department retrieve all the
    ** employees and display the department
    ** and employee data.
    */

process and display data;

END LOOP;
    EXEC SQL CLOSE detail_cursor;
END LOOP;

EXEC SQL CLOSE master_cursor;
```

7. Declare a cursor that is a union of three tables with identical typed columns (the columns have different names). As each row returns, record the information and add it to a new table. Ignore all errors.

```
EXEC SQL DECLARE shapes CURSOR FOR
    SELECT boxname, box# FROM boxes
    WHERE boxid > 100
    UNION
    SELECT toolname, tool# FROM tools
    UNION
    SELECT nailname, nail# FROM nails
    WHERE nailweight > 4;

EXEC SQL OPEN shapes;
EXEC SQL WHENEVER NOT FOUND GOTO done;

loop while more shapes

EXEC SQL FETCH shapes INTO :name, :number;
    record name and number;
    EXEC SQL INSERT INTO hardware
    (:name, :number);

END LOOP;

done:

EXEC SQL CLOSE shapes;
```

Declare Global Temporary Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DECLARE GLOBAL TEMPORARY TABLE statement creates a temporary table.

Syntax

The DECLARE GLOBAL TEMPORARY TABLE statement has the following format:

```
[EXEC SQL] DECLARE GLOBAL TEMPORARY TABLE [SESSION.]table_name
              (column_name format {, column_name format})
              ON COMMIT PRESERVE ROWS
              WITH NORECOVERY
```

To create a temporary table by selecting data from another table:

```
[EXEC SQL] DECLARE GLOBAL TEMPORARY TABLE [SESSION.]table_name
              (column_name {, column_name})
              AS subselect
              ON COMMIT PRESERVE ROWS
              WITH NORECOVERY
```

table_name

Defines the name of the temporary table.

ON COMMIT PRESERVE ROWS

(Required) Directs the DBMS Server to retain the contents of a temporary table when a COMMIT statement is issued.

WITH NORECOVERY

(Required) Suspends logging for the temporary table.

AS *subselect*

Defines the subselect, as described in Select (interactive) (see page 290).

Description

The DECLARE GLOBAL TEMPORARY TABLE statement creates a temporary table, also referred to as a *session-scope* table. Temporary tables are useful in applications that need to manipulate intermediate results and want to minimize the processing overhead associated with creating tables.

Embedded Usage

Do not specify the DECLARE GLOBAL TEMPORARY TABLE SESSION statement within the DECLARE section of an embedded program. Place the statement in the body of the embedded program.

Restrictions

Temporary tables may be subject to the following restrictions:

- Temporary tables cannot be used within database procedures.
- Temporary tables cannot be used in view definitions.
- The following SQL statements cannot be used with temporary tables:
  - CREATE INDEX
  - CREATE VIEW
  - GRANT
  - HELP
  - REVOKE
  - SET LOCKMODE
- Repeat queries referencing temporary tables cannot be shared between sessions.

The following commands cannot be issued with a temporary table name:

- auditdb
- copydb
- optimizedb
- statdump
- verifydb

Related Statements

Create Table

Delete

Drop

Insert

Select (interactive)

Update

Examples: Declare Global Temporary Table

The following are DECLARE GLOBAL TEMPORARY TABLE statement examples:

1. Create a temporary table.

```
exec sql declare global temporary table
 session.emps
  (name char(20) , empno char(5))
  on commit preserve rows
 with norecovery;
```

2. Use a subselect to create a temporary table containing the names and employee numbers of the highest-rated employees.

```
exec sql declare global temporary table
    session.emps_to_promote
    as select name, empno from employees
    where rating >= 9
    on commit preserve rows
    with norecovery
```

Declare Statement

Valid in: ESQL

The DECLARE statement lists one or more names that are used in a program to identify dynamic OpenSQL prepared statements.

The declaration of prepared statement names is not required; DECLARE statement is a comment statement, used for documentation in an embedded SQL program. No syntactic elements can be represented by host language variables.

The embedded SQL preprocessor does not generate any code for DECLARE statement. Therefore, in a language that does not allow empty control blocks (for example, COBOL, which does not allow empty IF blocks), this statement must not be the only statement in the block.

Syntax

The DECLARE statement has the following format:

```
EXEC SQL DECLARE statement_name {, statement_name) STATEMENT
```

Example: Declare Statement

The following example declares one statement name for a dynamic statement that is executed ten times:

```
  EXEC SQL DECLARE ten_times STATEMENT;

  loop while more input
       print
        'Type in statement to be executed 10 times?';
       read statement_buffer;

  EXEC SQL PREPARE ten_times
       FROM :statement_buffer;
  loop 10 times
       EXEC SQL EXECUTE ten_times;
  end loop;
end loop;
```

Declare Table

Valid in: ESQL

The DECLARE TABLE statement describes the characteristics of a database table.

Syntax

The DECLARE TABLE statement has the following format:

```
EXEC SQL DECLARE table_name TABLE
              (column_name data_type [WITH NULL | NOT NULL]
              {, column_name data_type})
```

Description

The DECLARE TABLE statement lists the columns and data types associated with a database table, for the purpose of program documentation. The DECLARE TABLE statement is a comment statement inside a variable declaration section and is not an executable statement. You cannot use host language variables in this statement.

The dclgen utility includes a DECLARE TABLE statement in the file it generates while creating a structure corresponding to a database table. For details, see the *Embedded SQL Companion Guide*.

The embedded SQL preprocessor does not generate any code for the DECLARE TABLE statement. Therefore, in a language that does not allow empty control blocks (for example, COBOL, which does not allow empty IF blocks), the DECLARE TABLE statement must not be the only statement in the block.

Example: Declare Table

The following is a DECLARE TABLE statement example for a database table:

```
EXEC SQL DECLARE employee TABLE
       (eno    INTEGER2 NOT NULL,
       ename   CHAR(20) NOT NULL,
       age     INTEGER1,
       job     INTEGER2,
       sal     FLOAT4,
       dept    INTEGER2 NOT NULL);
```

Delete

Valid in: SQL, ESQL

The DELETE statement deletes rows from the specified table that satisfy the *search_condition* in the WHERE clause. If the WHERE clause is omitted, the statement deletes all rows in the table. The result is a valid but empty table.

If the WHERE clause includes a subselect, the tables specified in the subselect cannot include the table from which you are deleting rows.

Syntax

The DELETE statement has the following formats:

Interactive version:

```
[EXEC SQL] DELETE FROM table_name
           [WHERE search_condition];
```

Embedded non-cursor version:

```
[EXEC SQL] [REPEATED] DELETE FROM table_name
           [WHERE search_condition];
```

Embedded cursor version:

```
[EXEC SQL] DELETE FROM table_name
           WHERE CURRENT OF cursor_name;
```

table_name

Specifies the table for which the constraint is defined.

Embedded Usage

There are two embedded versions of the DELETE statement: one deletes rows according to the search criteria specified in its WHERE clause, and the second deletes the row to which the specified cursor is positioned.

Non-Cursor Delete

The non-cursor version of the embedded OpenSQL DELETE statement is identical to the interactive delete. Host language variables can be used to represent constant expressions in the *search_condition* but they cannot specify names of database columns or include any operators. The complete search condition can be specified using a host string variable.

To reduce the overhead required to execute a (non-cursor) delete repeatedly, specify the keyword REPEATED. The REPEATED keyword directs the OpenSQL to encode the delete and save its execution plan the first time the statement is executed, thereby improving subsequent executions of the same delete. The REPEATED keyword is valid for non-cursor deletes only and is ignored if used with the cursor version of the statement. The repeated delete cannot be specified as a dynamic OpenSQL statement.

If the *search_condition* is dynamically constructed and the *search_condition* is changed after initial execution of the statement, the REPEATED option cannot be specified. The saved execution plan is based on the initial values in the *search_condition* and changes are ignored. This rule does not apply to simple variables used in *search_conditions*.

Cursor Delete

The cursor version immediately deletes the row to which the specified cursor is pointing. If the cursor is not currently pointing at a row when the delete is executed, then the DBMS generates an error indicating the need to issue a FETCH statement to position the cursor on a row. (After a deletion, the cursor points to a position after the deleted row, but before the next row, if any.)

The COMMIT and ROLLBACK statements close all open cursors. A common programming error is to delete the current row of a cursor, commit the change, and then loop to repeat the process. This process fails because the first commit closes the cursor.

In performing a cursor delete, make sure that certain conditions are met:

- A cursor must be declared in the same file in which any DELETE statements referencing that cursor appear. This applies also to any cursors referenced in dynamic DELETE statement strings.
- A cursor name in a dynamic DELETE statement must be unique among all open cursors in the current transaction.
- The cursor stipulated in the delete must be open before the statement is executed.
- The SELECT statement of the cursor must not contain a DISTINCT, GROUP BY, HAVING, ORDER BY, or UNION clause.
- The FROM clause of the delete and the FROM clause in the cursor's declaration must refer to the same database table.

The cursor name can be specified with a string constant or a host language variable.

If the statement does not delete any rows, the sqlcode variable in the SQLCA structure is set to 100.

The sqlerrd(3) variable in the SQLCA structure contains the number of rows deleted.

Example: Delete

The following example removes all employees who make over $35,000:

```
DELETE FROM employee WHERE salary > 35000;
```

Describe

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The DESCRIBE statement retrieves information about the result of a prepared dynamic SQL statement.

Syntax

The DESCRIBE statement has the following format:

```
EXEC SQL DESCRIBE statement_name  INTO|USING [:]descriptor_name [USING NAMES];
```

statement_name

Specifies a valid prepared statement. Specify the statement_name using a string literal or a host language string variable. If an error occurs when the specified statement is prepared, the statement is not valid. If a COMMIT or ROLLBACK statement is executed after the statement is prepared and before it is executed, the statement is discarded and cannot be described or executed.

descriptor name

Identifies an SQLDA. The descriptor name can be SQLDA or any other valid object name defined by the program when the structure is allocated. Because the SQLDA is not declared in a declaration section, the preprocessor does not verify that *descriptor_name* represents an SQLDA structure. If *descriptor_name* does not represent an SQLDA structure, undefined errors occur at runtime.

Descriptor_name can be preceded by a colon (:).

USING NAMES

Returns the names of result columns in the descriptor if the described statement is a SELECT statement. (The USING NAMES clause is optional and has no effect on the results of the DESCRIBE statement.)

Description

The DESCRIBE statement returns the data type, length, and name of the result columns of the prepared select. If the prepared statement is not a SELECT, describe returns a zero in the SQLDA sqld field.

The DESCRIBE statement cannot be issued until after the program allocates the SQLDA and sets the value of the SQLDA's sqln field to the number of elements in the SQLDA's sqlvar array. The results of the DESCRIBE statement are complete and valid only if the number of the result columns (from the SELECT) is less than or equal to the number of allocated sqlvar elements. (The maximum number of result columns that can be returned is 1024.)

The PREPARE statement can also be used with the INTO clause to retrieve the same descriptive information provided by describe.

Direct Execute Immediate

The DIRECT EXECUTE IMMEDIATE statement sends DBMS-specific commands to the DBMS without translation.

Syntax

The DIRECT EXECUTE IMMEDIATE statement has the following format:

```
EXEC SQL DIRECT EXECUTE IMMEDIATE string | string_variable
```

Description

The DIRECT EXECUTE IMMEDIATE statement allows statements to be sent to the Enterprise Access product or DBMS to which a session is connected. The Enterprise Access product does not translate the statement. If the statement is not supported by the DBMS or Enterprise Access product, an error is returned. The DIRECT EXECUTE IMMEDIATE statement cannot be used to return rows to a session.

A host language variable or string literal can be used to specify the statement. If you use a string literal, avoid embedding quotes in the literal. If you specify the statement using a host language variable, the OpenSQL string-delimiting conventions must be observed.

Disconnect

Valid in: ESQL

Terminates access to the database.

Syntax

The DISCONNECT statement has the following format:

```
EXEC SQL DISCONNECT [SESSION session_identifier | ALL]
```

session_identifier

Disconnects the specified session in a multi-session application. The *session_identifier* must be a positive integer constant or variable containing the session identifier. To determine the *session_identifier* for the current session, use the INQUIRE_SQL(:*session_id* = SESSION) statement. If an invalid session is specified, OpenSQL issues an error and does not disconnect the session.

ALL

Disconnects all open sessions.

Description

The DISCONNECT statement terminates access to the database, closes any open cursors, and commits any open transactions.

To disconnect the current session, issue the DISCONNECT statement, omitting the session identifier. Other sessions (if any) will remain connected. To switch sessions, use the SET_SQL statement.

Examples: Disconnect

1. Disconnect from the current database.

   ```
   exec sql disconnect;
   ```

2. On an error, roll back pending updates, then disconnect the database session.

   ```
   exec sql whenever sqlerror goto err;
   ...
   err:
      exec sql rollback;
      exec sql disconnect;
   ```

Drop

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP statement destroys one or more tables, indexes, or views.

Syntax

The DROP statement has the following format:

```
[EXEC SQL] DROP objecttype objectname [WITH with_clause]
```

objecttype

Specifies the type of object, which can be one of the following keywords:

TABLE

VIEW

INDEX

objectname

Specifies the name of a table, view, or index.

WITH *with_clause*

Specifies a comma-separated list of valid Enterprise Access product WITH clause options. For an overview of the Enterprise Access product WITH clause, see the chapter "OpenSQL Features." For a list of the valid WITH clause options for a specific Enterprise Access product, see the product guide.

Description

The DROP statement removes the specified tables, indexes, and views from the database. When a table is dropped, any indexes, views, or privileges defined on that table are automatically dropped also. When a view is dropped, all associated privileges and dependent views are dropped.

To ensure application portability, follow every DROP statement with a COMMIT statement.

Embedded Usage

You cannot replace any portions of the statement with host language variables.

Examples: Drop

1. Drop an index named, tindex.

```
drop index tindex;
commit;
```

2. Drop a base table and all related views, indexes, and permissions.

```
drop table employee;
commit;
```

3. In an embedded program, drop a view.

```
exec sql drop view tempview;
exec sql commit;
```

Drop Dbevent

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP DBEVENT statement drops the specified database event.

Syntax

The DROP DBEVENT statement has the following format:

```
[EXEC SQL] DROP DBEVENT [schema.]event_name;
```

Description

The DROP DBEVENT statement drops the specified database event. If applications are currently registered to receive the database event, the registrations are not dropped. If the database event was raised prior to being dropped, the database event notifications remain queued, and applications can receive them using the GET DBEVENT statement.

For a full description of database events, see Database Events (see page 191).

Embedded Usage

In an embedded DROP DBEVENT statement, *event_name* cannot be specified using a host string variable. *Event_name* can be specified as the target of a dynamic SQL statement string.

End Declare Section

Valid in: ESQL

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

Syntax

The END DECLARE SECTION statement has the following format:

```
EXEC SQL END DECLARE SECTION;
```

Description

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

A host language variable declaration section contains declarations of host language variables for use in an embedded OpenSQL program. The BEGIN DECLARE SECTION statement starts each variable declaration section.

Endselect

Valid in: ESQL

The ENDSELECT statement terminates a SELECT loop.

Syntax

The ENDSELECT statement has the following format:

```
EXEC SQL ENDSELECT;
```

Description

The ENDSELECT statement terminates embedded OpenSQL select loops. A select loop is a block of code delimited by begin and end statements and associated with a select statement. As the select statement retrieves rows from the database, each row is processed by the code in the select loop. (For more information about select loops, see Select (interactive) (see page 290) in this chapter.) When the ENDSELECT statement is executed, the program stops retrieving rows from the database and program control is transferred to the first statement following the select loop.

The ENDSELECT statement must be inside the select loop that it is intended to terminate. If an ENDSELECT statement is placed inside a forms statement code block that is syntactically nested within a select loop, the statement ends the nested construct as well as the select loop.

The statement must be terminated according to the rules of the host language.

To find out how many rows were retrieved before the ENDSELECT statement was issued, check the sqlerrd(3) variable of the SQLCA.

Example: Endselect

The following example breaks out of a select loop on a data loading error:

```
exec sql select ename, eno into :ename, :eno
        from employee;
exec sql begin;
        load ename, eno into data set;
        if error then
        print 'Error loading ', ename, eno;
        exec sql endselect;
        end if
exec sql end;
/* endselect transfers control to here */
```

Execute

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The EXECUTE statement executes a previously prepared dynamic OpenSQL statement.

Syntax

The EXECUTE statement has the following format:

```
exec sql EXECUTE statement_name
              [USING variable {, variable} | USING DESCRIPTOR descriptor_name];
```

statement_name

Identifies a valid object name specified using a a string literal or a host language variable. It must identify a valid prepared statement. If the statement identified by *statement_name* is invalid, the Enterprise Access product or server issues an error and aborts the execute statement. (A prepared statement is invalid if a transaction was committed or rolled back after the statement was prepared or if an error occurred while preparing the statement.)

Description

The EXECUTE statement executes the prepared statement specified by *statement_name*. EXECUTE can be used to execute any statement that can be prepared, except the SELECT statement.

Note: To execute a prepared SELECT statement, use the EXECUTE IMMEDIATE statement. For more information, see Prepare and Execute Statements (see page 142) and the Execute Immediate Statement (see page 141).

If the prepared statement refers to a cursor update or delete and the associated cursor is not open, the Enterprise Access product or server issues an error. For more information, see Update (see page 317) and Delete in this chapter.

If the prepared statement uses a question mark (?) to specify one or more constant expressions, the USING clause must be specified in the statement. If you know the number and data types of the expressions specified by question marks in the prepared statement, use the using *variable* {, *variable*} option. The number of the variables listed must correspond to the number of question marks in the prepared statement, and each variable's data type must be compatible with its usage in the prepared statement.

The following example prepares a statement containing one question mark from a buffer and executes it using a host language variable:

```
statement_buffer =
'delete from ' + table_name + ' where code = ?';
exec sql prepare del_stmt from :statement_buffer;
...

exec sql execute del_stmt using :code;
```

The value in the variable, code, replaces the ? in the where clause of the prepared delete statement.

If the number and data types of the prepared statement parameters are not known until runtime, use the using descriptor option. In this alternative, the *descriptor_name* identifies an SQLDA, a host language structure that must be allocated prior to its use. The SQLDA includes the sqlvar array. Each element of sqlvar is used to describe and point to a host language variable. The execute statement uses the values placed in the variables pointed to by the sqlvar elements to execute the prepared statement.

When the SQLDA is used for input, the program must set the sqlvar array element type, length, and data area for each portion of the prepared statement specified by question marks, prior to executing the statement.

Here are some of the ways the program can supply that information:

- When preparing the statement, the program can request all type and length information from the interactive user.
- Before preparing the statement, the program can scan the statement string, and build a select statement out of the clauses that include parameters. The program can then prepare and describe this select statement to collect data type information to be used on input.
- If another application development tool is being used to build the dynamic statements (such as a Vision frame or a VIFRED form), the data type information included in those objects can be used to build the descriptor. An example of this method is shown in the Examples section.

In addition, the program must correctly set the sqld field in the SQLDA structure. For a complete description of the structure of the SQLDA and how to use it, see the chapter "Dynamic OpenSQL."

The variables used by the USING clause can be associated with indicator variables if indicator variables are permitted with the same statement in the non-dynamic case. For example, because indicator variables are permitted in the INSERT statement VALUES clause, then the following dynamically defined INSERT statement can include indicator variables (name_ind and age_ind) in the EXECUTE statement:

```
statement_buffer = 'insert into employee (name, age)      values (?, ?)';
exec sql prepare s1 from :statement_buffer;
exec sql execute s1 using :name:name_ind,
           :age:age_ind;
```

However, a host structure variable cannot be used in the USING clause, even if the named statement refers to a statement which allows a host structure variable when issued non-dynamically.

This statement must be terminated according to the rules of the host language.

Examples: Execute

1. Although the COMMIT statement can be prepared, once the statement is executed, the prepared statement becomes invalid. For example, the following code causes an error on the second execute statement:

```
statement_buffer = 'commit';
exec sql prepare s1 from :statement_buffer;
process and update data;
exec sql execute s1;  /* Once committed, 's1' is lost */
process and update more data;
exec sql execute s1;
/* 's1' is NOT a valid statement name */
```

2. When leaving an application, each user deletes all their rows from a working table. User rows are identified by their different access codes. One user may have more than one access code.

```
read group id from terminal;
statement_buffer = 'delete from ' + group_id + '
where access_code =   ?';

exec sql prepare s2 from :statement_buffer;

read access_code from terminal;
loop while (access_code <> 0)

  exec sql execute s2 using :access_code;
    read access_code from terminal;

end loop;
exec sql commit;
```

3. This example uses the OpenSQL forms system and Dynamic OpenSQL. The program reads the forms descriptions using the formdata statement and then uses that information to fill an input SQLDA for a variety of statements. For details about forms programming, see the *Forms-based Application Development Tools User Guide.*

   In preparation, the program must allocate a large local SQLDA, called local_sqlda. At the start of form display, the program must retrieve descriptive information into the local SQLDA. The form name is only known at runtime through a command line flag.

```
exec frs formdata :form_name;
exec frs begin;

Using inquire_frs statements, retrieve the type
and length information from the form and fill the
corresponding element in the sqlvar. For each
field  on the form set the sqltype, sqllen and
sqldata fields. If the type is negative (nullable)
set the sqlind field too.

Build 3 dynamic statements into 3 statement
buffers to execute the insert, update and delete
operations, using the field names returned by inquire_frs.

exec frs end;
```

At this point, the program has built a SQLDA that it will use for input, and three statement buffers, each with a full list of field names and parameter markers. The insert statement buffer, insert_buffer, may look like:

```
'insert into table1 (field1, field2) values (?, ?)'
```

while the delete statement buffer, delete_buffer, may look like:

```
'delete from table1 where field1 = ? and field2 = ?'
```

Now prepare the statements:

```
  exec sql prepare insert_stmt from :insert_buffer;
  exec sql prepare update_stmt from :update_buffer;
  exec sql prepare delete_stmt from :delete_buffer;
```

Run the form allowing the user to enter data and execute an operation. Supply the menu items, Insert, Update, and Delete, as well as others. For example:

```
...

exec frs activate menuitem 'Insert';
exec frs begin;
Get values from the form and point the sqldata and
sqlind fields of local_sqlda to those values;
exec sql execute insert_stmt
using descriptor :local_sqlda;
exec frs end;

...
```

Execute Immediate

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The EXECUTE IMMEDIATE statement executes an SQL statement specified as a string literal or in a host language variable.

Syntax

The EXECUTE IMMEDIATE statement has the following format:

```
EXEC SQL EXECUTE IMMEDIATE statement_string
              [INTO variable {, variable} | USING [DESCRIPTOR] descriptor_name
              [EXEC SQL BEGIN;
                           program_code
               EXEC SQL END;]]
```

Description

The EXECUTE IMMEDIATE statement executes a dynamically built statement string. Unlike the prepare and execute sequence, this statement does not name or encode the statement and cannot supply parameters. The EXECUTE IMMEDIATE statement is equivalent to:

```
exec sql prepare statement_name from
        :statement_buffer;
exec sql execute statement_name;
'Forget' the statement_name;
```

EXECUTE IMMEDIATE can be used:

- To execute a dynamic statement once in your program
- To execute a dynamic select statement and process the result rows with a select loop

If you intend to execute the statement string repeatedly and it is not a SELECT statement, use the PREPARE and EXECUTE statements instead. For more information about the alternatives available for executing dynamic statements, see the chapter "Dynamic OpenSQL.” If the statement string is blank or empty, OpenSQL returns a runtime syntax error.

The EXECUTE IMMEDIATE statement must be terminated according to the rules of the host language.

The following OpenSQL statements cannot be executed using EXECUTE IMMEDIATE:

call	disconnect	inquire_sql
close	endselect	open
connect	execute	prepare
declare	fetch	set
describe	help	set_sql
direct execute immediate	include	whenever

The statement string must not include EXEC SQL, any host language terminators, or references to variable names. If your statement string includes embedded quotes, it is easiest to specify the string in a host language variable. If you choose to specify a string that includes quotes as a string constant, remember that quoted characters *within* the statement string must follow the OpenSQL string delimiting rules. Even if your host language delimits strings with double quotes, the quoted characters within the statement string must be delimited by single quotes. For complete information about embedding quotes within a string literal, see the *Embedded SQL Companion Guide*.

If the statement string is a cursor update or cursor delete, the declaration of the named cursor must appear in the same file as the EXECUTE IMMEDIATE statement executing the statement string.

The INTO or USING clause can only be used when the statement string is a SELECT statement. The INTO clause specifies variables to store the values returned by a select. This option can be used if the program knows the data types and lengths of the result columns before the select executes. The *variables* must be type compatible with the associated result columns. For information about the compatibility of host language variables and OpenSQL data types, see the *Embedded SQL Companion Guide*.

Include the USING clause if the program does not know the types and lengths of the result columns until runtime. The USING clause specifies an SQL Descriptor Area (SQLDA), a host language structure having, among other fields, an array of sqlvar elements. Each sqlvar element describes and points to a host language variable. When the USING clause is specified, OpenSQL places the result column values in the variables pointed at by the sqlvar elements.

If you intend to use the USING clause, the program can first prepare and describe the SELECT statement. This process returns data type, name, and length information about the result columns to the SQLDA. Your program can then use that information to allocate the necessary variables before executing the select. For more information and about executing dynamic SELECT statements and some examples of executing a dynamic select, see the chapter "Dynamic OpenSQL.”

If the SELECT statement will return more than one row, include the BEGIN/END statement block. This block defines a select loop. OpenSQL processes each row that the select returns using the program code that you supply in the select loop. The program code inside the loop must not include any other database statements, except the ENDSELECT statement. If the select returns multiple rows and you do not supply a select loop, the application receives only the first row and an error to indicate that others were returned but unseen.

Example: Execute Immediate

This example reads an SQL statement from the terminal into a host string variable, statement_buffer. If the user enters quit, the program ends. If an error occurs, the program informs the user.

```
exec sql include sqlca;

read statement_buffer from terminal;
loop while (statement_buffer <> 'QUIT')

exec sql execute immediate :statement_buffer;
  if (sqlcode = 0) then
    exec sql commit;
  else if (sqlcode = 100) then
    print 'No qualifying rows for statement:';
    print statement_buffer;
  else
   print 'Error  :', sqlcode;
   print 'Statement :', statement_buffer;
  end if;

  read statement_buffer from terminal;
end loop;
```

Execute Procedure

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The EXECUTE PROCEDURE statement invokes a database procedure.

Syntax

The EXECUTE PROCEDURE statement has the following formats:

Non-dynamic version:

```
[EXEC SQL] EXECUTE PROCEDURE [schema.]proc_name
              [(param_name=param_spec {,param_name= param_spec})] |
              [RESULT ROW (variable [:indicator_var]
                                        {, variable[:indicator_var]})]
              [INTO return_status]
              [EXEC SQL BEGIN;program code;
              EXEC SQL END;]
```

Dynamic version:

```
[EXEC SQL] EXECUTE PROCEDURE [schema.]proc_name
              [USING [DESCRIPTOR] descriptor_name]
              [INTO return_status]
```

proc_name

Specifies the name of the procedure, using a literal or a host string variable.

param_spec

Is a literal value, a host language variable containing the value to be passed (**:***hostvar*), or a host language variable passed by reference (BYREF(:*host_variable*)).

Description

The EXECUTE PROCEDURE statement executes a specified database procedure.

Database procedures can be executed from interactive SQL (the Terminal Monitor), an embedded SQL program, or from another database procedure.

This statement can be executed dynamically or non-dynamically. When executing a database procedure, you typically provide values for the formal parameters specified in the definition of the procedure.

If an EXECUTE PROCEDURE statement includes a RESULT ROW clause, it can only be executed non-dynamically.

Passing Parameters - Non-Dynamic Version

In the non-dynamic version of the EXECUTE PROCEDURE statement, parameters can be passed by *value* or by *reference*.

By value - To pass a parameter by value, use this syntax:

```
param_name = value
```

When passing parameters by value, the database procedure receives a copy of the value. *Values* can be specified using:

- Numeric or string literals
- SQL constants (such as TODAY or USER)
- Host language variables
- Arithmetic expressions

The data type of the value assigned to a parameter must be compatible with the data type of the corresponding parameter in the procedure definition. Specify date data using quoted character string values, and money using character strings or numbers. If the data types are not compatible, an error is issued and the procedure not executed.

By reference - To pass a parameter by reference, use this syntax:

```
param_name = BYREF(:host_variable)
```

When passing parameters by reference, the database procedure can change the contents of the variable. Any changes made by the database procedure are visible to the calling program. Parameters cannot be passed by reference in interactive SQL.

Each *param_name* must match one of the parameter names in the parameter list of the definition of the procedure. *Param_name* must be a valid object name, and can be specified using a quoted or unquoted string or a host language variable.

Passing Parameters - Dynamic Version

In the dynamic version of the EXECUTE PROCEDURE statement, the *descriptor_name* specified in the USING clause identifies an SQL Descriptor Area (SQLDA), a host language structure allocated at runtime.

Prior to issuing the EXECUTE PROCEDURE statement, the program must place the parameter names in the sqlname fields of the SQLDA sqlvar elements and the values assigned to the parameters must be placed in the host language variables pointed to by the sqldata fields. When the statement is executed, the USING clause ensures those parameter names and values to be used.

Parameter names and values follow the same rules for use and behavior when specified dynamically as those specified non-dynamically. For example, because positional referencing is not allowed when you issue the statement non-dynamically, when you use the dynamic version, any sqlvar element representing a parameter must have entries for both its sqlname and sqldata fields. Also, the names must match those in the definition of the procedure and the data types of the values must be compatible with the parameter to which they are assigned.

Any parameter in the definition of the procedure that is not assigned an explicit value when the procedure is executed is assigned a null or default value. If the parameter is not nullable and does not have a default, an error is issued.

For example, for the CREATE statement

```
create procedure p (i integer not null,
        d date, c varchar(100)) as ...
```

the following associated EXECUTE PROCEDURE statement implicitly assigns a null to parameter *d*.

```
exec sql execute procedure p (i = 123,
        c = 'String');
```

When executing a procedure dynamically, set the SQLDA sqld field to the number of parameters that you are passing to the procedure. The sqld value tells the DBMS Server how many sqlvar elements the statement is using (how many parameters are specified). If the sqld element of the SQLDA is set to 0 when you dynamically execute a procedure, it indicates that no parameters are being specified, and if there are parameters in the formal definition of the procedure, these are assigned null or default values when the procedure executes. If the procedure parameter is not nullable and does not have a default, an error is issued.

A parameter cannot be specified in the EXECUTE PROCEDURE statement that was not specified in the CREATE PROCEDURE statement.

Return_status is an integer variable that receives the return status from the procedure. If a *return_status* is not specified in the database procedure, or the return statement is not executed in the procedure, 0 is returned to the calling application.

Note: The INTO clause cannot be used in interactive SQL.

The statement must be terminated according to the rules of the host language.

Execute Procedure Loops

Use an execute procedure loop to retrieve and process rows returned by a row producing procedure using the RESULT ROW clause. The RESULT ROW clause identifies the host variables into which the values produced by the procedure return row statement are loaded. The entries in the RESULT ROW clause must match in both number and type the corresponding entries in the RESULT ROW declaration of the procedure.

The begin-end statements delimit the statements in the execute procedure loop. The code is executed once for each row as it is returned from the row producing procedure. Statements cannot be placed between the EXECUTE PROCEDURE statement and the BEGIN statement.

During the execution of the execute procedure loop, no other statements that access the database can be issued - this causes a runtime error. However, if your program is connected to multiple database sessions, you can issue queries from within the execute procedure loop by switching to another session. To return to the outer execute procedure loop, switch back to the session in which the EXECUTE PROCEDURE statement was issued. To avoid preprocessor errors, the nested queries cannot be within the syntactic scope of the loop but must be referenced by a subroutine call or some form of a GOTO statement.

There are two ways to terminate an execute procedure loop: run it to completion or issue the ENDEXECUTE statement. A host language GOTO statement cannot be used to exit or return to the execute procedure loop.

To terminate an execute procedure loop before all rows are retrieved the application must issue the ENDEXECUTE statement. This statement must be syntactically within the begin-end block that delimits the ENDEXECUTE procedure loop.

The following example retrieves a set of rows from a row producing procedure:

```
exec sql execute procedure deptsal_proc (deptid = :deptno)
result row (:deptname, :avgsal, :empcount);
exec sql begin;
 browse data;
if error condition then
exec sql endexecute;
 end if;
exec sql end;”
```

Permissions

To execute a procedure that you do not own, you must have EXECUTE privilege for the procedure, and must specify the *schema* parameter.

Locking

The locks taken by the procedure depend on the statements that are executed inside the procedure. All locks are taken immediately when the procedure is executed.

Performance

The first execution of the database procedure can take slightly longer than subsequent executions. For the first execution, the host DBMS may need to create a query execution plan.

Examples: Execute Procedure

The following EXECUTE PROCEDURE examples assume the following CREATE PROCEDURE statement has been successfully executed:

```
EXEC SQL CREATE PROCEDURE p
         (i INTEGER NOT NULL,
          d DATE,
          c VARCHAR(100)) AS ...
```

1. The following example uses a host language variable, a null constant, and an empty string.

   ```
   EXEC SQL EXECUTE PROCEDURE p
       (i=:ivar, d=null, c='')
       INTO :retstat;
   ```

2. The following example assumes the c parameter is null and uses a null indicator for the d parameter.

   ```
   EXEC SQL EXECUTE PROCEDURE p
       (i=:ivar, d=:dvar:ind)
       INTO :retstat;
   ```

3. The following example demonstrates the use of the WHENEVER statement for intercepting errors and messages from a database procedure.

   ```
   EXEC SQL WHENEVER SQLERROR GOTO err_exit;
   EXEC SQL WHENEVER SQLMESSAGE CALL SQLPRINT;

   EXEC SQL EXECUTE PROCEDURE p INTO :retstat;
   ...

   err_exit:
   EXEC SQL INQUIRE_SQL (:errbug = errortext);
   ```

4. The following example demonstrates a dynamically-executed EXECUTE PROCEDURE statement. The example creates and executes the dynamic equivalent of the following statement.

   ```
   EXEC SQL EXECUTE PROCEDURE enter_person
       (age = :i4_var, comment = :c100_var:indicator);
   ```

Dynamic version:

```
EXEC SQL INCLUDE sqlda;
allocate an SQLDA with 10 elements;
sqlda.sqln = 10;
sqlda.sqld = 2;

/* 20-byte character for procedure name */
proc_name = 'enter_person';

/* 4-byte integer to put into parameter "age" */
sqlda.sqlvar(1).sqltype = int;
sqlda.sqlvar(1).sqllen = 4;
sqlda.sqlvar(1).sqldata = address(i4_var)
sqlda.sqlvar(1).sqlind = null;
sqlda.sqlvar(1).sqlname ='age';

/* 100-byte nullable character to put into the
** parameter "comment"
*/
sqlda.sqlvar(2).sqltype = char;
sqlda.sqlvar(2).sqllen = 100;
sqlda.sqlvar(2).sqldata = address(c100_var);
sqlda.sqlvar(2).sqlind = address(indicator);
sqlda.sqlvar(2).sqlname = 'comment';

EXEC SQL EXECUTE PROCEDURE :proc_name
    USING DESCRIPTOR sqlda;
```

5. Call a database procedure, passing parameters by reference. This enables the procedure to return the number of employees that received bonuses and the total amount of bonuses conferred.

```
EXEC SQL EXECUTE PROCEDURE grant_bonuses
    (ecount = BYREF(:number_processed),
     btotal = BYREF(:bonus_total));
```

Fetch

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The FETCH statement fetches data from a database cursor into host language variables.

Syntax

The FETCH statement has the following formats:

Non-dynamic version:

```
EXEC SQL FETCH cursor_name
             INTO variable[:indicator_var] {, variable[:indicator_var]};
```

Dynamic version:

```
EXEC SQL FETCH cursor_name USING DESCRIPTOR descriptor_name;
```

cursor_name

Identifies an open cursor. *Cursor_name* can be either a string constant or a host language variable.

USING DESCRIPTOR *descriptor_name*

Identifies an SQLDA that contains type descriptions of one or more host language variables. Each element of the SQLDA is assigned the corresponding value in the current row of the cursor. For details, see the chapter "Working with Embedded SQL."

The variables listed in the INTO clause or within the descriptor must be type-compatible with the values being retrieved. If a result expression is nullable, the host language variable that receives that value must have an associated null indicator.

If the statement does not fetch a row--a condition that occurs after all rows in the set have been processed--the sqlcode of the SQLCA is set to 100 (condition not found) and no values are assigned to the variables.

The statement must be terminated according to the rules of the host language.

Description

The FETCH statement retrieves the results of the SELECT statement that is executed when a cursor is opened. When a cursor is opened, the cursor is positioned immediately before the first result row. The FETCH statement advances the cursor to the first (or next) row and loads the values in that row into the specified variables. Each FETCH statement advances the cursor one row.

There must be a one-to-one correspondence between variables specified in the INTO or USING clause of FETCH and expressions in the SELECT clause of the DECLARE CURSOR statement. If the number of variables does not match the number of expressions, the preprocessor generates a warning and, at runtime, the SQLCA variable sqlwarn3 is set to W.

The variables listed in the INTO clause can include structures that substitute for some or all of the variables. The structure is expanded by the preprocessor into the names of its individual variables; therefore, placing a structure name in the INTO clause is equivalent to enumerating all members of the structure in the order in which they were declared.

The variables listed in the INTO clause or within the descriptor must be type-compatible with the values being retrieved. If a result expression is nullable, the host language variable that receives that value must have an associated null indicator.

If the statement does not fetch a row—a condition that occurs after all rows in the set have been processed—the sqlcode of the SQLCA is set to 100 (condition not found) and no values are assigned to the variables.

The statement must be terminated according to the rules of the host language.

Examples: Fetch

1. Typical fetch, with associated cursor statements.

```
exec sql begin declare section;
  name  character_string(20);
  age   integer;
  exec sql end declare section;

exec sql declare cursor1 cursor for
  select ename, age
  from employee
  order by ename;
...

exec sql open cursor1;


loop until no more rows
  exec sql fetch cursor1
    into :name, :age;
  print name, age;
end loop;

exec sql close cursor1;
```

   Assuming the structure:

```
emprec
  name    character_string(20),
  age    integer;
```

   the fetch in the above example could have been written

```
exec sql fetch cursor1
  into :emprec;
```

   The preprocessor would then interpret that statement as though it had been written

```
exec sql fetch cursor1
  into :emprec.name, :emprec.age;
```

2. Fetch using an indicator variable.

```
exec sql fetch cursor2 into :name,
:salary:indicator_var;
```

Get Dbevent

Valid in: ESQL, OpenAPI

The GET DBEVENT statement gets an event previously defined by the CREATE DBEVENT statement.

The GET DBEVENT statement receives database events for which an application is registered. The GET DBEVENT statement returns the next database event from the database event queue. To obtain database event information, issue the INQUIRE_SQL statement.

Syntax

The GET DBEVENT statement has the following format:

```
EXEC SQL GET DBEVENT [WITH NOWAIT | WAIT [= wait_value]];
```

WITH NOWAIT

(Default) Checks the queue and returns immediately.

WITH WAIT[=*wait_value*]

Waits indefinitely for the next database event to arrive. If with wait = *wait_value* is specified, GET DBEVENT returns when a database event arrives or when *wait_value* seconds have passed, whichever occurs first. If GET DBEVENT times out before a database event arrives, no database event is returned.

Wait_value can be specified using an integer constant or integer host language variable.

The WITH WAIT option cannot be used within a select loop or a database procedure message processing routine called as the result of the WHENEVER SQLMESSAGE condition.

Help

Valid in: SQL

The HELP statement gets information about SQL and a variety of database objects.

The HELP statement displays information about the contents of the database or specific tables. In addition, help can be used at the terminal monitor to obtain information regarding OpenSQL, including such features as the syntax of OpenSQL statements and valid data types.

Syntax

The HELP statement has the following format:

```
HELP [*]
HELP objectname {, objectname}
HELP TABLEe tablename {, tablename}
HELP VIEW viewname {, viewname}
HELP INDEX indexname {, indexname}
HELP HELP
HELP SQL
HELP sql_statement
```

HELP

Lists all user tables, views, and indexes that exist in the current database. (System catalogs are not listed.)

HELP *

Displays information about all user-defined (not system) tables, views, and indexes in the database.

HELP *objectname*

Displays information on the specified table, view, or index.

For example, HELP *tablename* provides the name, owner, creation date and time, and the DBMS version under which the table was created. Displays the name, data type, length, nullability, default, and key sequence for each column in the table.

The asterisk (*) can be used as a pattern matching character when specifying an object name. For example, if you type HELP TABLE EMP*, you receive help on all tables in the database whose names begin with EMP. If you type, HELP TABLE *EMP, you receive help on all the tables whose names end with EMP.

HELP TABLE *tablename* {, *tablename*}

Displays detailed information about the specified tables.

Displays the same information as HELP *tablenam*e plus additional table information, depending on the particular Enterprise Access product or server.

HELP VIEW *viewname* {, *viewname*}

Displays detailed information about the specified views, including the text of the view, the view name, owner and the state of the check option.

HELP INDEX *indexname* {, *indexname*}

Displays detailed information about the specified indexes, including name, owner, creation date and time, DBMS version under which it was created, and, for each column, its name, data type, length, nullability, default attribute, and key sequence.

HELP HELP

Displays a list of OpenSQL features for which help is available.

HELP SQL

Displays general information about OpenSQL.

HELP *sql_statement*

Displays information on the specified SQL statement.

Examples: Help

1. Retrieve a list of all tables, views, and indexes in the database.

   ```
   help;
   ```

2. Retrieve help about the employee table.

   ```
   help employee;
   ```

3. Retrieve help about the employee and dept tables.

   ```
   help employee, dept;
   ```

4. Retrieve the definition of the view highpay.

   ```
   help view highpay;
   ```

5. List information on the select statement.

   ```
   help select;
   ```

Include

Valid in: ESQL

The INCLUDE statement incorporates external files into your program's source code.

Syntax

The INCLUDE statement has the following format:

```
EXEC SQL INCLUDE filename | SQLCA | SQLDA;
```

Description

The INCLUDE statement is typically used to include variable declarations, although it is not restricted to such use. When used to include variable declarations, it must be inside an embedded SQL declaration section.

Note: The file generated by dclgen must be specified using the INCLUDE statement.

The file specified by the INCLUDE statement must contain complete statements or declarations. For example, it is illegal to use INCLUDE in the following manner, where the file, predicate, contains a common predicate for SELECT statements.

Incorrect:

```
exec sql select ename
       from employee
       where
       exec sql include 'predicate';
```

Filename must be a quoted string constant specifying a file name or a logical or environment variable that contains a file name. If a file name is specified without an extension, the default extension of your host language is assumed.

The specified file can contain declarations, host language statements, embedded SQL statements, and nested includes. When the original source file is preprocessed, the INCLUDE statement is replaced by a host language include directive, and the included file is also preprocessed.

There are two special instances of the INCLUDE statement:

- **INCLUDE SQLCA -** Includes the SQL Communications Area.
- **INCLUDE SQLDA -** Includes the definitions associated with the SQL Descriptor Area.

Both these statements must be placed outside all declaration sections, preferably at the start of the program. The statement must be terminated as required by the rules of your host language.

Examples: Include

1. Include the SQLCA in the program.

```
exec sql include sqlca;
```

2. Include global variables.

```
exec sql begin declare section;
  exec sql include 'global.var';
exec sql end declare section;
```

3. Include a file that contains header files that list variable declarations.

```
exec sql begin declare section;
  exec sql include 'mypath:global.var';
exec sql end declare section;
```

Inquire_sql

Valid in: ESQL

The INQUIRE_SQL statement provides an application program with a variety of runtime information.

Syntax

The INQUIRE_SQL statement has the following format:

```
EXEC SQL INQUIRE_SQL (:variable = object {, variable = object});
```

variable

Specifies the name of a program variable.

object

Specifies a valid INQUIRE_SQL object name, as follows:

Object	Data Type	Description
dbeventname	Character	The name of the event (assigned using the CREATE DBEVENT statement). The receiving variable must be large enough for the full event name; if the receiving variable is too small, the event name is truncated to fit.
dbeventowner	Character	The creator of the event.
dbeventdatabase	Character	The database in which the event was raised.
dbeventtime	Date	The date and time at which the event was raised.
dbeventtext	Character	The text (if any) specified as the *event_text*

Object	Data Type	Description
		parameter when the event was raised. The receiving value must be a 256-character string; if the receiving variable is too small, the text is truncated to fit.
dbmserror	Integer	Returns the number of the error caused by the last query. This number corresponds to the value of sqlerrd(1), the first element of the sqlerrd array in the SQLCA. To specify whether a local or generic error is returned, use the SET_SQL(ERRORTYPE) statement.
column_name	Character	Valid only in a data handler routine that retrieves data (in conjunction with a SELECT or FETCH statement); returns the name of the column for which the data handler was invoked. The receiving variable must be a minimum of 32 bytes; if the host language uses null-terminated strings, an additional byte is required.
columntype	Integer	Valid only in a data handler routine that retrieves data (in conjunction with a SELECT or FETCH statement); returns an integer indicating the data type of the column for which the data handler was invoked.
connection_name	Character	Returns the connection name for the current session.
connection_target	Character	Returns the node and database to which the current session is connected; for example, 'bignode::mydatabase'.
endquery	Integer	Returns 1 if the previous fetch statement was issued after the last row of the cursor, 0 if the last fetch statement returned a valid row. This is identical to the NOT FOUND condition (value 100) of the SQLCA variable sqlcode, which can be checked after a fetch statement is issued. If endquery returns '1', the variables assigned values from the fetch are left unchanged.
errorno	Integer	Returns the error number of the last query as a positive integer. The error number is cleared before each embedded SQL statement. ERRORNO is meaningful only immediately after the statement in question. This error number is the same as the positive value returned in the SQLCA variable sqlcode, except in two cases: A single query generates multiple different errors,

Object	Data Type	Description
		in which case the sqlcode identifies the first error number, and the ERRORNO object identifies the last error. After switching sessions. In this case, sqlcode reflects the results of the last statement executed before switching sessions, while ERRORNO reflects the results of the last statement executed in the current session. If a statement executes with no errors or if a positive number is returned in sqlcode (for example, +100 to indicate no rows affected), the error number is set to 0.
errortext	Character	Returns the error text of the last query. The error text is only valid immediately after the database statement in question. The error text that is returned is the complete error message of the last error. This message can have been truncated when it was deposited into the SQLCA variable sqlerrm. The message includes the error number and a trailing end-of-line character. A character string result variable of size 256 must be sufficient to retrieve all error messages. If the result variable is shorter than the error message, the message is truncated. If there is no error message, a blank message is returned.
errortype	Character	Returns 'genericerror' if generic errors are returned to ERRORNO and sqlcode, or 'dbmserror' if local DBMS Server errors are returned to ERRORNO and sqlcode. For information about generic and local errors, see the chapter "Working with Transactions and Handling Errors."
messagenumber	Integer	Returns the number of the last message statement executed inside a database procedure. If there was no message statement, a zero is returned. The message number is defined by the database procedure programmer.
messagetext	Character	Returns the message text of the last message statement executed inside a database procedure. If there is no text, a blank is returned. If the result variable is shorter than the message text, the message is truncated. The message text is defined by the database procedure programmer.
object_key	Character	Returns the logical object key added by the last INSERT statement, or -1 (in the indicator variable)

Object	Data Type	Description
		if no logical key was assigned.
prefetchrows	Integer	Returns the number of rows the DBMS Server buffers when fetching data using readonly cursors. This value is reset every time a readonly cursor is opened. If your application is using this feature, be sure to set the value before opening a readonly cursor. For details, see the chapter "Working with Embedded SQL."
programquit	Integer	Returns 1 if the programquit option is enabled (using SET_SQL(PROGRAMQUIT). If programquit is enabled, the following errors cause embedded SQL applications to abort: ▪ Issuing a query when not connected to a database ▪ Failure of the DBMS Server ▪ Failure of communications services ▪ Returns 0 if applications continue after encountering such errors.
querytext	Character	Returns the text of the last query issued; valid only if this feature is enabled. To enable or disable the saving of query text, use the SET_SQL(SAVEQUERY=1|0) statement. A maximum of 1024 characters is returned; if the query is longer, it is truncated to 1024 characters. If the receiving variable is smaller than the query text being returned, the text is truncated to fit. If a null indicator variable is specified in conjunction with the receiving host language variable, the indicator variable is set to -1 if query text cannot be returned, 0 if query text is returned successfully. Query text cannot be returned if (1) savequery is disabled, (2) no query has been issued in the current session, or (3) the INQUIRE_SQL statement is issued outside of a connected session.
rowcount	Integer	Returns the number of rows affected by the last query. The following statements affect rows: INSERT, DELETE, UPDATE, SELECT, FETCH, MODIFY, CREATE INDEX, CREATE TABLE AS SELECT, and COPY. If any of these statements runs successfully, the value returned for rowcount is the same as the value of the SQLCA variable

Object	Data Type	Description
		sqlerrd(3). If these statements generate errors, or if statements other than these are run, the value of ROWCOUNT is negative and the value of sqlerrd(3) is zero. Exception: for MODIFY TO TRUNCATED, INQUIRE_SQL(ROWCOUNT) always returns 0.
savequery	Integer	Returns 1 if query text saving is enabled, 0 if disabled.
session	Integer	Returns the session identifier of the current database session. If the application is not using multiple sessions or there is no current session, session 0 is returned.
table_key	Character	Returns the logical table key added by the last INSERT statement, or -1 (in the indicator variable) if no logical key was assigned.
transaction	Integer	Returns a value of 1 if there is a transaction open.

All character values are returned in lower case. If no event is queued, an empty or blank string is returned (depending on your host language conventions).

Description

The INQUIRE_SQL statement enables an embedded OpenSQL program to retrieve a variety of runtime information, such as:

- Information about the last executed database statement
- Status information, such as the current session ID, the type of error (local or generic) being returned to the application, and whether a transaction is currently open

The INQUIRE_SQL statement does not execute queries; the information INQUIRE_SQL returns to the program reflects the results of the last query that was executed. For this reason, the INQUIRE_SQL statement must be issued after the database statement about which information is desired, and before another database statement is executed (and resets the values returned by INQUIRE_SQL).

Some of the information returned by INQUIRE_SQL is also available in the SQLCA. For example, the error number returned by the object errorno is also available in the SQLCA sqlcode field.

Similarly, when an error occurs, the error text can be retrieved using INQUIRE_SQL with the errortext object or it can be retrieved from the SQLCA sqlerrm variable. Errortext provides the complete text of the error message, which is often truncated in sqlerrm.

This statement must be terminated according to the rules of your host language.

Inquiring About Database Events

The following table lists the INQUIRE_SQL parameters that return information about a database event. All character values are returned in lower case. If no event is queued, an empty or blank string is returned (depending on your host language conventions).

Object	Data Type	Description
dbeventname	Character	The name of the event (assigned using the CREATE DBEVENT statement). The receiving variable must be large enough for the full event name; if the receiving variable is too small, the event name is truncated to fit.
dbeventowner	Character	The creator of the event.
dbeventdatabase	Character	The database in which the event was

Object	Data Type	Description
		raised.
dbeventtime	Date	The date and time at which the event was raised.
dbeventtext	Character	The text (if any) specified as the *event_text* parameter when the event was raised. The receiving value must be a 256-character string; if the receiving variable is too small, the text is truncated to fit.

Types of Inquiries

The following table lists the valid inquiries that can be performed using the INQUIRE_SQL statement:

Object	Data Type	Comment
dbmserror	integer	The number of the error caused by the last query. This number corresponds to the value of sqlerrd(1), the first element of the sqlerrd array in the SQLCA. You can specify whether a local or generic error is returned using SET_SQL(errortype).
endquery	integer	If the previous FETCH statement was issued after the last row of the cursor, endquery returns the value "1." If the last FETCH statement returns a valid row, the value returned is "0." This is identical to the NOT FOUND condition (value 100) of the SQLCA variable sqlcode, which can be checked after a fetch statement is issued. Like the NOT FOUND condition, when endquery returns "1," the variables assigned values from the fetch remain unchanged.
errorno	integer	A positive integer, representing the error number of the last query. The error number is cleared before each embedded OpenSQL statement, so that this object is only valid immediately after the statement in question. This error number is the same as the positive value of the SQLCA variable sqlcode, except in two cases:

Object	Data Type	Comment
		A single query generates multiple different errors, in which case the sqlcode identifies the first error number, and the errorno object identifies the last error. After switching sessions. In this case, sqlcode reflects the results of the last statement executed before switching sessions, while errorno will reflect the results of the last statement executed in the current session. If a statement executes with no errors or sqlcode is set to a positive number (for example, +100 to indicate no rows affected), then the error number is set to 0.
errortext	character	The error text of the last query. The error text is only valid immediately after the database statement in question. The text that is returned is the complete error message of the last error. This message may have been truncated when it was deposited into the SQLCA variable sqlerm. A character string result variable of size 512 should be sufficient to retrieve all OpenSQL error messages. If the result variable is shorter than the error message, the message is truncated. If there is no error message, a blank message is returned.
errortype	character	Returns genericerror if OpenSQL returns generic error numbers to errorno and sqlcode, or dbmserror if OpenSQL returns local DBMS error numbers to errorno and sqlcode. For information about generic and local errors, see "Chapter 7: OpenSQL Features."
programquit	integer	Returns 1 if applications quit: ▪ After issuing a query when not connected to a database. ▪ If the Enterprise Access product or server fails. ▪ If communications services fail. Returns 0 if applications continue after encountering such errors.
querytext	character	Returns the text of the last query issued. Valid

Object	Data Type	Comment
		only if this feature is enabled. To enable or disable the saving of query text, use set_sql(savequery). A maximum of 1024 characters is returned. If the query is longer, it is truncated to 1024 characters. If the receiving variable is smaller than the query text being returned, the text is truncated to fit. If a null indicator variable is specified together with the receiving host language variable, the indicator variable is set to -1 if query text cannot be returned, 0 if query text is returned successfully. Query text cannot be returned if (1) savequery is disabled, (2) no query has been issued in the current session, or (3) the inquire_sql statement is issued outside of a connected session.
rowcount	integer	The number of rows affected by the last query. "Affected" means subject to any of the following statements: insert, delete, update, select, fetch, create index, or create table as select. If any of these statements run successfully, the value of rowcount is the same as the value of the SQLCA variable sqlerrd(3). If these statements generate errors, or if statements other than these are run, then the value of rowcount is negative and the value of sqlerrd(3) is 0.
savequery	integer	Returns 1 if query text saving is enabled, 0 if disabled.
session	integer	Returns the session identifier of the current database session. If the application is not using multiple sessions or there is no current session, 0 is returned.
transaction	integer	Returns a value of 1 if there is a transaction open. Returns 0 if no transaction is open.

Example: Inquire_sql

Execute some database statements, and handle errors by displaying the message and aborting the transaction.

```
exec sql whenever sqlerror goto err_handle;

exec sql select name, sal
  into :name, :sal
  from employee
  where eno = :eno;

if name = 'Badman' then
  exec sql delete from employee where eno = :eno;
else if name = 'Goodman' then
  exec sql update employee set sal = sal + 3000
    where eno = :eno;
end if;

exec sql commit;

...

err_handle:

exec sql whenever sqlerror continue;
exec sql inquire_sql (:err_msg = errortext);
print 'Enterprise Access product error: ',
sqlca.sqlcode;
print err_msg;
exec sql rollback;

end if;
```

Insert

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The INSERT statement inserts rows into a table.

Syntax

The INSERT statement has the following format:

```
[EXEC SQL [REPEATED]]INSERT INTO table_name [(column {, column})]
              [VALUES (value{, value})] | [subselect]
```

Description

The INSERT statement inserts new rows into the specified table. Use either the values list or specify a *subselect*. When using the values list, only a single row can be inserted with each execution of the statement. If specifying a *subselect*, the statement inserts all the rows that result from the evaluation of the *subselect*. The subselect must not select rows from the table into which you are inserting rows; specifically, you cannot specify the same table in the INTO clause of the INSERT statement and the FROM clause of the subselect.

The *column* list identifies the columns of the specified table into which the values are placed. When the column list is included, OpenSQL places the first value in the values list or *subselect* into the first column named, the second value into the second column named, and so on. The data types of the values must be compatible with the data types of the columns in which they are placed.

The list of column names can be omitted only if:

- You specify a subselect that retrieves a value for each column in *table_name*. The values must be of an appropriate data type for each column and must be retrieved in an order corresponding to the order of the columns in *table_name*.
- There is a one-to-one correspondence between the values in the values list and the columns in the table. That is, the values list must have a value of the appropriate data type for each column and the values must be listed in an order corresponding to the order of the columns in the table.

Values in the values list must be string or numeric literals or one of the OpenSQL constants. When the column list is included, any columns in the table that are not specified in the column list are assigned their default value. A value must be specified for mandatory columns. (Mandatory columns are columns defined as not default or not null with no default specified.)

Embedded Usage

Host language variables can be used within expressions in the values clause or in the search condition of the *subselect*. Variables used in search conditions must denote constant values, and cannot represent names of database columns or include any operators. A host string variable can also replace the complete search condition of the subselect, as when it is used in the forms system query mode. Host language variables that correspond to column expressions can include null indicator variables.

The keyword REPEATED directs the Enterprise Access product or server to encode the insert and save its execution plan when it is first executed. This encoding can improve the performance of subsequent executions of the same insert.

Do not specify the REPEATED option for INSERT statement that is constructed using dynamic OpenSQL. A dynamic WHERE clause cannot be used in a repeated insert: the query plan is saved when the query is first executed, and subsequent changes to the WHERE clause are ignored.

The VALUES clause can include structure variables that substitute for some or all of the expressions. The structure is expanded by the preprocessor into the names of its individual members. Therefore, placing a structure name in the VALUES clause is equivalent to enumerating all members of the structure in the order in which they were declared.

The sqlerrd(3) of the SQLCA indicates the number of rows inserted by the statement. If no rows are inserted (for example, if no rows satisfied the *subselect* search condition), then the sqlcode variable of the SQLCA is set to 100.

Examples: Insert

The following are INSERT statement examples:

1. Add a row to an existing table.

```
INSERT INTO emp (name, sal, bdate)
    VALUES ('Jones, Bill', 10000, 1944);
```

2. Insert into the job table all rows from the newjob table where the job title is not Janitor.

```
INSERT INTO job (jid, jtitle, lowsal, highsal).
SELECT job_no, title, lowsal, highsal
FROM newjob
WHERE title <> 'Janitor';
```

3. Add a row to an existing table, using the default columns.

```
INSERT INTO emp
    VALUES ('Jones, Bill', 10000, 1944);
```

4. Use a structure to insert a row.

```
  /* Description of table employees from
    database deptdb */

  EXEC SQL DECLARE employees TABLE
   (eno           SMALLINT NOT NULL,
    ename         CHAR(20) NOT NULL,
    age           SMALLINT,
    jobcode       SMALLINT,
    sal           FLOAT NOT NULL,
    deptno        SMALLINT);

EXEC SQL BEGIN DECLARE SECTION;

    emprec
        int           eno;
        char          ename[21];
        int           age;
        int           job;
        float         sal;
        int           deptno;

EXEC SQL END DECLARE SECTION;


  /* Assign values to fields in structure */

  eno = 99;
  ename = "Arnold K. Arol”;
  age = 42;
  jobcode = 100;
  sal = 100000;
  deptno=47;

  EXEC SQL CONNECT deptdb;
  EXEC SQL INSERT INTO employees VALUES (:emprec);
  EXEC SQL DISCONNECT;
```

5. Insert explicit values into a t1 row regardless of the identity column definition. Without the OVERRIDING clause, this statement would generate a syntax error.

```
INSERT INTO t1 OVERRIDING SYSTEM VALUE VALUES(1, 2, 3)
```

Open

Valid in: ESQL

The OPEN statement opens a cursor for processing.

Syntax

The OPEN statement has the following format:

Non-dynamic version:

```
EXEC SQL OPEN cursor_name [FOR READONLY];
```

Dynamic version:

```
EXEC SQL OPEN cursor_name [FOR READONLY]
             [USING variable {, variable} |
             USING DESCRIPTOR descriptor_name];
```

FOR READONLY

Opens the cursor for reading only, even though the cursor may have been declared for update. This clause improves the performance of data retrieval, and can be used whenever appropriate.

USING *variable* {,*variable*} | USING DESCRIPTOR *descriptor_name*

Provides values for the constants that are in the prepared SELECT statement.

Description

The OPEN statement executes the SELECT statement specified when the cursor was declared and positions the cursor immediately before the first row returned. (To actually retrieve the rows, use the FETCH statement.) A cursor must be opened before it can be used in any data manipulation statements such as FETCH, UPDATE, or DELETE and a cursor must be declared before it can be opened.

When a cursor that was declared for a dynamically prepared SELECT statement is opened, use the USING clause if the prepared SELECT statement contains constants specified with question marks. For information about using question marks to specify constants in prepared statements, see Prepare (see page 283).

The USING clause provides the values for these "unspecified" constants in the prepared SELECT so that the OPEN statement can execute the SELECT. For example, assume that your application contains the following dynamically prepared SELECT statement:

```
statement_buffer =
'select * from' + tablename + 'where low < ? and
        high > ?';
exec sql prepare sel_stmt from :statement_buffer;
```

When opening the cursor for this prepared SELECT statement, values must be provided for the question marks in the WHERE clause. The USING clause performs this task. For example:

```
Declare the cursor for sel_stmt;
assign values to variables named "low" and "high";
exec sql open cursor1
using :low, :high;
```

The values in the low and high variables replace the question marks in the WHERE clause and the DBMS can evaluate the SELECT statement accordingly. If Descriptor Area (SQLDA) is used, then the values that replace the question marks are taken from variables pointed to by the sqlvar elements of the descriptor. Allocate the SQLDA and the variables to which the sqlvar elements point and place values in the variables before using the descriptor in an OPEN CURSOR statement.

The same cursor can be opened and closed (with the CLOSE statement) any number of times in a single program. It must be closed, however, before it can be reopened.

A string constant or a host language variable can be used to represent *cursor_name*. This statement must be terminated according to the rules of your host language.

Examples: Open

The following are OPEN statement examples:

1. Declare and open a cursor.

```
exec sql declare cursor1 cursor for
    select :one + 1, ename, age
    from employee
    where age :minage;
...
exec sql open cursor1;
```

   When the OPEN statement is encountered, the variables, one and minage, are evaluated. The first statement that follows the opening of a cursor must be a FETCH statement to define the cursor position and retrieve data into the indicated variables:

```
exec sql fetch cursor1
    into :two, :name, :age;
```

   The value of the expression, :one + 1, is assigned to the variable, two, by the fetch.

2. The following example demonstrates the dynamic SQL syntax. In a typical application the prepared statement and its parameters are constructed dynamically.

```
select_buffer =
    'select * from employee where eno = ?';
exec sql prepare select1 from :select_buffer;
exec sql declare cursor2 cursor for select1;
eno = 1234;
exec sql open cursor2 using :eno;
```

Prepare

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The PREPARE statement prepares and names a dynamically constructed OpenSQL statement for execution.

Syntax

The PREPARE statement has the following format:

```
EXEC SQL PREPARE statement_name
             [INTO descriptor_name [USING NAMES]]
             FROM string_constant | string_variable;
```

Description

The PREPARE statement encodes the dynamically constructed OpenSQL statement string in the from clause and assigns it the specified *statement_name*.

When the program subsequently executes the prepared statement, it uses the name to identify the statement, rather than the full statement string. Both the name and statement string can be represented by either a string constant or a host language variable.

Within the statement string, replace constant expressions in WHERE clauses, INSERT VALUES clauses, and UPDATE SET clauses with question marks. When the statement executes, these question marks are replaced with specified values. Question marks cannot be used in place of table or column names or reserved words.

To illustrate, the following example prepares and executes a DELETE statement on a dynamically defined table:

```
statement_buffer =
'delete from ' + table_name + ' where code = ?';
exec sql prepare del_stmt from :statement_buffer;

…

exec sql execute del_stmt using :code;
```

The value in the variable, code, replaces the ? in the WHERE clause of the prepared DELETE statement.

Illustrating incorrect usage, the following example is wrong because it includes a parameter specification in place of the table name:

```
exec sql prepare bad_stmt
from 'delete from ? where code = ?';
```

Whenever an application executes a prepared statement that contains parameters specified with questions marks, the program must supply values for each question mark. If the statement string is blank or empty, OpenSQL returns a runtime syntax error.

If the statement name identifies an existing prepared statement, the existing statement is destroyed and the new statement takes effect. This rule holds across the dynamic scope of the application. The statement name must not identify an existing statement name that is associated with an open cursor. The cursor must be closed before its statement name can be destroyed. Once prepared, the statement can be executed any number of times.

However, if a transaction is rolled back or committed, the prepared statement becomes invalid. If the prepared statement is to be executed only once, EXECUTE IMMEDIATE should be used on the statement string. If the prepared statement is to be executed repeatedly, the prepare and execute sequence should be used.

The following statements cannot be prepared and executed dynamically:

call	disconnect	inquire_sql
close	endselect	open
connect	execute immediate	set
declare	execute	set_sql
describe	fetch help	whenever
direct execute immediate	include	

In addition, you cannot prepare and dynamically execute OpenSQL statements that include the keyword REPEATED.

If the statement string is a SELECT statement, the select must not include an INTO clause. The SELECT statement string can include the different clauses of the cursor SELECT statement, such as the FOR UPDATE and ORDER BY clauses.

As with EXECUTE IMMEDIATE, the statement string must not include exec sql, any host language terminators, or references to variable names. If your statement string includes embedded quotes, it is easiest to specify the string in a host language variable. If you specify a string that includes quotes as a string constant, remember that quoted characters *within* the statement string must follow the OpenSQL string delimiting rules. Consequently, even if your host language delimits strings with double quotes, the quoted characters within the statement string must be delimited by single quotes. For complete information about embedding quotes within a string literal, see the *Embedded SQL Companion Guide*.

The INTO *descriptor_name* clause is equivalent to issuing the DESCRIBE statement after the statement is successfully prepared. For example, the PREPARE statement

```
exec sql prepare prep_stmt
  into sqlda from :statement_buffer;
```

is equivalent to the following PREPARE and DESCRIBE statements:

```
exec sql prepare prep_stmt from :statement_buffer;
exec sql describe prep_stmt into sqlda;
```

The INTO clause returns the same information as does the DESCRIBE statement. If the prepared statement is a SELECT, the descriptor will contain the data types, lengths, and names of the result columns. If the statement was not a SELECT, the descriptor's sqld field will contain a zero. For more information about the results of describing a statement, see the chapter "Dynamic OpenSQL" and Describe.

This statement must be terminated according to the rules of your host language.

Example: Prepare

A two-column table, whose name is defined dynamically but whose columns are called high and low, is manipulated within an application, and statements to delete, update and select the values are prepared.

```
get table_name from a set of names;

statement_buffer = 'DELETE FROM ' + table_name +
    ' WHERE high = ? AND low = ?';
EXEC SQL PREPARE del_stmt FROM :statement_buffer;

statement_buffer = 'INSERT INTO ' + table_name +
    ' VALUES (?, ?)';
EXEC SQL PREPARE ins_stmt FROM :statement_buffer;

statement_buffer = 'SELECT * FROM ' + table_name
    + ' WHERE low ?';
EXEC SQL PREPARE sel_stmt FROM :statement_buffer;

...

EXEC SQL EXECUTE del_stmt USING :high, :low;

...

EXEC SQL EXECUTE ins_stmt USING :high, :low;

...

EXEC SQL DECLARE sel_csr CURSOR FOR sel_stmt;
EXEC SQL OPEN sel_csr USING :high, :low;
loop while more rows
    EXEC SQL FETCH sel_csr INTO :high1, :low1;
    ...
end loop;
```

Raise Dbevent

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The RAISE DBEVENT statement enables an application to notify other applications of its status. That is, it communicates status information to other sessions that are registered to receive *event_name*.

For a full description of database events, see Database Events (see page 191).

Syntax

The RAISE DBEVENT statement has the following format:

```
[EXEC SQL] RAISE DBEVENT [schema.]event_name [event_text]
```

event_name

Specifies an existing database event name.

event_text

Passes a string (maximum 256 characters) to receiving applications. To obtain the text, receiving applications must use the INQUIRE_SQL(dbeventtext) statement.

Embedded Usage

In an embedded RAISE DBEVENT statement, *event_name* cannot be specified using a host language variable, though *event_text* can be specified using a host string variable.

Register Dbevent

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REGISTER DBEVENT statement specifies that an application is to be notified when an event is raised.

Syntax

The REGISTER DBEVENT statement has the following format:

```
[EXEC SQL] REGISTER DBEVENT [schema.]event_name;
```

Description

The REGISTER DBEVENT statement enables a session to specify the database events it intends to receive. For a full description of database events, see Database Events (see page 191).

A session receives only the database events for which it has registered. To remove a registration, use the REMOVE statement. After registering for a database event, the session receives the database event using the GET DBEVENT statement.

Embedded Usage

Event_name cannot be specified using a host language variable.

Remove Dbevent

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REMOVE DBEVENT statement removes a database event for which an application has previously registered.

Syntax

The REMOVE DBEVENT statement has the following format:

```
[EXEC SQL] REMOVE DBEVENT [schema.]event_name;
```

Description

The REMOVE DBEVENT statement specifies that an application no longer intends to receive the specified database event.

If the database event has been raised before the application removes the registration, the database event remains queued to the application and will be received when the application issues the GET DBEVENT statement.

Rollback

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ROLLBACK statement backs out the changes made during the current transaction.

Syntax

The ROLLBACK statement has the following format:

```
[EXEC SQL] ROLLBACK [WORK]
```

WORK

Is an optional keyword included for compatibility with the ISO and ANSI SQL standards. It has no effect.

Embedded Usage

In addition to aborting the current transaction, an embedded ROLLBACK statement:

- Closes all open cursors
- Discards all statements that were prepared in the current transaction

Performance

Executing a rollback undoes some or all of the work done by a transaction. The time required to do this is generally the same amount of time taken to perform the work.

Select (interactive)

Valid in: SQL, OpenAPI, ODBC, JDBC, .NET

The SELECT (interactive) statement returns values from tables or views.

Syntax

The SELECT (interactive) statement has the following format:

```
SELECT [FIRST rowCount] [ALL | DISTINCT]
              * | result_expression{, result_expression}
              [FROM from_source {, from_source}
              [WHERE search_condition]
              [GROUP BY column{, column}]
              [HAVING search_condition]
              {UNION [ALL]
              (select)
              [ORDER BY result_column [ASC | DESC]
                           {, result_column [ASC | DESC]}];
```

where *result_expression* is one of the following:

- [*schema*.]*tablename*.*

  Selects all columns

- [[*schema*.]*tablename*.]*columnname* [[AS] *result_column*]

  Selects one column

- expression [AS] result_column

Description

The SELECT statement returns values from one or more tables or views in the form of a single result table. Using the various clauses of the SELECT statement, you can specify:

- Qualifications for the values in the result table
- Sorting and grouping of the values in the result table

This statement description presents details of the SELECT statement in interactive OpenSQL (ISQL). In ISQL the results of a query are displayed on your terminal. In embedded OpenSQL (ESQL), results are returned in host language variables. For details about using the SELECT statement in ESQL, see Select (embedded) (see page 306).

The following sections describe the clauses of the SELECT statement, explain how to create simple queries, and explain how the results of a query are obtained.

Select Statement Clauses

The SELECT statement has the following clauses:

- SELECT
- FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- OFFSET
- FETCH FIRST
- UNION

Select Clause

The SELECT clause specifies which values are to be returned. To display all the columns of a table, use the asterisk wildcard character (*). For example, the following query displays all rows and columns from the employees table:

```
select * from employees;
```

To select specific columns, specify the column names. For example, the following query displays all rows, but only two columns from the employees table:

```
select ename, enumber from employees;
```

To specify the table from which the column is to be selected, use the [*schema*.]*table.columnname* syntax. For example:

```
select personnel.managers.name,
            personnel.employees.name
```

In the preceding example, both source tables contain a column called, name. The column names are preceded by the name of the source table. The first column of the result table contains the values from the name column of the managers table, and the second column contains the values from the name column of the employees table. If a column name is used in more than one of the source tables, you must qualify the column name with the table to which it belongs, or with a correlation name. For details, see From Clause (see page 295).

The number of rows in the result table can be limited using the **first** clause. *RowCount* is a positive integer value that indicates the maximum rows in the result table. The query is effectively evaluated without concern for the **first** clause, but only the first "n" rows (as defined by *rowCount*) are returned. This clause cannot be used in a WHERE clause subselect and it may only be used in the first of a series of UNIONed selects. However, it may be used in the CREATE TABLE...AS SELECT and INSERT INTO...SELECT statements. When used with CREATE TABLE...AS SELECT and INSERT INTO…SELECT statements, first n should not be used with the ORDER BY clause.

To eliminate duplicate rows from the result table, specify the keyword DISTINCT. To preserve duplicate rows, specify the keyword all. By default, duplicate rows are preserved.

For example, the following table contains order information. The partno column contains duplicate values, because different customers have placed orders for the same part.

partno	customerno	qty	unit_price
123-45	101	10	10.00
123-45	202	100	10.00
543-21	987	2	99.99
543-21	654	33	99.99
987-65	321	20	29.99

The following query displays the part numbers for which there are orders on file:

```
select distinct partno from orders
```

The result table looks like this:

Partno
123-45
543-21
987-65

A constant value can be included in the result table. For example:

```
select 'Name:', ename, date('today'),
  edept  from employees;
```

The preceding query selects all rows from the employees table. The result table is composed of the string constant 'Name:', the employee's name, today's date (specified using the constant today), and the employee's department, or if there is no department assigned, the string constant 'Unassigned'.

The result table looks like this (depending, of course, on the data in the employees table):

COL1	ename	COL3	COL4
Name:	Mike Sannicandro	Aug 8, 1999	Shipping
Name:	Dave Murtagh	Aug 8, 1999	Purchasing

COL1	ename	COL3	COL4
Name:	Benny Barth	Aug 8, 1999	Unassigned
Name:	Dean Reilly	Aug 8, 1999	Lumber
Name:	Al Obidinski	Aug 8, 1999	Unassigned

The SELECT clause can be used to obtain values calculated from the contents of a table. For example:

```
select ename, annual_salary/52 from employees;
```

The preceding query calculates each employee's weekly salary based on their annual salary.

Aggregate functions can be used to calculate values based on the contents of column. For example:

```
select max(salary), min(salary), avg(salary)
  from employees;
```

The preceding query returns the highest, lowest, and average salary from the employees table. These values are based on the amounts stored in the salary column. For details about aggregate functions, see the chapter "Elements of OpenSQL Statements."

To specify a name for a column in the result table, use the AS *result_column* clause. For example:

```
select ename, annual_salary/52 as weekly_salary
from employees;
```

In the preceding example, the name, weekly_salary, is assigned to the second result column. If you omit a result column name for columns that are not drawn directly from a table (for example, calculated values or constants), the result columns are assigned the default name COL*n*, where *n* is the column number. Result columns are numbered from left to right. Column names cannot be assigned in SELECT clauses that use the asterisk wildcard (*) to select all the columns in a table.

From Clause

The FROM clause specifies the source tables and views from which data is to be read. The specified tables and views must exist at the time the query is issued. The tables or views must be specified using the following syntax:

[*schema.*]*table* [*corr_name*]

where *table* is the name of a table or view. To ensure program portability, specify no more than 15 tables in a query, including the tables in the from list and tables in subqueries. (Individual host database management systems may allow more than 15 tables.)

Specifying Tables and Views

The syntax rules for specifying table names in queries also apply to views.

To select data from a table you own, specify the name of the table. To select data from a table you do not own, specify *schema.table*, where *schema* is the name of the user that owns the table. However, if the table is owned by the DBA, the schema qualifier is not required. You must have the appropriate permissions to access the table (or view) granted by the owner.

A *correlation name* can be specified for any table in the FROM clause. A correlation name is an alias (or alternate name) for the table. For example:

```
select... from employees e, managers m...
```

The preceding example assigns the correlation name "e" to the employees table and "m" to the managers table. Correlation names are useful for abbreviating long table names and for joining a table to itself.

If you assign a correlation name to a table, you must refer to the table using the correlation name. For example:

Correct:

```
select e.name, m.name
from employees e, managers m...
```

Incorrect:

```
select employees.name, managers.name
from employees e, managers m...
```

WHERE Clause

The WHERE clause specifies criteria that restrict the contents of the results table. You can test for simple relationships or, using subselects, for relationships between a column and a set of columns.

Simple WHERE Clauses

Using a simple WHERE clause, the contents of the results table can be restricted, as follows:

Comparisons:

```
SELECT ename FROM employees
       WHERE manager = 'Jones';
SELECT ename FROM employees
       WHERE salary > 50000;
```

Ranges:

```
SELECT ordnum FROM orders
       WHERE odate BETWEEN date('jan-01-1999') AND
       date('today');
```

Set membership:

```
SELECT * FROM orders
       WHERE partno IN ('123-45', '678-90');
```

Pattern matching:

```
SELECT * FROM employees
       WHERE ename LIKE 'A%';
```

Nulls:

```
SELECT ename FROM employees
       WHERE edept IS NULL;
```

Combined restrictions using logical operators:

```
SELECT ename FROM employees
       WHERE edept IS NULL AND
       hiredate = date('today');
```

Note: Aggregate functions cannot appear anywhere in a WHERE clause.

Joins

Joins combine information from multiple tables and views into a single result table, according to column relationships specified in the WHERE clause.

For example, given the following two tables:

Employee Table

ename	edeptno
Benny Barth	10
Dean Reilly	11
Rudy Salvini	99
Tom Hart	123

Department Table

ddeptno	dname
10	Lumber
11	Sales
99	Accounting
123	Finance

The following query joins the two tables on the relationship of equality between values in the edeptno and ddeptno columns. The result is a list of employees and the names of the departments in which they work:

```
SELECT ename, dname FROM employees, departments
       WHERE edeptno = ddeptno;
```

A table can be joined to itself using correlation names; this is useful when listing hierarchical information. For example, the following query displays the name of each employee and the name of the manager for each employee.

```
SELECT e.ename, m.ename
       FROM employees e, employees m
       WHERE e.eno = m.eno
```

Tables can be joined on any number of related columns. The data types of the join columns must be comparable.

Outer Joins

Data can be combined from two or more tables to produce an intermediate results table using an outer join.

Note: Outer join functionality is available **only** if OUTER_JOIN is set to Y in the iidbcapabilities table.

Note: Outer joins specified in the FROM clause are not the same as joins specified in the WHERE clause: the FROM clause specifies sources of data, while the WHERE clause specifies restrictions to be applied to the sources of data to produce the results table.

Outer joins are specified in the FROM clause, using the following syntax:

source join_type **join** *source*
on *search_condition*

where:

- The *source* parameter is the table, view, or outer join where the data for the left or right side of the join originates.
- The *join_type* parameter specifies INNER, LEFT, RIGHT, or FULL outer join. The default join type is INNER.
- The *search_condition* is a valid restriction, subject to the rules for the WHERE clause. The search condition must not include aggregate functions or subselects.

Think of an outer join is as the union of two SELECT statements: the first query returns rows that fulfill the join condition, and the second returns nulls for rows that do not.

There are three types of outer joins:

- **Left outer join** - Returns all values from the left source
- **Right outer join** - Returns all values from the right source
- **Full outer join** - Returns all values from both sources

Note: Right and left joins are symmetrical: (table1 right-join table2) returns the same results as (table2 left-join table1).

By default, joins are evaluated left to right. To override the default order of evaluation, use parentheses.

A source can itself be an outer join, and the results of joins can be joined with the results of other joins, as illustrated in the following pseudocode:

```
(A join B) join (C join D)
```

```
The placement of restrictions is important in obtaining correct results. For
example:

A join B on cond1 and cond2
```

does not return the same results as:

```
A join B on cond1 where cond2
```

In the first example, the restriction determines which rows in the join result table will be assigned null values; in the second example, the restriction determines which rows will be omitted from the result table.

The following example uses an outer join in the FROM clause to display all employees along with the name of their department, if any:

```
select e.ename, d.dname from
(employees e left join departments d
                on e.edept = d.ddept);
```

Join Relationships

The simple joins illustrated in the two preceding examples depend on equal values in the join columns. This type of join is called an *equijoin*. Other types of relationships can be specified in a join. For example, the following query lists salespersons who have met or exceeded their sales quota:

```
SELECT s.name, s.sales_ytd
       FROM sales s, quotas q
       WHERE s.empnum = d.empnum AND
              s.sales_ytd >= d.quota;
```

Subselects

Subselects (also known as subqueries) are SELECT statements placed in a WHERE or HAVING clause; the results returned by the subselect are used to evaluate the conditions specified in the WHERE or HAVING clause. Subselects are also referred to as *subqueries*.

Subselects must return a single column, and cannot include an ORDER BY or UNION clause.

The following example uses a subselect to display all employees whose salary is above the average salary:

```
SELECT * FROM employees WHERE salary >
       (SELECT avg(salary) FROM employees);
```

In the preceding example, the subselect returns a single value: the average salary. Subselects can also return sets of values. For example, the following query returns all employees in all departments managed by Barth.

```
SELECT ename FROM employees WHERE edept IN
       (SELECT ddept FROM departments
              WHERE dmgr = 'Barth');
```

For details about the operators used in conjunction with subqueries, see Predicates (see page 92).

Order By Clause

The ORDER BY clause specifies the columns on which the results table is to be sorted. Columns in the ORDER BY clause can be specified using either the column name or a number corresponding to the position of the column in the FROM clause. (You must specify unnamed result columns using a number.) In a union select, use numbers to specify the columns in the ORDER BY clause; column names cannot be used.

For example, if the employees table contains the following data:

ename	edept	emanager
Murtagh	Shipping	Myron
Obidinski	Lumber	Myron
Reilly	Finance	Costello
Barth	Lumber	Myron
Karol	Editorial	Costello

ename	edept	emanager
Smith	Shipping	Myron
Loram	Editorial	Costello
Delore	Finance	Costello
Kugel	Food prep	Snowden

then this query:

```
select emanager, ename, edept from employees
order by emanager, edept, ename
```

produces this list of managers, the departments they manage, and the employees in each department:

Manager	Department	Employee
Costello	Editorial	Karol
Costello	Editorial	Loram
Costello	Finance	Delore
Costello	Finance	Reilly
Myron	Lumber	Barth
Myron	Lumber	Obidinski
Myron	Shipping	Murtagh
Myron	Shipping	Smith
Snowden	Food prep	Kugel

and this query:

```
select ename, edept, emanager from employees
order by ename
```

produces this alphabetized employee list:

Employee	Department	Manager
Barth	Lumber	Myron
Delore	Finance	Costello
Karol	Editorial	Costello
Kugel	food prep	Snowden

Employee	Department	Manager
Loram	Editorial	Costello
Murtagh	Shipping	Myron
Obidinski	Lumber	Myron
Reilly	Finance	Costello
Smith	Shipping	Myron

To display result columns sorted in descending order (numeric or alphabetic), specify ORDER BY *columnname* desc. For example, to display the employees in each department from oldest to youngest:

```
select edept, ename, eage from employees
order by edept, eage desc;
```

If a nullable column is specified in the order by clause, nulls are sorted to the beginning or end of the results table, depending on the host DBMS.

Note: If the ORDER BY clause is omitted, the order of the rows in the results table is not guaranteed by the DBMS. In particular, the order of the rows in the results table is not guaranteed to have any relationship to the source tables' storage structure or key structure.

Group By Clause

The GROUP BY clause combines results for identical values in a column. This clause is typically used in conjunction with aggregate functions to generate a single figure for each unique value in a column.

For example, to obtain the number of orders for each part number in the orders table:

```
select partno, count(*) from orders
group by partno;
```

The preceding query returns one row for each part number in the orders table, even though there may be many orders for the same part number.

Nulls are used to represent unknown data, and two nulls are typically not considered equal in OpenSQL comparisons. However, the GROUP BY clause treats nulls as equal and returns a single row for nulls in a grouped column.

Grouping can be performed on multiple columns. For example, to display the number of orders for each part placed each day:

```
select odate, partno, count(*) from orders
group by odate, partno;
```

If the GROUP BY clause is specified, all columns in the SELECT clause must be specified in the GROUP BY clause or be aggregate functions.

Having Clause

The HAVING clause filters the results of the GROUP BY clause, in the same way the WHERE clause filters the results of the SELECT...FROM clauses. The HAVING clause uses the same restriction operators as the WHERE clause.

For example, to return the number of orders placed today for each part:

```
select odate, partno, count(*) from orders
group by odate, partno
having odate = date('today');
```

UNION Clause

The UNION clause combines the results of SELECT statements into a single result table. For example, to list all employees in the table of active employees plus those in the table of retired employees:

```
SELECT ename FROM active_emps
UNION
SELECT ename FROM retired_emps;
```

By default, the UNION clause eliminates any duplicate rows in the result table. To retain duplicates, specify UNION ALL. Any number of SELECT statements can be combined using the UNION clause, and both UNION and UNION ALL can be used when combining multiple tables.

Unions are subject to the following restrictions:

- The SELECT statements must return the same number of columns.
- The columns returned by the SELECT statements must correspond in order and data type, although the column names do not have to be identical.
- The SELECT statements cannot include individual ORDER BY clauses.

To sort the result table, specify the ORDER BY clause following the last SELECT statement. The result columns returned by a union are named according to the first SELECT statement.

By default, unions are evaluated from left to right. To specify a different order of evaluation, use parentheses.

Any number of SELECT statements can be combined using the UNION clause. There is a maximum of 126 tables allowed in any query.

Note: The maximum number of tables referenced in a single query is dependent on the host DBMS. The 126 maximum listed here is for the Ingres DBMS; other DBMSs supported by Enterprise Access and EDBC may have a higher or lower limit.

Query Evaluation

The logic applied to the evaluation of SELECT statements, as described here, does not precisely reflect how the DBMS Server evaluates your query to determine the most efficient way to return results. However, by applying this logic to your queries and data, the results of your queries can be anticipated.

1. **Evaluate the FROM clause.** Combine all the sources specified in the FROM clause to create a *Cartesian product* (a table composed of all the rows and columns of the sources). If joins are specified, evaluate each join to obtain its results table, combine it with the other sources in the FROM clause. If SELECT DISTINCT is specified, discard duplicate rows.
2. **Apply the WHERE clause.** Discard rows in the result table that do not fulfill the restrictions specified in the WHERE clause.
3. **Apply the GROUP BY clause.** Group results according to the columns specified in the GROUP BY clause.
4. **Apply the HAVING clause.** Discard rows in the result table that do not fulfill the restrictions specified in the HAVING clause.
5. **Evaluate the SELECT clause.** Discard columns that are not specified in the SELECT clause. (In case of SELECT first n… UNION SELECT …., the first n rows of the result from union are chosen.)
6. **Perform any unions**. Combine result tables as specified in the UNION clause. (In case of SELECT first n… UNION SELECT …., the first n rows of the result from union are chosen.)
7. **Apply the ORDER BY clause.** Sort the result rows as specified.

Examples: Select (interactive)

1. Find all employees who make more than their managers. This example illustrates the use of correlation names.

```
select e.ename
from employee e, dept, employee m
where e.dept = dept.dno and dept.mgr = m.eno
and e.salary > m.salary;
```

2. Select all information for employees that have salaries above the average salary.

```
select * from employee
where salary > (select avg(salary) from employee);
```

3. Select employee information sorted by department and, within department, by name.

```
select e.ename, d.dname from employee e, dept d
where e.dept = d.dno
order by dname, ename;
```

4. Select lab samples analyzed by lab #12 from both production and archive tables.

```
select * from samples s
here s.lab = 12

union
select * from archived_samples s
where s.lab = 12
```

Select (embedded)

Valid in: ESQL

The embedded SELECT statement retrieves values from the database.

Values are returned from tables to host language variables in an embedded OpenSQL program.

For details about the various clauses of the SELECT statement, see Select (interactive) (see page 290).

Syntax

The SELECT (embedded) statement has the following format:

Non-cursor version:

```
EXEC SQL [REPEATED] SELECT [ALLl | DISTINCT]
              * | result_expression{, result_expression}
              INTO variable[:indicator_var] {, variable[:indicator_var]}
              [FROM from_source {, from_source}
              [WHERE search_condition]
              [GROUP BY column {, column}]
              [HAVING search_condition]]
              [UNION  [ALL] full_select]
              [ORDER BY result_column [ASC | DESC]
                           {, result_column [ASC | DESC]}]
[EXEC SQL BEGIN;
              program code;
EXEC SQL END;]
```

Cursor version (embedded in a DECLARE CURSOR statement):

```
SELECT [ALL|DISTINCT]
       * | result_expression {, result_expression}
       [FROM from_source {, from_source}
       [WHERE search_condition]
       [GROUP BY column {, column}]
       [HAVING search_condition]
       [UNION [ALL] full_select]
       [ORDER BY result_column [ASC|DESC]
              {, result_column [ASC|DESC]}]
```

where *result_expression* is one of the following:

- [*schema*.]*table_name*.*

  Selects all columns

- [[*schema*.]*table_name*.]*column_name* AS *result_column*

  Selects one column

- *expression* AS *result_column*

Non-Cursor Select

The non-cursor version of the embedded SELECT statement can be used to retrieve a single row or a set of rows from the database.

If the optional begin-end block syntax is not used, the embedded SELECT statement can retrieve only one row from the database. This kind of SELECT statement is called the *singleton* select and is compatible with the ANSI standard. If the singleton select does try to retrieve more than one row, an error occurs and the result variables hold information from the first row.

For example, the following example retrieves a single row from the database:

```
EXEC SQL SELECT ename, sal
       INTO :ename, :sal
       FROM employee
       WHERE eno = :eno;
```

Select Loops

A select loop can be used to read a table and process its rows individually. When a program needs to read a table without issuing any other database statements during the retrieval (such as for report generation), use a select loop. In other cases, such as when database updates are required, or when other tables need to be browsed while the current retrieval is in progress, use a cursor.

The BEGIN-END statements delimit the statements in the select loop. The code is executed once for each row as it is returned from the database. Statements cannot be placed between the SELECT statement and the BEGIN statement.

Within the select loop, no other statements that access the database can be issued. This will cause a runtime error. To see how to manipulate and update rows and tables within the database while data is being retrieved, see Data Manipulation with Cursors (see page 114) in the chapter "Embedded OpenSQL."

However, if your program is connected to multiple database sessions, queries can be issued from within the select loop by switching to another session. To return to the outer select loop, switch back to the session in which the SELECT statement was issued.

To avoid preprocessor errors, the nested queries cannot be within the syntactic scope of the loop but must be referenced by a subroutine call or some form of a goto statement. For more information about multiple sessions, see the chapter "OpenSQL Features."

There are two ways to terminate the select loop: run it to completion or issue the endselect statement. A host language go to statement cannot be used to exit or return to the select loop.

To terminate a select loop before all rows are retrieved the application must issue the ENDSELECT statement. The endselect statement must be syntactically within the BEGIN-END block that delimits the select loop. For more information, see Endselect (see page 244).

The following example retrieves a set of rows from the database:

```
exec sql select ename, sal, eno
into :ename, :sal, :eno
from employee
order by eno;
exec sql begin;
browse data;
if error condition then
  exec sql endselect;
end if;
exec sql end;
```

Retrieving Values into Host Language Variables

The INTO clause specifies the host program variables into which the values retrieved by the select are loaded. There must be a one-to-one correspondence between expressions in the SELECT clause and the variables in the INTO clause. If the statement does not retrieve any rows, the variables are not modified. If the number of values retrieved from the database is different from the number of columns, an error is issued and the sqlwarn3 variable of the SQLCA is assigned the value 'W'. Each result variable may have an indicator variable for null data.

Host language variables can be used as expressions in the SELECT clause and the *search_condition*, in addition to their use in the INTO clause. Variables used in *search_conditions* must denote constant values and cannot represent names of database columns or include any operators. Host string variables can also substitute for the complete search condition.

Host Language Variables in Union Clause

When SELECT statements are combined using the UNION clause, the INTO clause must appear only after the first list of select result expressions, because all result rows of the SELECT statements that are combined by the UNION clause must be identical. The following example shows the correct use of host language variables in a union; result variables are specified only for the first SELECT statement:

```
EXEC SQL SELECT ename, enumber
               INTO :name, :number
               FROM employee
       UNION
       SELECT dname, dnumber
              FROM directors
              WHERE dnumber < 100;
```

Repeated Queries

To reduce the overhead required to repeatedly execute a SELECT query statement, specify the query as a REPEATED query. For repeated queries, the DBMS Server saves the query execution plan after the first time the query is executed. This can significantly improve the performance of subsequent executions of the same select.

If your application needs to be able to change the search conditions, dynamically constructed search conditions cannot be used with repeated queries. The saved execution plan is based on the initial value of the search condition and subsequent changes are ignored.

Cursor Select

The cursor SELECT statement is specified as part of a DECLARE CURSOR statement. Within the DECLARE CURSOR statement, the SELECT statement is not preceded by EXEC SQL. The cursor SELECT statement specifies the data to be retrieved by the cursor. When executed, the DECLARE CURSOR statement does not perform the retrieval-the retrieval occurs when the cursor is opened. If the cursor is declared for update, the select cannot see more than one table, cannot see a view and cannot include a GROUP BY, HAVING, ORDER BY, or UNION clause.

The cursor select can return multiple rows, because the cursor provides the means to process and update retrieved rows one at a time. The correlation of expressions to host language variables takes place with the FETCH statement, so the cursor select does not include an INTO clause. The rules for the remaining clauses are the same as in the non-cursor select.

Error Handling for Embedded SELECT

If the SELECT statement retrieves no rows, the SQLCA variable sqlcode is set to 100. The number of rows returned from the database is in the SQLCA variable sqlerrd(3). In a select loop, if the ENDSELECT statement was issued, sqlerrd(3) contains the number of rows retrieved before ENDSELECT was issued.

Embedded Usage

Host language variables can be used as expressions in the SELECT clause and the *search_condition*s. Variables used in *search_conditions* must specify constant values and cannot represent names of database columns or include any operators. Host string variables can also substitute for the complete search condition.

Examples: Select (embedded)

The following examples illustrate the non-cursor SELECT statement:

1. Retrieve the name and salary of an employee. Drop locks by committing the following transaction.

```
EXEC SQL SELECT ename, sal
    into :namevar, :salvar
    from employee
    where eno = :numvar;
exec sql commit;
```

2. Select all columns in a row into a host language variable structure. (The emprec structure has members that correspond in name and type to columns of the employee table.)

```
EXEC SQL SELECT *
    into :emprec
    from employee
    where eno = 23;
```

3. Select a constant into a variable.

```
EXEC SQL SELECT 'Name: ', ename
    into :title, :ename
    from employee
    where eno >= 148 and age = :age;
```

4. Select the row in the employee table whose number and name correspond to the variables, numvar and namevar. The columns are selected into a host structure called emprec. Because this statement is issued many times (in a subprogram, perhaps), it is formulated as a repeat query.

```
EXEC SQL REPEATED SELECT *
    INTO :emprec
    FROM employee
    WHERE eno = :numvar AND ename = :namevar;
```

5. Example of a select loop: insert new employees, and select all employees and generate a report. If an error occurs during the process, end the retrieval and back out the changes. No database statements are allowed inside the select loop (BEGIN-END block).

```
error = 0;
EXEC SQL INSERT INTO employee
    SELECT * FROM newhires;
EXEC SQL SELECT eno, ename, eage, esal, dname
        INTO :eno, :ename, :eage, :esal, :dname
    FROM employee e, dept d
    WHERE e.edept = d.deptno
    GROUP BY ename, dname
EXEC SQL BEGIN;
    generate report of information;
    if error condition then
        error = 1;
        exec sql endselect;
    end if;
EXEC SQL END;
/*
** Control transferred here by completing the
** retrieval or because the endselect statement
```

```
** was issued.
*/
if error = 1
    print 'Error encountered after row',
        sqlca.sqlerrd(3);
    exec sql rollback;
else
    print 'Successful addition and reporting';
    exec sql commit;
end if;
```

6. The following SELECT statement uses a string variable to substitute for the complete search condition. The variable *search_condition* is constructed from an interactive forms application in query mode, and during the select loop the employees who satisfy the qualification are displayed.

```
run forms in query mode;
construct search_condition of employees;

EXEC SQL SELECT  *
    INTO :emprec
    FROM employee
    WHERE :search_condition;
EXEC SQL BEGIN;
    load emprec into a table field;
EXEC SQL END;
display table field for browsing;
```

7. This example illustrates session switching inside a select loop. The main program processes sales orders and calls the new_customer subroutine for every new customer.

   The main program:

```
...
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;

/* Include output of dclgen for declaration of record order_rec */
EXEC SQL INCLUDE 'decls';
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT customers session 1;
EXEC SQL CONNECT sales session 2;
...

EXEC SQL SELECT * INTO :order_rec FROM orders;
EXEC SQL BEGIN;

if (order_rec.new_customer = 1) then

        call new_customer(order_rec);

    endif

    process order;

EXEC SQL END;
...

EXEC SQL DISCONNECT;
```

The subroutine, new_customer, which is from the select loop, contains the session switch:

```
subroutine new_customer(record order_rec)
begin;

EXEC SQL SET_SQL(session = 1);
    EXEC SQL INSERT INTO accounts
        VALUES (:order_rec);

process any errors;

EXEC SQL SET_SQL(session = 2);

/* Reset status information before resuming select loop */

sqlca.sqlcode = 0;
    sqlca.sqlwarn.sqlwarn0 = ' ';

end subroutine;
```

Set

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The SET statement specifies a runtime option for the current session. The selected option remains in effect until the session is terminated or the option is changed using another SET statement.

Syntax

The SET statement has the following format:

```
[EXEC SQL] SET AUTOCOMMIT ON|OFF
```

AUTOCOMMIT ON

Causes an implicit commit to occur after every successfully executed query.

AUTOCOMMIT OFF

Causes an explicit COMMIT statement to be required to commit a transaction.

Default: AUTOCOMMIT OFF

The SET AUTOCOMMIT statement cannot be issued within a transaction. For a description of OpenSQL transaction behavior, see the chapter "OpenSQL Features."

Note: For additional SET option support for your particular gateway, see the Enterprise Access or EDBC documentation.

Set_sql

Valid in: ESQL

The SET_SQL statement specifies runtime options for the current session.

The SET_SQL statement can switch sessions in a multiple session application, specify the type of DBMS error to be returned to an application, change the default behavior when a connection error is experienced, set trace functions, and set other session characteristics.

SET_SQL can be used to override II_EMBED_SET. For more information about II_EMBED_SET, see the *Ingres System Administrator Guide*.

Syntax

The SET_SQL statement has the following format:

```
EXEC SQL SET_SQL (object = value {, object = value})
```

The valid objects and values for the SET_SQL statement are as follows:

Object	Data Type	Description
dbeventdisplay	integer	Enables or disables the display of events as they are queued to an application. Specify 1 to enable display, 0 to disable display.
dbeventhandler	function pointer	Specifies a user-defined routine to be called when an event notification is queued to an application. The event handler must be specified as a function pointer.
dbmserror	integer	Sets the value returned by the inquire_sql(dbmserror) statement. For details about the values returned by the inquire_sql(dbmserror) statement, see Local and Generic Errors (see page 166).
errorhandler	function pointer	Specifies a user-defined routine to be called when an OpenSQL error occurs in an embedded application. The error handler must be specified as a function pointer.
errorno	integer	Sets the value returned by the inquire_sql(errorno) statement. For details about the values returned by the inquire_sql(errorno) statement, see Local and Generic Errors (see page 166).

Object	Data Type	Description
errortype	character string	Specifies the type of error number returned to errorno and sqlcode. Value can be either genericerror, specifying generic error numbers, or dbmserror, specifying local DBMS error numbers. Generic error numbers are returned by default. For information about local and generic errors, see Local and Generic Errors (see page 166).
gcafile	character string	Specifies an alternate text file to which OpenSQL writes GCA information. The default file name is "iiprtgca.log". To enable this feature, use the set_sql printgca option. If a directory or path specification is omitted, the file is created in the current default directory.
printgca	integer	Turns the printgca debugging feature on or off. Printgca prints all communications (GCA) messages from the application as it executes (by default, to the file "iiprtgca.log" in the current directory). Value can be either 1, to turn the feature on, or 0, to turn the feature off.
printqry	integer	Turns the printqry debugging feature on or off. Printqry prints all query text and timing information from the application as it executes (by default to the file "iiprtqry.log" in the current directory). Value can be either 1, to turn the feature on, or 0, to turn the feature off.
printtrace	integer	Enable/disable trapping of DBMS trace messages to a text file (by default, "iiprttrc.log"). Specify 1 to enable trapping of trace output, 0 to disable trapping.
programquit	integer	Specifies whether OpenSQL aborts on one of the following errors: An application issues a query, but is not connected to a database. The Enterprise Access product or DBMS fails. Communications services fail. Specify 1 to abort on these conditions, 0 to continue.
qryfile	character string	Specifies an alternate text file to which OpenSQL writes query information. The default file name is "iiprtqry.log". To enable this feature, use the set_sql printqry option. If a directory or path specification is omitted, the file is created in the current default directory.
savequery	integer	Enables/disables saving of the text of the last query issued. Specify 1 to enable, 0 to disable. To obtain the

Object	Data Type	Description
		text of the last query, issue the inquire_sql(querytext) statement. To determine whether saving is enabled, use the inquire_sql(savequery) statement.
session	integer	Sets the current session. Value can be any session identifier associated with an open session in the application.
tracefile	character string	Specifies an alternate text file to which OpenSQL writes tracepoint information. The default file name is "iiprttrc.log". To enable this feature, use the set_sql printtrace option. If a directory or path specification is omitted, the file is created in the current default directory.

Update

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The UPDATE statement updates column values in a table.

Syntax

The UPDATE statement has the following format:

Interactive version:

```
UPDATE table_name
              SET column_name = expression {, column_name = expression}
              [WHERE search_condition]
```

Embedded non-cursor version:

```
EXEC SQL [REPEATED] UPDATE table_name
              SET column = expression {, column = expression}
              [WHERE search_condition]
```

Embedded cursor version:

```
EXEC SQL UPDATE table_name
              SET column = expression {, column = expression}
              WHERE CURRENT OF cursor_name;
```

Description

The UPDATE statement replaces the values of the specified columns by the values of the specified expressions for all rows of the table that satisfy the *search_condition*. For a discussion of search conditions, see the chapter "Elements of OpenSQL Statements."

The expressions in the SET clause can use constants or expressions involving column values from the table being updated. The data type of the column must agree with the data type of the value being assigned to it. To place a null in a nullable column, use the null constant.

If an update to a row would violate an integrity constraint defined on the table, that row remains unchanged.

If a subselect is specified, the subselect must not select rows from the table in which you are updating rows.

Embedded Usage

Host language variables can only be used within expressions in the set clause and the *search_condition*. (Variables used in *search_condition* must denote constant values and cannot represent names of database columns or include any operators.) A host string variable can be used to specify the complete search condition.

If the update did not update any rows, the sqlcode of the SQLCA is set to 100. If the update succeeded, the sqlerrd(3) of the SQLCA contains the number of rows updated by the statement.

To formulate the non-cursor update as a repeated query, specify the keyword repeated. The repeated keyword directs OpenSQL to encode the update and save its execution plan when the update is first executed. This encoding can improve the performance of subsequent executions of the same update. The repeated keyword is available only for non-cursor updates, and is ignored if used with the cursor or dynamic versions.

If your statement includes a dynamically constructed *search_condition*, that is, if the complete *search_condition* is specified by a host string variable, do not use the repeated option if you intend to change the *search_condition* after the statement's initial execution. The saved execution plan is based on the initial value of the *search_condition* and any changes to *search_condition* would be ignored. This rule does not apply to simple variables used in a *search_condition*.

Cursor Updates

The cursor version of UPDATE is similar to the interactive update, except for the where clause. The WHERE clause, required in the cursor update, specifies that the update occur to the row the cursor currently points to. If the cursor is not pointing to a row, as would be the case immediately after an OPEN or DELETE statement, a runtime error message is generated indicating that a fetch must first be performed. If the row the cursor is pointing to has been deleted from the underlying database table (as the result, for example, of a non-cursor delete), no row is updated and the sqlcode is set to 100. Following a cursor update, the cursor continues to point to the same row.

Two cursor updates not separated by a fetch may cause the same row to be updated twice, or may cause an error, depending on the host DBMS.

In performing a cursor update, make sure that certain conditions are met:

- A cursor must be declared in the same file in which any UPDATE statement referencing that cursor appears. This applies also to any cursor referenced in a dynamic UPDATE statement string.
- A cursor name in a dynamic UPDATE statement must be unique among all open cursors in the current transaction.
- The cursor stipulated in the update must be open before the statement is executed.
- The UPDATE statement and the FROM clause in the cursor's declaration must refer to the same database table.
- The columns in the SET clause must have been declared for update at the time the cursor was declared (see the declare cursor statement).
- Host language variables can be used only for the cursor names or for expressions in the SET clause.

The COMMIT and ROLLBACK statements close all open cursors. A common programming error is to update the current row of a cursor, commit the change, and then attempt to loop and repeat the process. The commit closes the cursor, and subsequent fetches will fail.

Examples: Update

1. Give all employees who work for Smith a 10% raise.

```
update emp
    set salary = .1 * salary
    where dept in
        (select dno
        from dept
        where mgr in
            (select eno
            from emp
            where ename like '%Smith'));
```

2. Set all salaried people who work for Smith to null.

```
update emp
  set salary = null
  where dept in
      (select dno
          from dept
          where mgr in
              (select eno
              from emp
              where ename like '%Smith'));
```

Whenever

Valid in: ESQL

The WHENEVER statement enables your application to handle error and exception conditions arising from embedded OpenSQL database statements. The WHENEVER statement stipulates that a specified action be performed when a specified condition occurs. The WHENEVER statement detects conditions by checking SQLCA variables, so an SQLCA must be included in your program before you issue the WHENEVER statement.

After a WHENEVER has been declared, it remains in effect until another WHENEVER is specified for the same condition. Since WHENEVER is a declarative and not an executable statement, its physical location in the program's source code, rather than its sequence in the program's execution, determines its scope.

WHENEVER statements can be repeated for the same condition and can appear anywhere after the INCLUDE SQLCA statement.

Syntax

The WHENEVER statement has the following format:

```
exec sql WHENEVER condition action
```

condition

Defines the condition that triggers the action. The *condition* can be any of the following:

SQLWARNING

Indicates that the last embedded SQL database statement produced a warning condition. The sqlwarn0 variable of the SQLCA is set to W.

SQLERROR

Indicates that an error occurred as a result of the last embedded OpenSQL database statement. The sqlcode of the SQLCA is set to a negative number.

NOT FOUND

Indicates that a SELECT, FETCH, UPDATE, DELETE, INSERT, COPY, CREATE INDEX, or CREATE AS...SELECT statement affected no rows. The sqlcode variable of the SQLCA is set to 100.

DBEVENT

Indicates that an event has been raised. The sqlcode variable of the SQLCA is set to 710. This condition occurs only for events that the application is registered to receive.

action

Specifies the *action*. Valid actions include:

CONTINUE

No action is taken when the condition occurs. The program continues with the next executable statement. If a fatal error occurs, an error message is printed and the program aborts.

STOP

Displays an error message and terminates when the condition occurs. If the program is connected to a database when the condition occurs, the program disconnects from the database without committing pending updates. The stop action cannot be specified for the not found condition.

GOTO *label*

Transfers control to the specified label (same as a host language go to statement). The *label* (or paragraph name in COBOL) must be specified using the rules of your host language. (The keyword GOTO can also be specified as GO TO).

CALL *procedure*

Calls the specified procedure (in COBOL, performs the specified paragraph). The procedure must be specified according to the conventions of the host language. No arguments can be passed to the procedure. To direct the program to print any error or warning message and continues with the next statement, specify call SQLPRINT. (SQLPRINT is a procedure provided by Ingres, not a user-written procedure.)

If your program does not include an SQLCA (and therefore no WHENEVER statements), OpenSQL displays all errors. If your program includes an SQLCA, OpenSQL continues execution (and does not display errors) for all conditions for which you do not issue a WHENEVER statement. To override the continue default and direct OpenSQL to display errors and messages, set II_EMBED_SET to sqlprint.

The program's condition is automatically checked after each embedded OpenSQL database statement. If one of the conditions has become true, the *action* specified for that condition is taken. If the *action* is goto, then the label must be within the scope of the statements affected by the WHENEVER statement at compile time.

An *action* specified for a *condition* affects all subsequent embedded OpenSQL source statements until another WHENEVER is encountered for that condition.

The embedded SQL preprocessor does not generate any code for the WHENEVER statement. Therefore, in a language that does not allow empty control blocks, (for example, COBOL, which does not allow empty IF blocks), the WHENEVER statement should not be the only statement in the block.

Be careful to avoid coding potentially infinite loops with WHENEVER statements. Within a sequence of statements functioning as an error handling block for a particular condition, the first statement should be a whenever continue that turns off the action. For example, consider the following program fragment:

```
exec sql whenever sqlerror goto error_label;
exec sql create table worktable
(workid integer2, workstats varchar(15));
 ...

process data;
  ...

error_label:
exec sql whenever sqlerror continue;
exec sql drop worktable;
exec sql disconnect;
  ...
```

If the error handling block did not specify continue for condition sqlerror and the DROP statement caused an error, at runtime the program would infinitely loop between the DROP statement and the label, error_label.

Host language variables cannot be used in a WHENEVER statement. This statement must be terminated according to the rules of your host language.

Examples: Whenever

1. During program development, print all errors and continue with next statement.

```
exec sql whenever sqlerror call sqlprint;
```

2. During database cursor manipulation, close the cursor when no more rows are retrieved.

```
exec sql open cursor1;
    exec sql whenever not found goto close_cursor;

    loop until whenever not found is true
        exec sql fetch cursor1
            into :var1, :var2;
        print and process the results;
    end loop;

    close_cursor:
        exec sql whenever not found continue;
        exec sql close cursor1;
```

3. Stop program upon detecting an error or warning condition.

```
exec sql whenever sqlerror stop;
    exec sql whenever sqlwarning stop;
```

4. Reset WHENEVER actions to default within an error handling block.

```
error_handle:
        exec sql whenever sqlerror continue;
        exec sql whenever sqlwarning continue;
        exec sql whenever not found continue;
    ...
    handle cleanup;
    ...
```

5. Always confirm that the connect statement succeeded before continuing.

```
exec sql whenever sqlerror stop;
    exec sql connect :dbname;
    exec sql whenever sqlerror continue;
```

Chapter 9: Extended Statements

This section contains the following topics:

This chapter lists statements and extensions that may be available in OpenSQL.

If these statements and extensions are supported, the following row in is the iidbcapabilities catalog:

CAP_CAPABILITIES	CAP_VALUE
SQL92_COMPLIANCE	ENTRY

If the statements and extensions are not supported, the cap_value column contains NONE.

Create Schema

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE SCHEMA statement creates a named collection of database objects.

Syntax

The CREATE SCHEMA statement has the following format:

```
[EXEC SQL] CREATE SCHEMA AUTHORIZATION schema_name
             [object_definition {object_definition}];
```

schema_name

Specifies the effective user for the session issuing the CREATE SCHEMA statement.

object_definition

Is a CREATE TABLE, CREATE VIEW, or GRANT statement.

Description

The CREATE SCHEMA statement creates a named collection of database objects (tables, views and privileges). The *schema_name* parameter must be the same as the effective user for the session issuing the create schema statement. All objects specified in the CREATE SCHEMA statement are owned by that user. You cannot create a schema for another user. Each user has one schema per database.

The statements in the CREATE SCHEMA statement must not be separated by semicolon delimiters. However, the CREATE SCHEMA statement must be terminated with a semicolon following the last object definition statement (CREATE TABLE, CREATE VIEW, or GRANT).

If object definitions are omitted, an empty schema is created. For details about the statements used to create tables and privileges, see Create Table (extended) (see page 328) and Grant (see page 347) respectively. If an error occurs within the CREATE SCHEMA statement, the entire statement is rolled back. If you issue a CREATE SCHEMA specifying an existing schema (*schema_name*), OpenSQL issues an error.

To add objects to your schema, issue the required create statements outside of a CREATE SCHEMA statement. If no schema exists for your user identifier, one is implicitly created when you create any database object. Thereafter, if you issue a CREATE SCHEMA statement, OpenSQL issues an error.

If, within a CREATE SCHEMA statement, you create tables that have referential constraints, the order of CREATE TABLE statements is not significant. This is unlike the requirements for creating tables with referential constraints outside of a CREATE SCHEMA statement, where the referenced table must exist before a constraint that references it can be created. For details about referential constraints, see Create Table (extended) (see page 328) in this chapter.

Other users can reference objects in your schema if you have granted them the required permissions. To reference an object in a schema other than your own, specify the object name as follows:

schema.object

For example, user harry can select data from user joe's employees table (if joe has granted harry select permission). Harry can issue the following SELECT statement:

```
select lname, fname from joe.employees
  where dname = 'accounting';
```

Restrictions

The following restrictions apply to CREATE TABLE statements within a CREATE SCHEMA statement:

- CREATE TABLE...AS SELECT cannot be used.
- A WITH clause cannot be specified.
- The following data types cannot be used:
  - integer2
  - integer4
  - float4
  - float8
  - date

The only valid WITH clause option for CREATE VIEW statements within a CREATE SCHEMA statement is WITH CHECK OPTION.

Embedded Usage

You cannot use host language variables in an embedded CREATE SCHEMA statement.

Permissions

This statement is available to all users.

Example: Create Schema

Create a schema authorization containing tables, views, and privileges.

```
create schema authorization joe
create table employees(lname character(30) not null,
                       fname character(30) not null,
                       salary decimal,
                       dname character(10)
                          references dept(deptname),
                       primary  key  (lname,  fname)
 create table dept(deptname character(10)
                      not null unique,
                   budget decimal,
                   expenses decimal default 0)
 create view mgr (mlname, mfname, mdname) as
  select lname, fname, deptname from employees,dept
  where dname = deptname
grant references(lname, fname)
on table employees to harry;
```

Create Table (extended)

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE TABLE statement creates a base table. The table is owned by the user who issues the statement.

If you use the CREATE TABLE...AS SELECT syntax, the table that you create will contain a subset of the columns and values in existing tables specified by the *subselect*.

Syntax

The CREATE TABLE statement has the following format:

```
[EXEC SQL] CREATE TABLE table_name
              [(column_specification {, column_specification }
               (column_specification {, column_specification }
              [, [CONSTRAINT constraint_name] table_constraint
              {, [CONSTRAINT constraint_name] table_constraint}])
              [WITH with_clause]
```

The CREATE TABLE...AS SELECT statement (which creates a table and load rows from another table) has the following format:

```
[EXEC SQL] CREATE TABLE table_name
              (column_name {, column_name}) AS
                           subselect
                           {UNION [ALL]
                           subselect}
              [WITH with_clause]
```

table_name

Specifies the name of the new table, and must be a valid object name (see page 24).

column_specification

Specifies the characteristics of the column, as described in Column Specifications (see page 331).

subselect

Specifies a SELECT clause, described in detail in Select (see page 355) in this chapter.

table_constraint

Specifies a table constraint (see page 335), as one or more of the following:

```
UNIQUE (column_name {, column_name}) [WITH constraint_with_
clause]

PRIMARY KEY (column_name {, column_name}) [WITH constraint_with_clause]

FOREIGN KEY (column_name {, column_name})
REFERENCES [schema.]table_name [(column_name {, column_name})]
    [WITH constraint_with_clause]
```

WITH *with_clause*

Specifies WITH clause options.

For an overview of the Enterprise Access product WITH clause, see DBMS Extensions (see page 187).

For valid WITH clause options for a specific Enterprise Access, see your Enterprise Access product guide.

For valid WITH clause options for the Ingres DBMS, see the *SQL Reference Guide.*

Column Specifications

The column_specification in a CREATE TABLE statement describes the characteristics of the column.

The column_specification has the following syntax:

```
column_name datatype
[[WITH] DEFAULT default_spec | WITH DEFAULT | NOT DEFAULT]
[WITH NULL | NOT NULL|
[[CONSTRAINT constraint_name] column_constraint
{[CONSTRAINT constraint_name] column_constraint}]
```

where *column_constraint* is one or more of the following:

```
UNIQUE [WITH constraint_with_ clause]

PRIMARY KEY [WITH constraint_with_clause]

REFERENCES[schema.]table_name[(column_name)]
    [WITH constraint_with_clause]
```

column_name

Assigns a valid name to the column.

datatype

Assigns a valid data type to the column. If CREATE TABLE...AS SELECT is specified, the new table takes its column names and formats from the results of the SELECT clause of the *subselect* specified in the AS clause (unless different column names are specified).

[[WITH] DEFAULT *default_spec* | WITH DEFAULT | NOT DEFAULT]

Specifies whether the column is mandatory, as described in Default Clause (see page 332).

WITH NULL | NOT NULL

Specifies whether the column accept nulls, as described in Null Clause (see page 334).

[CONSTRAINT *constraint_name*] *column_constraint*

Specifies checks to be performed on the contents of the column to ensure appropriate data values, as described in Constraints (see page 335).

Default Clause

The WITH | NOT DEFAULT clause in the column specification specifies whether a column requires an entry.

This clause has the following format:

```
[WITH] DEFAULT default_spec | WITH DEFAULT | NOT DEFAULT
```

NOT DEFAULT

Indicates the column requires an entry (is mandatory).

WITH DEFAULT

Indicates that if no value is provided, 0 is inserted for numeric and money columns, or an empty string for character and date columns.

[WITH] DEFAULT *default_spec* | USER | NULL

Inserts the specified default value if the user or program does not provide a value for the column. The default value must be compatible with the data type of the column.

USER

Specifies the session's current user ID as the default value.

NULL

Specifies NULL as the default value for nullable columns.

If the DEFAULT clause is omitted, the column default depends on whether the column is nullable. Nullable columns default to nulls, and non-nullable columns are mandatory.

The following is an example of the DEFAULT option:

```
create table dept( dname  character(10),
  budget         decimal  default 100000.00,
  creation       date     default date('01/01/94'));
```

Restrictions on the Default Value for a Column

The following considerations and restrictions apply when specifying a default value for a column:

- The data type and length of the default value must not conflict with the data type and length of the column.
- The maximum length for a default value is 1500 characters or the declared length of the column, whichever is shorter.
- For fixed length string columns, if the column is wider than the default value, the default value is padded with blanks to the full width of the column.

- For numeric columns that accept fractional values (floating-point and decimal), the decimal point character specified for the default value must match the decimal point character in effect when the value is inserted. To specify the decimal point character, set II_DECIMAL.
- For date columns, the default value must be a valid date specified using the date() function. If the time zone is omitted, the time zone defaults to the time zone of the user inserting the row.

Null Clause

The WITH|NOT NULL clause in the column specification specifies whether a column accepts null values.

This clause has the following format:

```
WITH NULL | NOT NULL
```

WITH NULL

Indicates that the column accepts nulls. If no value is supplied by the user, null is inserted as the default value.

NOT NULL

Indicates that the column does not accept nulls. If the DEFAULT clause is omitted or NOT DEFAULT is specified, the column is mandatory.

With|Not Null and With|Not Default Combinations

The WITH|NOT NULL clause works in combination with the WITH|NOT DEFAULT clause, as follows:

WITH NULL

The column accepts nulls. If no value is provided, a null is inserted.

NOT NULL

The column is mandatory and does not accept nulls, which is typical for primary key columns.

WITH NULL WITH DEFAULT

The column accepts null values. If no value is provided, the default value is inserted.

WITH NULL NOT DEFAULT

The column accepts null values. The user must provide a value (mandatory column).

NOT NULL WITH DEFAULT

The column does not accept nulls. If no value is provided, the default value is inserted. (The specified default value cannot be NULL.)

NOT NULL NOT DEFAULT (or NOT NULL)

The column is mandatory and does not accept nulls, which is typical for primary key columns.

Constraints

To ensure that the contents of columns fulfill your database requirements, specify *constraints*.

Constraints are checked at the end of every statement that modifies the table. If the constraint is violated, an error is returned and the statement is aborted. If the statement is within a multi-statement transaction, the transaction is not aborted.

Note: Constraints are not checked when adding rows to a table using the COPY statement.

Constraints can be specified for individual columns or for the entire table. For details, see Column-Level Constraints and Table-Level Constraints.

The types of constraints are:

- **Unique constraint–**Ensures that a value appears in a column only once. Unique constraints are specified using the UNIQUE option.
- **Check constraint–**Ensures that the contents of a column fulfills user-specified criteria (for example, "salary >0"). Check constraints are specified using the CHECK option.
- **Referential constraint–**Ensures that a value assigned to a column appears in a corresponding column in another table. Referential constraints are specified using the REFERENCES option.
- **Primary key constraint–**Declares one or more columns for use in referential constraints in other tables. Primary keys must be unique.

Unique Constraint

To ensure that no two rows have the same value in a particular column or set of columns, specify UNIQUE NOT NULL.

Note: If a column is specified as UNIQUE, NOT NULL must also be specified.

The following example of a column-level unique constraint ensures that no two departments have the same name:

```
create table dept (dname character(10)
    not null unique, ...);
```

In the preceding example, the unique constraint ensures that no two departments have the same name.

To ensure that the data in a group of columns is unique, specify the unique constraint at the table level (rather than for individual columns). A maximum of 32 columns can be specified in a table-level unique constraint.

The following example of a table-level unique constraint ensures that no two departments in the same location have the same name. The columns are declared not null, as required by the unique constraint:

```
create table depts (dname character(10) not null,
                    dlocation character(10) not null,
 unique (dname, dlocation));
```

Any column or set of columns that is designated as the primary key is implicitly unique and must be specified as NOT NULL. A table can have only one primary key, but can have any number of unique constraints.

Note: Unique constraints may create system indexes that cannot be explicitly dropped by the table owner. These indexes are used to enforce the unique constraint.

Check Constraint

To create conditions that a particular column or set of columns must fulfill, specify a check constraint using the CHECK option. For example, to ensure that salaries are positive numbers:

```
create table emps (name character(25), sal decimal
check (sal > 0));
```

The expression (see page 92) specified in the check constraint must be a Boolean expression.

To specify a check constraint for a group of columns, the check constraint must be specified at the table level (rather than specifying check constraints for individual columns).

The following example of a table-level check constraint ensures that each department has a budget and that expenses do not exceed the budget:

```
create table dept (dname character(10),
  location character(10),
  budget decimal,
  expenses decimal,
 check (budget > 0 and expenses <= budget));
```

Check constraints cannot include the following:

- Subqueries
- Set functions (aggregate functions)
- Dynamic parameters
- Host language variables

Column-level check constraints cannot refer to other columns.

Referential Constraint

To validate an entry against the contents of a column in another table (or another column in the same table), specify a referential constraint using the REFERENCES option. The references option maintains the referential integrity of your tables.

The column-level referential constraint has the following syntax:

```
REFERENCES [schema.] table_name (column_name)[referential actions]
[constraint_with_clause]
```

The following example of a column-level referential constraint ensures that no employee is assigned to a department that is not present in the dept table:

```
CREATE TABLE emp (ename CHAR(10),
edept CHAR(10) REFERENCES dept(dname));
```

The table-level referential constraint has the following syntax, including the FOREIGN KEY... REFERENCES option:

```
FOREIGN KEY (column_name{,column_name})
REFERENCES [schema.] table_name [(column_name{,column_name}][referential actions]
[constraint_with_clause]
```

The following example of a table-level referential constraint verifies the contents of the name and empno columns against the corresponding columns in the emp table to ensure that anyone entered into the table of managers is on file as an employee:

```
CREATE TABLE mgr (name CHAR(10),
        empno CHAR(5),
        ...
FOREIGN KEY (name, empno) REFERENCES emp);
```

The preceding example omits the names of the referenced column; the emp table must have a primary key constraint that specifies the corresponding name and employee number columns.

referential actions

Allow the definition of alternate processing options in the event a referenced row is deleted, or referenced columns are updated when there are existing matching rows. A referential action specifies either an *update rule* or a *delete rule*, or both, in either sequence.

The ON UPDATE and ON DELETE rules have the following syntax:

```
ON UPDATE {CASCADE | SET NULL | RESTRICT | NO ACTION}
```

or

```
ON DELETE {CASCADE | SET NULL | RESTRICT | NO ACTION}
```

ON UPDATE CASCADE

Causes the values of the updated referenced columns to be propagated to the referencing columns of the matching rows of the referencing table.

ON DELETE CASCADE

Specifies that if a delete is attempted on a referenced row that has matching referencing rows, the delete is "cascaded" to the referencing table as well. That is, the matching referencing rows are also deleted. If the referencing table is itself a referenced table in some other referential relationship, the delete rule for that relationship is applied, and so forth. (Because rule types can be mixed in a referential relationship hierarchy, the second delete rule can be different from the first delete rule.) If an error occurs somewhere down the line in a cascaded operation, the original delete fails, and no update is performed.

NO ACTION

Is the default behavior of returning an error upon any attempt to delete or update a referenced row with matching referencing rows.

RESTRICT

Behaves the same as NO ACTION, but returns a different error code. Both options are supported for ANSI SQL compliance.

SET NULL

Causes the referencing columns of the matching rows to be set to the null value (signifying that they do not currently participate in the referential relationship). The columns can be updated later to a non-null values, at which time the resulting row must find a match somewhere in the referenced table.

Example

Here is an example of the delete and update rules:

```
CREATE TABLE employee (empl_no INT NOT NULL)
        emp_name CHAR(20) NOT NULL,
        dept_id CHAR(6) REFERENCES department (dept_id)
        ON DELETE CASCADE ON UPDATE CASCADE,
        mgrno INT REFERENCES employee (empl_no) ON UPDATE CASCADE
        ON DELETE SET NULL);
```

If a department row is deleted, all employees in that department are also deleted. If a department ID is changed in the department table, it is also changed in all referencing employee rows.

If a manager's ID is changed, his employees are changed to match. If the manager is fired, all his employees have mgr_id set to null.

The following considerations apply to the table and column being referenced (the column specified following the keyword references):

- The referenced table must be an existing base table (it cannot be a view).
- The data types of the columns must be comparable.
- You must have references privilege for the referenced columns.
- If the table and column names are specified, the referenced columns must compose a unique or primary key constraint for the referenced table.
- In a table-level referential constraint, if multiple columns are specified, the columns specified for the referencing table must correspond in number, data type, and position to the columns specified for the referenced table, and must compose a unique or primary key constraint for the referenced table.
- If the referenced table is specified and the column name is omitted, the referenced table must have a primary key constraint; the referencing columns are verified against the primary key of the referenced table.

Primary Key Constraint

The primary key constraint is used to denote one or more columns to which other tables refer in referential constraints. A table can have only one primary key; the primary key for a table is implicitly unique and must be declared not null.

This is an example of a primary key constraint and a related referential constraint:

Referenced table:

```
CREATE TABLE partnumbers(partno INT PRIMARY KEY...);
```

Referencing table:

```
create table inventory(ipartno INT...
    FOREIGN KEY (ipartno) REFERENCES partnumbers);
```

In this case, the part numbers in the inventory table are checked against those in the partnumbers table; the referential constraint for the inventory table is a table-level constraint and therefore must specify the FOREIGN KEY clause. The referential constraint for the inventory does not specify the column that is referenced in the partnumbers table. By default, the DBMS checks the column declared as the primary key. For related details, see Referential Constraint.

Column-Level Constraints and Table-Level Constraints

Constraints can be specified for:

- Individual columns as part of the column specification (*column*-level constraints)
- Groups of columns as part of the table definition (*table*-level constraints)

For example:

Column-level constraints:

```
create table mytable(name char(10) not null,
         id integer references idtable(id),
         age integer check (age > 0));
```

Table-level constraints:

```
create table yourtable(firstname char(20) not null,
         lastname char(20) not null,
         unique(firstname, lastname));
```

Note: Multiple column constraints are separated by a space.

Names can be assigned to both column-level and table-level constraints. If the constraint name is omitted, the DBMS assigns one. To drop a constraint (using the ALTER TABLE statement), specify the constraint name. It is advisable to specify a name when creating a constraint; otherwise system catalogs must be queried to determine the name assigned by the DBMS when the constraint was created.

Constraint Index Options

The primary key, unique, and referential constraint definitions can optionally include a WITH clause to describe the characteristics of the indexes that are created by Ingres to enforce the constraints.

The constraint_with_clause can be appended to both column- and table-level constraint definitions.

The column_constraint has the following syntax:

```
UNIQUE [WITH constraint_with_clause]
PRIMARY KEY [WITH constraint_with_clause]
REFERENCES [schema.]table_name[(column_name)][referential_actions][WITH
constraint_with_clause]
```

The table_constraint has the following syntax:

```
UNIQUE (column_name {,column_name}) [WITH constraint_with_clause]
PRIMARY KEY (column_name {,column_name}) [WITH constraint_with_clause]
     FOREIGN KEY (column_name {,column_name})
     REFERENCES [schema.]table_name[(column_name
{,column_name})][referential_actions][WITH constraint_with_clause]
```

constraint_with_clause

Describes the index characteristics as one or more of the following options.

If options are used in combination, they must be separated by commas and enclosed in parentheses.
For example: WITH (STRUCTURE = HASH, FILLFACTOR = 70).

- NO INDEX
- INDEX = BASE_TABLE_STRUCTURE
- INDEX = *index_name*
- STRUCTURE = HASH | BTREE | ISAM
- FILLFACTOR = *n*
- MINPAGES = *n*
- MAXPAGES = *n*
- LEAFFILL = *n*
- NONLEAFFILL = *n*
- ALLOCATION = *n*
- EXTEND = *n*
- LOCATION = (*location_name*{, *location_name*})

Note: The NO INDEX and INDEX = BASE_TABLE_STRUCTURE options cannot be used in combination with any other constraint WITH option.

No Index Option

The NO INDEX option indicates that no secondary index is created to support the constraint. This option is permissible for referential constraints only and results in no index being available to check the integrity of deletes and updates to the referenced table. The database procedures that perform the integrity checks still execute in the absence of these indexes. The query plan, however, may use some other user-defined index on the same columns; or it may resort to a full table scan of the referencing table, if there is no alternative.

To avoid poor performance, the NO INDEX option must be used only if:

- An alternate index on referencing columns is available
- There are very few rows in the referencing table (as in a prototype application)
- Deletes and updates are rarely (if ever) performed on the referenced table

Index = Base Table Structure Option

The INDEX = BASE TABLE STRUCTURE option indicates that the base table structure of the constrained table be used for integrity enforcement, rather than a newly created secondary index. The base table structure must not be heap and must match the columns in the constraint definition. Because only non-heap base table structures can be specified using the MODIFY statement (after the table has been created), WITH INDEX = BASE TABLE STRUCTURE can be used only for table constraints defined with the ALTER TABLE (rather than the CREATE TABLE) statement.

The ALTER TABLE statement, which adds the constraint, must be preceded by the WITH INDEX = BASE TABLE STRUCTURE statement.

For example:

```
modify [schema.]table_name to unique_scope = statement
```

which indicates that the uniqueness semantics enforced by the index are consistent with Ingres and ANSI rules.

Index = Index_Name Option

The INDEX = *index_name* option can be used for several purposes. If the named index already exists and is consistent with the columns constrained by the constraint definition, no new index is created. If the named index does not exist, the generated index created for constraint enforcement uses the name, *index_name.* Finally, if more than one constraint in the same table definition specifies INDEX = *index_name* with the same *index_name*, an index is generated with that name and is shared among the constraints.

In cases where an existing index is used for a constraint or a single index is shared among several constraints, the key columns of the index and the columns of the constraints must be compatible.

All other *constraint with options* perform the same function as the corresponding WITH options of the CREATE INDEX statement and the index related WITH options of the CREATE TABLE...AS SELECT statement. They are limited, however, to those options documented above. For example, the KEY and COMPRESSION options of CREATE INDEX and CREATE TABLE...AS SELECT are **not** supported for constraint definition.

Using Create Table...As Select

The CREATE TABLE...AS SELECT syntax creates a table from another table or tables. The new table is populated with the set of rows resulting from execution of the specified SELECT statement.

Note: The CREATE TABLE...AS SELECT syntax is an Ingres extension and not part of the ANSI/ISO Entry SQL-92 standard.

By default, the storage structure of the table is heap with compression. To override the default, issue the SET result_structure statement prior to issuing the CREATE TABLE...AS SELECT statement or specify the WITH STRUCTURE option.

By default, the columns of the new table have the same names as the corresponding columns of the base table from which you are selecting data. Different names can be specified for the new columns.

The data types of the new columns are the same as the data types of the source columns. The nullability of the new columns is determined as follows:

- If a source table column is nullable, the column in the new table is nullable.
- If a source table column is not nullable, the column in the new table is defined as not null.

If the source column has a default value defined, the column in the new table retains the default definition. However, if the default value in the source column is defined using an expression, the default value for the result column is unknown and its nullability depends on the source columns used in the expression. If all the source columns in the expression are not nullable, the result column is not nullable. If any of the source columns are nullable, the result column is nullable. Also see Default Type Conversion.

A SYSTEM_MAINTAINED logical key column cannot be created using the CREATE TABLE...AS SELECT syntax. When creating a table using CREATE TABLE...AS SELECT, any logical key columns in the source table that are reproduced in the new table are assigned the format of NOT SYSTEM_MAINTAINED.

Embedded Usage

In an embedded CREATE TABLE statement:

- Host language variables can be used to specify constant expressions in the *subselect* of a CREATE TABLE...AS SELECT statement.
- *Location_name* can be specified using a host language string variable.
- The preprocessor does not validate the syntax of the *with_clause*.

Permissions

This statement is available to all users.

Examples: Create Table (extended)

1. Create the employee table with columns eno, ename, age, job, salary, and dept.

```
create table employee
  (eno    smallint,
   ename  varchar(20) not null,
   age    smallint,
   job    smallint,
   salary float4,
   dept   smallint);
```

2. Create a table listing employee numbers for employees who make more than the average salary.

```
create table highincome as
  select eno
  from employee
  where salary >
     (select avg (salary)
     from employee);
```

3. Create a table specifying defaults.

```
create table dept
( dname         char(10)
  location      char(10) default 'LA'
  creation_date date default date('1/1/93'));
```

4. Create a table specifying referential constraints. When a department number is assigned to an employee, it will be checked against the entries in the dept table.

```
create table emps (
  empno   char(5),
  deptno  char(5) references dept),
  ...);
```

5. Create a table specifying check constraints. In this example, department budgets default to $100,000, expenses to $0. The check constraint insures that expenses do not exceed the budget.

```
create table dept (
  dname    char(10),
  budget   decimal default 100000,
  expenses decimal default 0,
  check    (budget >= expenses));
```

6. Create a table specifying unique constraints and keys.

```
create table dept (
  deptno    char(5) primary key,
  dname     char(10) not null,
  dlocation char(10) not null,
unique (dname, dlocation));
```

7. Create a table specifying null constraints.

```
create table emp (
  salary    decimal not default with null ,
  hiredate  date not default with null,
  sickdays  float default 5.0 with null );
```

8. Unique constraint uses base table structure, not a generated index:

```
alter table department add primary key (dept_id)
    with index = base table structure;
```

9. Unique constraint generates index in non-default location. First referential constraint generates no index at all:

```
create table employee (empl_no int not null
    unique with location = (ixloc1),
    emp_name char(20) not null,
    dept_id char(6) references department    (dept_id) with no index,
    mgrno int references employee (empl_no));
```

10. Referential and primary key constraints share the same named index:

```
create table assignment (empl_no int not null
    references employee (empl_no) with (index =      assnpkix,
    location = (ixloc2)),
    proj_id int not null references project  (proj_id),
    task char(20),
    primary key (empl_no, proj_id) with index =
    assnpkix);
```

11. Referential action:

```
create table employee (empl_no int not null
    unique with location = (ixloc1),
    emp_name char(20) not null,
    dept_id char(6) references department (dept_id)
        on delete cascade on update cascade with no index,
    mgrno int references employee (empl_no) on update cascade
        on delete set null);
```

Grant

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The GRANT statement grants privileges on the database as a whole or on individual tables, views, or procedures.

Syntax

The GRANT (privilege) statement has the following format:

```
[EXEC SQL] GRANT ALL [PRIVILEGES] | privilege {, privilege}
              [ON [TABLE] [schema.]table_name
                           TO PUBLIC | auth_id {, auth_id}
              [WITH GRANT OPTION];
```

privilege

Specifies the type of privilege.

table_name

Specifies the name of the table for which the privilege is being defined.

PUBLIC

Grants the privilege to all users.

auth_id

Specifies the name of the users to which you are granting privileges.

Description

The grant statement enables a DBA or user to control access to tables. To remove privileges, use the REVOKE statement. The following table describes the GRANT statement parameters.

By default, neither the public nor any user has any table privileges. Table privileges must be granted explicitly. Valid table privileges are:

- Select
- Insert
- Update

For update, a list of columns can optionally be specified; if the column list is omitted, update privilege is granted to all updatable columns of the table or view.

- Delete
- References-The references privilege enables specified users to create referential constraints that reference the specified tables and columns. For details about referential constraints, see Create Table (extended) (see page 328).

  A list of columns can optionally be specified. If the column list is omitted, references privilege is granted to all updatable columns of the table. You cannot grant the references privilege on a view.
- All [privileges]-All grants select, insert, update, delete, and references on the specified objects to the specified users.

The Grant All Privileges Option

To grant a privilege on an object you do not own, you must have been granted the privilege WITH GRANT OPTION--only the privileges for which you have grant option are granted.

The results of granting all privileges on a view you do not own are determined as follows:

- **Select** - Granted if you can grant select privilege on all tables and views in the view definition.
- **Update** - Granted for all columns for which you can grant update privilege; if you were granted update...with grant option on a subset of the columns of a table, update is granted only for those columns.
- **Insert** - Granted if you can grant insert privilege on all tables and views in the view definition.
- **Delete** - Granted if you can grant delete privilege on all tables and views in the view definition.
- **References** - The references privilege is not valid for views.

The following example illustrates the results of the GRANT ALL PRIVILEGES option. The accounting_mgr user creates the following employee table:

```
create table employee (name character(25),
   department character(5), salary decimal)...
```

Using the following GRANT statement, grants the accounting_supervisor user the ability to select all columns but only allows accounting_supervisor to update the department column (to prevent unauthorized changes of the salary column):

```
grant select, update (department) on table employees
   to accounting_supervisor with grant option;
```

If the accounting_supervisor user issues the following GRANT statement:

```
grant all privileges on table employees to
accounting_clerk;
```

the accounting_clerk user receives select and update(department) privileges.

The Grant Option

To enable a user to grant a privilege to another user, specify the WITH GRANT OPTION clause.

For example, if user tony, creates a table called mytable, and issues the following statement:

```
grant select on tony.mytable to laura
  with grant option;
```

user laura can select data from tony.mytable and can authorize user evan to select data from tony.mytable by issuing the following statement:

```
grant select on tony.mytable to evan;
```

Because laura did not specify the WITH GRANT OPTION clause, evan cannot authorize another user to select data from tony.mytable.

The owner of an object can grant any privilege to any user (or to public). The user to whom the privilege is granted with grant option can grant only the specified privilege. In the preceding example, laura can grant select privilege but cannot grant, for example, insert privilege.

In the previous example, the second grant (to evan) depends on the first grant (to laura). If tony revokes select permission from laura (using the REVOKE statement), tony must specify how OpenSQL should handle dependent grants that laura has issued. The choices are:

- **Revoke with cascade**-Revokes all dependent grants; in the preceding example, select permission will be revoked from user, evan.
- **Revoke with restrict**-Do not revoke specified grant if there are dependent grants. In the preceding example, select permission will not be revoked from laura because her grant to evan depends on the grant she received from tony.

For more details, see Revoke (see page 352) in this chapter, and in the *Database Administrator Guide.*

Embedded Usage

Specify the with clause using a host string variable (with :*hostvar*).

Permissions

To grant privileges on an object, you must own the object or have the grant option for the privilege you are granting.

Examples: Grant

1. Grant update privileges on the columns, empname and empaddress in the employee table to the users, joank and gerryr.

```
grant update(empname, empaddress)
  on table employee
to joank, gerryr;
```

2. Enable any user to select data from the employee roster.

```
grant select on emp_roster to public;
```

3. Enable the accounting manager, rickr, complete access to salary information and to grant permissions to other users.

```
grant all on employee to rickr with grant option;
```

4. Enable any user to create a table constraint that references the employee roster.

```
grant references on emp_roster to public;
```

Revoke

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REVOKE statement removes database privileges granted to the specified users or public. You cannot revoke privileges granted by other users.

For more information about privileges, see Grant (see page 347).

Syntax

The REVOKE statement has the following format:

```
[EXEC SQL] REVOKE [GRANT OPTION FOR]
              ALL [PRIVILEGES] | privilege {, privilege}
              [ON [TABLE] [schema.]table_name
              FROM PUBLIC | auth_id {, auth_id}
              [CASCASE | RESTRICT];
```

privilege

Specifies the privilege to revoke. To revoke all privileges, use ALL. The privileges must be one of the following:

- Select
- Update
- Insert
- Delete
- References

table_name

Specifies the name of the table on which the privileges were granted.

auth_id

Specifies the authorization identifier from which privileges are being revoked.

Revoking the Grant Option

The GRANT statement GRANT OPTION enables users other than the owner of an object to grant privileges on that object. For example:

```
grant select on employee_roster to mike with grant
  option;
```

enables mike to grant the select privilege (with or without GRANT OPTION) to other users.

The GRANT option can be revoked without revoking the privilege with which it was granted. For example:

```
revoke grant option for select on employees to
   mike...
```

means that mike can still select data from the employees table, but cannot grant the select privilege to other users.

Restrict versus Cascade

The RESTRICT and CASCADE options specify how OpenSQL handles dependent privileges. The CASCADE option (default) directs OpenSQL to revoke the specified privileges plus all privileges that depend on the privileges being revoked. The RESTRICT option directs OpenSQL not to revoke the specified privilege if there are any dependent privileges.

The owner of an object can grant privileges on that object to any user. Privileges granted by users who do not own the object are dependent on the privileges granted WITH GRANT OPTION by the owner.

For example, if user jerry owns the employees table, he can grant tom the ability to select data from the table and to enable other users to select data from the table:

```
grant select on employees to tom with grant option;
```

User tom can now enable another user to select data from the employees table:

```
grant select on employees to sylvester with grant option;
```

The grant tom conferred on sylvester is dependent on the grant the table's owner jerry conferred on tom. In addition, sylvester can enable other users to select data from the employees table.

- To remove his grant to tom and all grants tom may have issued, jerry must specify REVOKE...CASCADE:

  ```
  revoke select on employees from tom cascade;
  ```

  As a result of this statement, the select privilege granted by tom to sylvester is revoked, as are any select grants issued by sylvester to other users conferring select privilege for the employees table.

- To prevent dependent privileges from being revoked, jerry must specify revoke... restrict:

  ```
  revoke select on employees from tom restrict;
  ```

  Because there are dependent privileges (tom has granted select privilege on the employees table to sylvester), this revoke statement will fail, and no privileges will be revoked.

The RESTRICT and CASCADE parameters have the same effect whether you are revoking a specific privilege or the grant option for a specific privilege. In either case, RESTRICT prevents the operation from occurring if there are dependent privileges, and CASCADE causes dependent privileges to be deleted. When you revoke a grant option with CASCADE, all dependent privileges are revoked, not just the grant option portion of the dependent privileges.

Embedded Usage

You cannot use host language variables in an embedded REVOKE statement.

Permissions

This statement is available to all users.

Example: Revoke

Prevent any user from granting any form of access to the payroll table. Delete all dependent grants.

```
revoke grant option for all on payroll
  from public cascade;
```

Select

Valid in: SQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The SELECT statement retrieves values from tables or views.

Select Syntax

The SELECT statement has the following format:

```
SELECT [ALL | DISTINCT]  * | result_expression {,result_expression}
               [FROM from_source {, from_source}
               [WHERE search_condition]
               [GROUP BY column{, column}]
               [HAVING search_condition]
               {UNION [ALL]
               (select)
               [ORDER BY result_column [ASC | DESC]
                            {, result_column [ASC | DESC]}];
```

where *result_expression* is one of the following:

- [*schema*.]*table_name*.*****

  Selects all columns

- [[*schema*.]*table_name*.]*column_name* [AS *result_column*]

  Selects one column

- *expression* [AS] *result_column*

For SQL-92 compliant installations, the AS keyword in the result expression is optional. All other select syntax and semantics are described in the chapter "OpenSQL Statements."

Chapter 10: OpenSQL Limits

This section contains the following topics:

OpenSQL Limits

To maintain application portability, your OpenSQL application should observe the limits listed in the following table. Individual host DBMSes may permit values that exceed these.

Item	Min/Max	Limit
Char length	Max	240 characters
Columns in index	Max	16 columns
Columns in ORDER BY clause	Max	16 columns
Columns in table	Max	127 columns
Columns in view	Max	127 columns
Columns: total length in GROUP BY clause	Max	2000 bytes
Columns: total length in ORDER BY clause	Max	2000 bytes
Elements in SELECT list	Max	127 elements
Negative float value	Min	Processor-dependent
Negative float value	Max	Processor-dependent
Positive float value	Min	Processor-dependent
Positive float value	Max	Processor-dependent
Host variables in OpenSQL statement	Max	256 variables
Integer value	Min	-2,147,483,648
Integer value	Max	+2,147,483,647
Predicates in HAVING clause	Max	50 predicates
Predicates in WHERE clause	Max	50 predicates
Row length (including overhead)	Max	2000 bytes
Scalar functions in select list	Max	127 functions

Item	Min/Max	Limit
Smallint value	Min	-32,768
Smallint value	Max	32,767
SQL identifier	Max	18 characters
Tables in SQL statement	Max	15 tables
User ID	Max	18 characters
Varchar length	Max	2000 characters

Chapter 11: OpenSQL Standard Catalogs

This section contains the following topics:

This chapter describes the Standard Catalog Interface catalogs.

Standard Catalog Level

The Standard Catalog Interface described here corresponds to the formats you will find when the iidbcapabilities catalog contains the values in the following table.

CAP_CAPABILITY	CAP_VALUE
STANDARD_CATALOG_LEVEL	00602

System Catalog Characteristics

Unless otherwise noted, values in system catalogs are left justified, and columns are non-nullable.

The length of char fields, as listed in the Data Type column, is a maximum length. The actual length of the field is installation-dependent. When developing applications that access these catalogs, allocate storage based on the length as shown in the Data Type column.

All dates stored in system catalogs have the following format (underscores and colons are required):

yyyy_mm_dd hh:mm:ss GMT (Greenwich Mean Time)

Note: In this chapter, "default" means the assumed value if no other value is present.

Standard Catalog Interface

The Standard Catalog Interface catalogs are read-only views built on system catalogs of the underlying DBMS. Users who need to query the system catalogs must use the Standard Catalog Interface. The Standard Catalog Interface provides a portable representation for information about OpenSQL.

The iidbcapabilities Catalog

The iidbcapabilities catalog contains information about the capabilities provided by the Enterprise Access product or Ingres DBMS. The following table describes the columns in the iidbcapabilities catalog:

Column Name	Data Type	Description
cap_capability	char(32)	Contains one of the values listed in the capability column of the following table.
cap_value	char(32)	The contents of this field depend on the capability. See the Values column in the following table.

The cap_capability Column

The cap_capability column in the iidbcapabilities catalog contains one or more of the following values:

Capability	Value
CAP_SLAVE2PC	Indicates if the DBMS supports Ingres two-phase commit slave protocol: Version 6.3 and above: Y Ingres Star: Y Enterprise Access product: usually N
DBEVENTS	Y if the DBMS supports database events, N if not.
DBEVENT_GRANT	Y if DBEvent related grant statements are accepted, N if rejected.
DB_NAME_CASE	Case sensitivity of the database with respect to database object names: LOWER, UPPER, MIXED. Defaults to

Capability	Value
	LOWER. If the value is MIXED, case must be carefully preserved when specifying database objects. This field applies to names of database objects (tables, views, columns, and owners.) Names of user interface objects (such as forms or reports) are always lower case.
DB_DELIMITED_CASE	Case conversion performed by the DBMS for object names specified using delimited identifiers (that is, in double quotes). LOWER if delimited identifiers are translated to lower case, UPPER if delimited identifiers are translated to upper case, or MIXED if no case translation is performed.
DBMS_TYPE	The type of DBMS the application is communicating with. Valid values are the same as those accepted by the WITH DBMS clause used in queries. Examples: INGRES, STAR, RMS. The default value is INGRES.
DISTRIBUTED	Y if the DBMS is distributed, N if not.
ESCAPE	Contains Y if DBMS supports the ESCAPE clause of the LIKE predicate in the WHERE clause of search statements. Contains N if ESCAPE is not supported.
ESCAPE_CHAR	Character used to escape pattern characters in LIKE predicate.
IDENT_CHAR	Characters permitted in non-delimited identifiers beyond alphanumeric and '_'.
INGRES	Set to Y if the DBMS supports all versions of Ingres Release 6 and Ingres Release 1; otherwise N. Default is Y.
INGRES/SQL_LEVEL	Version of Ingres SQL supported by the DBMS. Examples: 00600 = 6.0 00601 = 6.1

Capability	Value
	00602 = 6.2 00603 = 6.3 00604 = 6.4 00605 = OpenIngres 1.x 00800 = OpenIngres 2.0 and Ingres II 2.0 00850 = Ingres II 2.5 00860 = Ingres 2.6 00902 = Ingres r3 00904 = Ingres 2006 00910 = Ingres 2006 Release 2 00920 = Ingres 9.2 00000 = DBMS does not support Ingres SQL Default is 00000.
INGRES/QUEL_LEVEL	Version of Ingres QUEL supported by the DBMS. Examples: 00600 = 6.0 00601 = 6.1 00602 = 6.2 00603 = 6.3 00604 = 6.4 00605 = OpenIngres 1.x 00800 = OpenIngres 2.0 and Ingres II 2.0 00850 = Ingres II 2.5 00860 = Ingres 2.6 00902 = Ingres r3 00904 = Ingres 2006 00910 = Ingres 2006 Release 2 00920 = Ingres 9.2 00000 = Does not support QUEL Default is 00000.

Capability	Value
INGRES_AUTH_GROUP	Y if the DBMS supports group identifiers.
INGRES_AUTH_ROLE	Y if the DBMS supports role identifiers.
INGRES_LOGICAL_KEY	Y if the DBMS supports Ingres logical keys.
INGRES_RULES	Y if the DBMS supports Ingres rules; N if it does not.
INGRES_UDT	Y if the DBMS supports Ingres user-defined data types, N if the DBMS does not support user-defined data types.
MAX_COLUMNS	Maximum number of columns allowed in a table. Default is 127.
NATIONAL_CHARACTER_SET	Y if the DBMS supports Unicode, N if it does not.
NULL_SORTING	How NULL values are sorted relative to other values: HIGH NULLS are considered the highest possible value. LOW NULLS are considered the lowest possible value. FIRST NULLs appear at start regardless of ascending/descending. LAST NULLS appear at end regardless of ascending/descending.
OPEN_SQL_DATES	Contains LEVEL 1 if the Enterprise Access product supports the OpenSQL date data type.
OPEN/SQL_LEVEL	Version of OpenSQL supported by the DBMS. Examples: 00600 = 6.0 00601 = 6.1 00602 = 6.2 00603 = 6.3 00604 = 6.4 00605 = OpenIngres 1.x

Capability	Value
	00800 = OpenIngres 2.0 and Ingres II 2.0 00850 = Ingres II 2.5 00860 = Ingres 2.6 00902 = Ingres r3 00904 = Ingres 2006 Default is 00602.
OPENSQL_SCALARS	Can be one of three values: 'NATIVE', 'FULL' or LEVEL 1. The default value is 'NATIVE'. 'NATIVE' indicates only native DBMS scalar functions are supported. 'FULL' indicates full Ingres scalar function support is provided. LEVEL 1 indicates *some* mapping of Ingres scalar functions. When OPENSQL_SCALARS is set to LEVEL 1, an additional table, iigwscalars, is provided which shows support details for individual scalar functions. See The iigwscalars Catalog (see page 367).
OUTER_JOIN	Whether outer joins are supported: N for no, Y for yes.
OWNER_NAME	Contains N if *owner.table* table name format not supported. Contains Y if *owner.table* format supported. Contains QUOTED if *owner.table* supported with optional quotes (“owner”.table).
PHYSICAL_SOURCE	T indicates that both iitables and iiphysical_tables contain physical table information. P indicates that only iiphysical_tables contains the physical table information.
SAVEPOINTS	Y if savepoints behave exactly as in Ingres, else N. Default is Y.
STANDARD_CATALOG_LEVEL	Version of the standard catalog interface supported by this

Capability	Value
	database. Valid values: 00602 (default) 00604 00605 00800 00850 00860 00902 00904 For catalog formats, see the appendix "System Catalogs" in the *Database Administrator Guide*.
SQL92_COMPLIANCE	The level of SQL-92 supported: NONE — SQL-92 Entry level not supported ENTRY — SQL-92 Entry level FIPS-IEF — SQL-92 Entry level plus FIPS Integrity Enhancements features INTERMEDIATE — SQL-92 Intermediate level FULL — SQL-92 Full level
SQL_MAX_CHAR_COLUMN_LEN	Maximum length of a CHAR column. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_VCHR_COLUMN_LEN	Maximum length of a VARCHAR column. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_NCHAR_COLUMN_LEN	Maximum number of characters for an NCHAR column.
SQL_MAX_NVCHR_COLUMN_LEN	Maximum number of characters for an NVARCHAR column.
SQL_MAX_BYTE_COLUMN_LEN	Maximum length of a BYTE column. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_VBYT_COLUMN_LEN	Maximum length of a VARBYTE

Capability	Value
	column. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_CHAR_LITERAL_LEN	Maximum length of a string literal. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_BYTE_LITERAL_LEN	Maximum length of a hex literal. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_USER_NAME_LEN	Maximum length of a user name. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_ROW_LEN	Maximum length of a row. Permits 0 for unsupported and -1 for unknown or unlimited.
SQL_MAX_STATEMENTS	Maximum number of active (prepared) statements. Permits 0 for unsupported and -1 for unknown or unlimited.
UNION	Whether UNION selects are supported: N No Y Yes ALL Yes and UNION ALL.
UNIQUE_KEY_REQ	Set to Y if the database service requires that some or all tables have a unique key. Set to N or not present if the database service allows tables without unique keys.

The iidbconstants Catalog

The iidbconstants catalog contains values required by the Ingres tools. The following table describes the columns in the iidbconstants catalog:

Column Name	Data Type	Description
user_name	char(32)	Name of the current user
dba_name	char(32)	Name of the owner of the database
system_owner	char(32)	The name of the system catalog

Column Name	Data Type	Description
		owner (for example, $ingres)

The iievents Catalog

The iievents catalog contains an entry for each database event that has been created. This catalog is present only if the DBEVENTS entry in the iidbcapabilities catalog section has a value of Y. For complete information about database events, see Database Events (see page 191).

The information is stored in the following format:

Column Name	Data Type	Description
event_name	char(32)	Name of the event. This name is unique among all events owned by user.
event_owner	char(32)	Owner of the event. This name can be referenced in the different event statements to qualify the event.
text_sequence	integer	Text sequence of create dbevent text.
text_segment	varchar (240)	Text segment of create dbevent text.

The iigwscalars Catalog

The iigwscalars catalog contains an entry for each function that an Enterprise Access Product supports. This catalog is present only if the OPENSQL_SCALARS entry in the iidbcapabilities catalog section has a value of LEVEL 1. If this catalog is present in your database, it details the level of support provided for functions.

The information is stored in the following format:

Column Name	Data Type	Description
function_name	char(32)	Name identifying the function.
support	char(10)	This column has one of four values: RESTRICT indicates this function is supported but with restrictions. A restriction may be as simple as requiring a literal value for a parameter, or may indicate a slight variation from the

Column Name	Data Type	Description
		standard Ingres behavior for this function.
		COMPAT indicates a native DMBS function exactly matches the Ingres function. No translation is performed. TRANS indicates that the function is translated to the target DMBS SQL without restrictions. NO indicates that a function is not supported.
parm1	char(10)	A value of LITERAL indicates that the first parameter to this function must be a literal value. A value of EXPR indicates that both literal values and expressions are allowed for this parameter.
parm2	char(10)	LITERAL or EXPR.
parm3	char(10)	LITERAL or EXPR.
mapping	varchar(253)	This column documents the target DBMS SQL that is generated when the Enterprise Access product does translations.
comments	varchar(400)	Comments.

The iitables Catalog

The iitables catalog contains an entry for each queryable object in the database (table, view, or index). To find out what tables, views, and indexes are owned by you or the DBA, you can query this catalog. For example:

```
select * from iitables where (table_owner = user or
table_owner = (select dba_name from iidbconstants))
```

Column Name	Data Type	Description
table_name	char(32)	The object's name. Must be a valid object name.
table_owner	char(32)	The owner's user name. Generally, the creator of the object is the owner.
create_date	char(25)	The object's creation date. Blank if

Column Name	Data Type	Description
		unknown.
alter_date	char(25)	The last time this table was altered. This date is updated whenever the logical structure of the table changes, either through changes to the columns in the table or changes to the primary key. Physical changes to the table, such as changes to data, secondary indexes, or physical keys, do not change this date. Blank if unknown.
table_type	char(8)	Type of query object: T = Table V = View I = Index Further information about tables can be found in iiphysical_tables. Further information about views can be found in iiviews.
table_subtype	char(8)	Specifies the type of table or view. Possible values are: N (native) - For standard Ingres databases. L (links) - For Ingres Star. I (imported tables) - For Enterprise Access products. (blank) - If unknown
table_version	char(8)	Version of the object. Enables the user interfaces to determine where additional information about this particular object is stored. This reflects the database type, as well as the version of an object within a given database. For Ingres tables, the value for this field is II2.5.
system_use	char(8)	Contains S if the object is a system object, U if user object, or blank if unknown. Used by utilities to determine which tables need reloading. If the value is unknown, the utilities use the naming convention of "ii" for tables to distinguish between system and user catalogs. In addition, any table beginning with ii_ is assumed to be a user interface object, rather than a DBMS system

Column Name	Data Type	Description
		object. The standard system catalogs themselves must be included in the iitables catalog and are considered system tables.
table_size	integer	Stores the page size of a table.

The following columns in iitables have values only if table_type is T (table) or I (index).

Enterprise Access products that do not supply this information set these columns to -1 for numeric data types, blank for character data types.

Column Name	Data Type	Description
table_stats	char(8)	Contains Y if this object has entries in the iistats table, N if this object does not have entries. If this field is blank, then query iistats to determine if statistics exist. This column is used for optimization of Ingres databases.
table_indexes	char(8)	Contains Y if this object has entries in the iiindexes table that refer to this as a base table, or N if this object does not have entries. If the field is blank, then query iiindexes on the base_table column. This field is used for optimization of Ingres databases.
is_readonly	char(8)	Contains one of these values: N - If updates are physically allowed Y - If no updates are allowed Blank - If unknown Used for tables that are defined to the Enterprise Access product only for retrieval, such as tables in hierarchical database systems. If this field is set to Y, then no updates will work, independent of what permissions might be set. If it is set to N, updates may be allowed, depending

Column Name	Data Type	Description
		on whether the permissions allow it.
concurrent_access	char(1)	Y if concurrent access is allowed.
num_rows	integer	The estimated number of rows in the table. Set to -1 if unknown.
storage_structure	char(16)	The storage structure for the table: heap, hash, btree, or isam. Blank if unknown.
is_compressed	char(8)	Contains Y if the table is stored in compressed format, N if the table is uncompressed, blank if unknown.
key_is_compressed	char(8)	Contains Y if the table uses key compression, N if no key compression, or blank if unknown.
duplicate_rows	char(8)	D - If the table allows duplicate rows. U - If the table does not allow duplicate rows. Blank - If unknown. The table storage structure (unique or non-unique keys) can override this setting.
unique_rule	char(8)	D - Indicates that duplicate physical storage structure keys are allowed. (A unique alternate key may exist in iialt_columns and any storage structure keys may be listed in iicolumns.) U - If the object is an Ingres object, indicates that the object has unique storage structure key. If the object is not an Ingres object, then it indicates that the object has a unique key, described in either iicolumns or iialt_columns. Blank - If uniqueness is unknown or does not apply.
number_pages	integer	The estimated number of physical pages in the table. Set to -1 if unknown.
overflow_pages	integer	The estimated number of overflow pages in the table. Set to -1 if unknown.

Column Name	Data Type	Description
row_width	integer	The size, in bytes, of the uncompressed binary value for a row of this query object.
unique_scope	char(8)	R if this object is row-level, S if statement-level, blank if not applicable.
allocation_size	integer	The allocation size, in pages. Set to -1 if unknown.
extend_size	integer	The extend size, in pages. Set to -1 if unknown.
allocated_pages	integer	Total number of pages allocated to the table.

The following columns are used by the Ingres DBMS Server. If an Enterprise Access does not supply this information, it will set these columns to the default values: -1 for numeric columns and a blank for character columns. The information in the following section is not duplicated in iiphysical_tables.

Column Name	Data Type	Description
expire_date	integer	Expiration date of table. This is an Ingres _bintim date.
modify_date	char(25)	The date on which the last physical modification to the storage structure of the table occurred. Blank if unknown or inapplicable.
location_name	char(24)	The first location of the table. If there are additional locations for a table, they are shown in the iimulti_locations table and multi_locations is set to Y.
table_integrities	char(8)	Contains Y if this object has Ingres style integrities. If the value is blank, query the iiintegrities table to determine if integrities exist.
table_permits	char(8)	Contains Y if this object has Ingres style permissions.
all_to_all	char(8)	Contains Y if this object has Ingres permit all to all, N if not.
ret_to_all	char(8)	Contains Y if this object has Ingres permit retrieve to all, N if not.

Column Name	Data Type	Description
is_journalled	char(8)	Contains Y if Ingres journaling is enabled on this object, N if not.
view_base	char(8)	Contains Y if object is a base for a view definition, N if not, or blank if unknown.
multi_locations	char(8)	Contains Y if the table is in multiple locations, N if not.
table_ifillpct	smallint	Fill factor for the index pages used on the last modify command in the nonleaffill clause, expressed as a percentage (0 to 100). Used for Ingres btree structures to rerun the last modify command.
table_dfillpct	smallint	Fill factor for the data pages used on the last modify command in the fillfactor clause, expressed as a percentage (0 to 100). Used for Ingres btree, hash, and isam structures to rerun the last modify command.
table_lfillpct	smallint	Fill factor for the leaf pages used on the last modify command in the leaffill clause, expressed as a percentage (0 to 100). Used for Ingres btree structures to rerun the last modify command.
table_minpages	integer	Minpages parameter from the last execution of the modify command. Used for Ingres hash structures only.
table_maxpages	integer	Maxpages parameter from the last execution of the modify command. Used for Ingres hash structures only.
table_relstamp1	integer	High part of last create or modify timestamp for the table.
table_relstamp2	integer	Low part of last create or modify timestamp for the table.
table_reltid	integer	The first part of the internal relation ID.
table_reltidx	integer	The second part of the internal relation ID.
table_relversion	integer	Stores the version of table.

Column Name	Data Type	Description
table_reltotwidth	integer	This width includes all deleted columns.
table_reltcpri	integer	Indicates a table's priority in the buffer cache. Values can be between 0 and 8. Zero is the default, and 1-8 can be specified using the priority clause in create table or modify table.

The iicolumns Catalog

For each queryable object in the iitables catalog, there are one or more entries in the iicolumns catalog. Each row in iicolumns contains the logical information on a column of the object. User interfaces and user programs use the iicolumns catalog to perform dictionary operations and dynamic queries.

Column Name	Data Type	Description
table_name	char(32)	The name of the table.
table_owner	char(32)	The owner of the table.
column_name	char(32)	The column name.
column_datatype	char(32)	The column data type name returned to users and applications: ▪ Decimal ▪ Integer ▪ Int ▪ Float ▪ Real ▪ Double precision ▪ Char ▪ Character ▪ Varchar ▪ Date
column_length	integer	The length of the column returned to users and applications. If a data type contains two length specifiers, this column uses the first length. Set to zero for the data types that are

Column Name	Data Type	Description
		specified without length (date). This length is not the actual length of the column's internal storage. For decimal columns, contains the precision.
column_scale	integer	The second number in a two-part user length specification. For typename (len1, len2) it will be len2.
column_nulls	char(8)	Contains Y if the column can contain null values, N if the column cannot contain null values.
column_defaults	char(8)	Contains Y if the column is given a default value when a row is inserted, or N if the column is not given a default value.
column_sequence	integer	The number of this column in the corresponding table's create statement, numbered from 1.
key_sequence	integer	The order of this column in the primary key, numbered from 1. For an Ingres table, this indicates the column's order in the primary storage structure key. If 0, then this column is not part of the primary key.
sort_direction	char(8)	Defaults to A (for ascending) when key_sequence is greater than 0; otherwise, this value is a blank.
column_ingdatatype	smallint	Contains the numeric Ingres representation of the column's external data type (the data type returned to users and applications). If the installation has user-defined data types (UDTs), this column contains the data type that the UDT is converted to when returned to an Ingres user interface product. If the value is positive then the column is not nullable. If the value is negative, then the column is nullable. The data types and their corresponding values are:

Column Name	Data Type	Description
		decimal - 10/10 integer - 30/30 float - 31/31 date* - 3/3 char - 20/20 varchar - 21/21 * Returned as a string
column_internal_ datatype	char(32)	The internal data type of the column: char, c, varchar, text, integer, float, date, money, table_key, object_key. If the installation has user-defined data types, this column contains the user-specified name.
column_internal_ length	smallint	The internal length of the column. For example, for data type smallint, this column contains 2. Contains 0 if the data type is fixed-length. The length does not include the null indicator byte for nullable columns, or the length specifier byte for varchar and text columns.
column_internal_ ingtype	smallint	Contains the numeric representation of the internal datatype. For a list of valid values, see column_ingdatatype. If the installation has user-defined data types, this column contains the user-specified data type number.
column_system_ maintained	char(8)	Contains Y if the column is system-maintained, or N if not system-maintained.
column_updatable	char(8)	Contains Y if the column can be updated, N if not, or blank if unknown.
column_has_default	char(8)	Contains Y if the column has a default value, N if not, or blank if unknown.
column_default_ value	varchar(150 1)	The default value defined for the column.

The iiphysical_tables Catalog

The information in the iiphysical_tables catalog overlaps with some of the information in iitables. This information is provided as a separate catalog primarily for use by Enterprise Access products. Query the physical_source column, in iidbcapabilities, to determine whether you must query iiphysical_tables. If you do not want to query iidbcapabilities, then you must always query iiphysical_tables to be sure of getting the correct information.

If a queryable object is type T or I (for index, in an Ingres installation only), then it is a physical table and may have an entry in iiphysical_tables as well as iitables.

In most Enterprise Access products, this table is keyed on table_name plus table_owner:

Column Name	Data Type	Description
table_name	char(32)	The table name. This is an object name.
table_owner	char(32)	The table owner's user name.
table_stats	char(8)	Y if this object has entries in the iistats table.
table_indexes	char(8)	Y if this object has entries in the iiindexes table that refer to this as a base table.
is_readonly	char(8)	Y if updates are physically allowed on this object.
concurrent_access	char(8)	Y if concurrent access is allowed.
num_rows	integer	The estimated number of rows in the table. Set to -1 if unknown.
storage_structure	char(16)	The storage structure of the table. Possible values are: heap, btree, isam, or hash.
is_compressed	char(8)	Indicates if the table is stored in compressed format. Y if it is compressed, N if not compressed, or blank if unknown.
key_is_compressed	char(8)	Indicates if the table uses compression. Y if the table uses compression, N if no compression, or blank if unknown.
duplicate_rows	char(8)	Contains U if rows must be unique, D if duplicates are allowed, or blank if

Column Name	Data Type	Description
		unknown.
unique_rule	char(8)	Contains U if the storage structure is unique, D if duplicates are allowed, or blank if unknown or inapplicable.
number_pages	integer	The estimated number of physical pages in the table. Set to -1 if unknown.
overflow_pages	integer	The estimated number of overflow pages in the table. Set to -1 if unknown.
row_width	integer	The size (in bytes) of the uncompressed binary value for a row in the object for Ingres. Set to -1 if this is unknown.
allocation_size	integer	The table allocation size, in pages. Set to -1 if unknown.
extend_size	integer	The extend size , in pages. Set to -1 if unknown.
allocated_pages	integer	The total number of pages allocated to the table.
table_pagesize	integer	Stores the pages of a table

The iiviews Catalog

The iiviews catalog contains one or more entries for each view in the database (views are indicated in iitables by table type = "V"). Because the text_segment column is limited to 240 characters per row, a single view can require more than one row to contain all its text. In this case, the text will be broken in mid-word across the sequenced rows. The text column is pure text, and can contain newline characters.

Column Name	Data Type	Description
table_name	char(32)	The view name. Must be a valid object name.
table_owner	char(32)	The view owner's user name.
view_dml	char(8)	The language in which the view was created: S (for SQL) or Q (for QUEL).
check_option	char(8)	Contains Y if the check option was specified in the create view statement, N if

Column Name	Data Type	Description
		not, or blank if unknown.
text_sequence	integer	The sequence number for the text field, starting with 1.
text_segment	varchar(256)	The text of the view definition.

The iiindexes Catalog

Each table with a table_type of I (index) in the iitables table has an entry in iiindexes. In Ingres, all indexes also have an entry in iiphysical_tables.

Column Name	Data Type	Description
index_name	char(32)	The index name. Must be a valid object name.
index_owner	char(32)	The index owner's user name.
create_date	char(25)	Creation date of index.
base_name	char(32)	The base table name. Must be a valid object name.
base_owner	char(32)	The base table owner. Must be a valid user name.
storage_structure	char(16)	The storage structure for the index: heap, hash, isam, or btree.
is_compressed	char(8)	Contains Y if the table is stored in compressed format, N if the table is uncompressed, or blank if unknown.
unique_rule	char(8)	Contains U if the index is unique, D if duplicate key values are allowed, or blank if unknown.
unique_scope	char(8)	Contains S if uniqueness is checked after completion of queries or R if uniqueness is checked after each row is modified or inserted.
system_use	char(8)	Contains S if the index was created by the DBMS, U if created by a user, or blank if unknown. (The Ingres DBMS creates unique indexes to enforce unique constraints on tables.)

Column Name	Data Type	Description
persistent	char(8)	Contains Y if the index is retained when its base table is modified (using the Ingres modify statement), or N if the index is dropped when the table is modified.

The iiindex_columns Catalog

For indexes, any Ingres columns that are defined as part of the primary index key will have an entry in iiindex_columns. For a full list of all columns in the index, use the iicolumns catalog.

Column Name	Data Type	Description
index_name	char(32)	The index containing *column_name*. This is an object name.
index_owner	char(32)	The index owner. Must be a valid user name.
column_name	char(32)	The name of the column. Must be a valid object name.
key_sequence	integer	Sequence of column within the key, numbered from 1.
sort_direction	char(8)	Defaults to A (ascending).

The iialt_columns Catalog

All columns defined as part of an alternate key have an entry in iialt_columns.

Column Name	Data Type	Description
table_name	char(32)	The table to which column_name belongs.
table_owner	char(32)	The table owner.
key_id	integer	The number of the alternate key for this table.
column_name	char(32)	The name of the column.

Column Name	Data Type	Description
key_sequence	smallint	Sequence of column within the key, numbered from 1.

The iistats Catalog

This catalog contains entries for columns that have statistics.

Column Name	Data Type	Description
table_name	char(32)	The name of the table.
table_owner	char(32)	The table owner's user name.
column_name	char(32)	The column name to which the statistics apply.
create_date	char(25)	The date on which statistics were gathered.
num_unique	float8	The number of unique values in the column.
rept_factor	float8	The repetition factor.
has_unique	char(8)	Contains Y if the column has unique values; otherwise, N.
pct_nulls	float8	The percentage (fraction of 1.0) of the table which contains NULL for the column.
num_cells	integer	The number of cells in the histogram.
column_domain	integer	Identifies the domain from which the column draws its values.
is_complete	char(8)	Contains Y if the column contains all possible values in its domain, N if the column does not contain all possible values in its domain, or blank if unknown.
stat_version	char(8)	Version of statistics (for example, ING6.5).
hist_data_length	integer	Length of the histogram boundary values.

The iihistograms Catalog

The iihistograms table contains histogram information used by the optimizer.

Column Name	Data Type	Description
table_name	char(32)	The table for the histogram. Must be a valid object name.
table_owner	char(32)	The table owner's user name.
column_name	char(32)	The name of the column.
text_sequence	integer	The sequence number for the histogram, numbered from 1. There may be several rows in this table, used to order the "text_segment" data when histogram is read into memory.
text_segment	char(228)	The encoded histogram data, created by optimizedb.

The iiprocedures Catalog

The iiprocedures catalog contains one or more entries for each database procedure defined on a database. Because the text of the procedure definition can contain more than 240 characters, iiprocedures may contain more than one entry for a single procedure. The text may contain newlines and may be broken mid-word across rows.

This table is keyed on procedure_name and procedure_owner:

Column Name	Data Type	Description
procedure_name	char(32)	The database procedure name, as specified in the create procedure statement.
procedure_owner	char(32)	The procedure owner's Ingres username.
create_date	char(25)	The procedure's creation date.
proc_subtype	char(8)	The subtype of this procedure. For standard Ingres procedures, this will be N (native). For Ingres Star, this may be I (imported).
text_sequence	smallint	The sequence number for the

Column Name	Data Type	Description
		test_segment.
text_segment	varchar(240)	The text of the procedure definition.
system_use	char(8)	Contains S if the procedure is system-generated, U if created by a user, or blank if unknown. Ingres generates procedures to enforce table constraints.

The iiregistrations Catalog

The iiregistrations catalog contains the text of register statements, and is used by Ingres Star and Enterprise Access products.

Column Name	Data Type	Description
object_name	char(32)	The name of the registered table, view, or index.
object_owner	char(32)	The name of the owner of the table, view, or index.
object_dml	char(8)	The language used in the registration statement. S for SQL or Q for QUEL.
object_type	char(8)	Describes the object type of object_name. The values are T if the object is a table, V if it is a view, or I if it is an index.
object_subtype	char(8)	Describes the type of table or view created by the register statement. For Ingres Star, this will be L (link). For an Enterprise Access product, this will be I (imported object).
text_sequence	smallint	The sequence number of the text field, numbered from 1.
text_segment	varchar (240)	The text of the register statement.

The iisynonyms Catalog

The iisynonyms catalog contains information about the synonyms that have been defined for the database. Entries appear in iisynonyms when a create synonym statement is issued. Entries are removed when a drop synonym statement is issued for an existing synonym, or when a drop table|view|index statement drops the table on which the synonym is defined.

Column Name	Data Type	Description
synonym_name	char(32)	The name of the synonym.
synonym_owner	char(32)	The owner of the synonym.
table_name	char(32)	The name of the table, view or index for which the synonym was created.
table_owner	char(32)	The owner of the table.

Mandatory and Ingres-Only Standard Catalogs

Mandatory catalogs are required to be present on all installations. Ingres-*only* catalogs are required for Ingres installations. This section lists the catalogs in each category. For a detailed description on all catalogs, see the *Database Administrator Guide*.

Mandatory Catalogs with Entries Required

The following catalogs must be present on both Enterprise Access product and Ingres installations. These catalogs must contain entries.

- iidbcapabilities
- iidbconstants
- iitables
- iicolumns

Mandatory Catalogs Without Entries Required

The following catalogs must be present on both Enterprise Access product and Ingres installations. However, these catalogs are not required to contain entries.

- iiphysical_tables
- iiviews
- iiindexes
- iiindex_columns
- iialt_columns
- iistats
- iihistograms
- iiaudittables
- iiconstraint_indexes
- iiconstraints
- iikeys
- iiref_constraints
- iisecurity_alarms

Ingres-Only Catalogs

The following catalogs are required by Ingres installations.

- iipermits
- iiintegrities
- iimulti_locations
- iirules
- iilog_help
- iifile_info

Appendix A: Terminal Monitor

This section contains the following topics:

Terminal Monitors

You use a terminal monitor to interactively enter, edit, and execute individual queries or files containing queries. Terminal monitors also allow operating system level commands to be executed.

There are two releases of the Terminal Monitor:

- Forms-based release
- Line-based release

This appendix describes the line-based release, and includes instructions on how to invoke the Terminal Monitor and issue queries interactively.

For information about the forms-based release of the Terminal Monitor, see the *Character-Based Querying and Reporting Tools User Guide*.

To display characters correctly, the Ingres character-based utilities like Terminal Monitor must use the correct code page setting:

Windows: Run the Terminal Monitor under the supplied Ingres command prompt, which has the correct code page and font settings.

UNIX: The code page setting for the console window on which the Terminal Monitor runs must match the character set value in II_CHARSETxx for the database.

sql Command—Access Line-based Terminal Monitor

To access the line-based SQL terminal monitor, you must type the following command at the operating system prompt:

```
sql [flags]
```

This sql command accepts a variety of flags that define how the Terminal Monitor and the DBMS Server operate during your session.

For a list of SQL option flags that can used, see the *Command Reference Guide*.

Line-mode flags are as follows:

-a

Disables the autoclear function. This means that the query buffer is never automatically cleared; it is as if the \append command was inserted after every \go. This flag requires the query buffer to be cleared using \reset after every query.

-d

Turns off the display of the dayfile (a text file that displays when the Terminal Monitor is invoked).

-s

Suppresses status messages. All messages except error messages are turned off, including login and logout messages, the dayfile, and prompts.

-vX

Sets the column separator to the character specified by X. The default is the vertical bar (|).

-P *password*

Defines the user password.

-R*role-name role-password*

Defines the role name and optional role password. Separate the name and password with a slash (/).

-history_recall

Invokes the terminal monitor with history recall functionality. The recall functionalities include the following:

left- and right- arrow

Browses the line entered.

Backspace

Erases a character to the left of the cursor.

Up- and Down- arrow

Retrieves the history of the commands typed in this session.

Ctrl+U

Erases the line.

Ctrl+K

Erases the line from the cursor to the end.

Note: The history recall feature is available on Linux platforms only.

Terminal Monitor Query Buffering

In the Terminal Monitor, each query that is typed is placed in a query buffer, rather than executed immediately. The queries are executed when the execution command (\go or \g) is typed. The results, by default, appear on your terminal.

For example, assume you have a table called, employee, that lists all employees in your company. If you want to see a list of those employees who live in a particular city (cityA), enter the following statement:

```
select name from employee where city=cityA
\g
```

The query is placed in the query buffer and executed when you enter \g. The returned rows display on your terminal. (If you type \g twice, your query is executed twice.)

Several other operations can also be performed on the query buffer, including:

- Editing the contents.
- Printing the contents.
- Writing the contents to another file.

After a \go command the query buffer is cleared if another query is typed in, unless a command that affects the query buffer is typed first. Commands that retain the query buffer contents are:

```
\append     or     \a
\edit       or     \e
\print      or     \p
\bell
\nobell
```

For example, typing:

```
help parts
\go
select * from parts
```

results in the query buffer containing:

```
select * from parts
```

Whereas, typing:

```
help parts
\go
\print
select * from parts
```

results in the query buffer containing:

```
help parts
select * from parts
```

This feature can be overridden by executing the \append command before executing the \go command, or by specifying the -a flag when issuing the sql command to begin your session.

Terminal Monitor Commands

Terminal Monitor commands can manipulate the contents of the query buffer or your environment. Unlike the SQL statements that are typed into the Terminal Monitor, terminal monitor commands are executed as soon as the Return key is pressed.

All Terminal Monitor commands must be preceded with a backslash (\). If a backslash is entered literally, it must be enclosed in quotes. For example, the following statement inserts a backslash into the Test table:

```
insert into test values('\')\g
```

Some Terminal Monitor commands accept a file name as an argument. These commands must appear alone on a single line. The Terminal Monitor interprets all characters appearing on the line after such commands as a file name. Those Terminal Monitor commands that do not accept arguments can be stacked on a single line. For example:

```
\date\go\date
```

returns the date and time before and after execution of the current query buffer.

Terminal Monitor commands include:

\g *or* \go

Processes the current query. The contents of the buffer are transmitted to the DBMS Server and run.

\r *or* \reset

Erases the entire query (reset the query buffer). The former contents of the buffer are lost and cannot be retrieved.

\p *or* \print

Prints the current query. The contents of the buffer are printed on the user terminal.

\e *or* \ed *or* \edit *or* \editor [*filename*]

Invokes a text editor.

Windows and UNIX: Enter the text editor of the operating system (designated by the startup file). Use the appropriate editor exit command to return to the Terminal Monitor. If no file name is given, the current contents of the query buffer are sent to the editor, and upon return, the query buffer is replaced with the edited query. If a file name is given, the query buffer is written to that file. On exit from the editor, the file contains the edited query, but the query buffer remains unchanged.

VMS: Enter the text editor. See the VAX EDT Editor Manual. Use the EDT command exit or the sequence of commands, write followed by quit, to return to the Terminal Monitor. If no file name is given, the current contents of the query buffer are sent to the editor, and upon return, the query buffer is replaced with the edited query. If a file name is given, the query buffer is written to that file, and on exit from the editor, the file contains the edited query, but the workspace remains unchanged

\time *or* \date

Prints the current time and date.

\a *or* \append

Appends to the query buffer. Typing \append after completion of a query overrides the auto-clear feature and guarantees that the query buffer is not reset until executed again.

\s *or* \sh *or* \shell

Escapes to the operating system.

UNIX: Escapes to the UNIX shell (command line interpreter). Press Ctrl+D to exit the shell and return to the Terminal Monitor.

VMS: Escapes to the command line interpreter to execute VMS commands. The VAX command line interpreter (DCL) is initiated. Type the logout command to exit DCL and return to the Terminal Monitor.

\q *or* \quit

Exits the Terminal Monitor

\cd *or* \chdir *dir_name*

Changes the working directory of the monitor to the named directory.

\i *or* \include or \read filename

Reads the named file into the query buffer. Backslash characters in the file are processed as they are read.

\w *or* \write filename

Writes the contents of the query buffer to the named file.

\script [*filename*]

Writes/stops writing the subsequent SQL statements and their results to the specified file. If no file name is supplied with the \script command, output is logged to a file called script.ing in the current directory.

The \script command toggles between logging and not logging your session to a file. If a file name is supplied on the \script command that terminates logging to a file, the file name is ignored.

This command can be used to save result tables from SQL statements for output. The \script command does not impede the terminal output of your session.

\[no]suppress

Suppresses (\suppress) or does not suppress (\nosuppress) the printing of the resulting data that is returned from the query.

\[no]bell

Includes (\bell) or does not include (\nobell) a bell (that is, Ctrl+G) with the continue or go prompt. The default is \nobell.

\[no]continue

Continues (\continue) or does not continue (\nocontinue) statement processing on error. In either case, the error message displays. The command can be abbreviated to \co (\continue) or \noco (\nocontinue). The default action is to continue. This command can be used to change that behavior. The default can also be changed by setting II_TM_ON_ERROR (which is described in the *System Administrator Guide*).

Terminal Monitor Messages and Prompts

The Terminal Monitor has a variety of messages to keep you informed of its status and that of the query buffer.

When logging in, the Terminal Monitor prints a login message that tells the release number and the login time. Following that message, the dayfile appears.

When the Terminal Monitor is ready to accept input and the query buffer is empty, the message go appears. The message, continue, appears instead if there is something in the query buffer.

The prompt >>editor indicates that you are in the text editor.

Terminal Monitor Character Input and Output

When non-printable ASCII characters are entered through the Terminal Monitor, the Terminal Monitor replaces these characters with blanks. Whenever this occurs, the Terminal Monitor displays the message:

```
Non-printing character nnn converted to blank
```

where *nnn* is replaced with the actual character.

For example, if you enter the statement:

```
insert into test values('^La')
```

the Terminal Monitor converts the ^L to a blank before sending it to the DBMS Server and displays the message described above.

To insert non-printing data into a char or varchar field, specify the data as a hexadecimal value. For example:

```
insert into test values (x'07');
```

This feature can be used to insert a newline character into a column:

```
insert into test values ('Hello world'+x'0a');
```

This statement inserts 'Hello world\n' into the test table.

On output, if the data type is char or varchar, any binary data are shown as octal numbers (\000, \035, and so on.). To avoid ambiguity, any backslashes present in data of the char or varchar type are displayed as double backslashes. For example, if you insert the following into the test table:

```
insert into test values('\aa')
```

when you retrieve that value, you see:

```
\\aa
```

The Help Statement

The Help statement displays information about a variety of SQL statements and features. For a complete list of help options, see Help in the chapter "SQL Statements."

Aborting the Editor (VMS only)

Important! In VMS environments, do not type Ctrl+Y and Ctrl+C while escaped to an editor (unless the editor assigns its own meaning to Ctrl+C) or VMS. VMS does not properly signal these events to the initiating process.

Appendix B: Keywords

This section contains the following topics:

Reserved Keywords and Identifiers

You should avoid assigning object names that use the keywords listed in this section. In addition to the keywords, all identifiers starting with the letters ii are reserved for Ingres system catalogs.

The keywords listed do not necessarily correspond to supported Ingres features. Some words are reserved for future or internal use, and some words are reserved to provide backward compatibility with older features.

Abbreviations Used in Keyword Lists

Column headings in the tables in this appendix use the following abbreviations:

- **ISQL** (Interactive SQL)—keywords reserved by the DBMS
- **ESQL** (Embedded SQL)—keywords reserved by the SQL preprocessors
- **IQUEL** (Interactive QUEL)—keywords reserved by the DBMS
- **EQUEL** (Embedded QUEL)—keywords reserved by the QUEL preprocessors
- **4GL**—keywords reserved in the context of SQL or QUEL in Ingres 4GL routines

Note: The ESQL and EQUEL preprocessors also reserve forms statements. For details about forms statements, see the *Forms-based Application Development Tools User Guide*.

Reserved Single Keywords

The following table displays OpenSQL keywords:

	SQL	QUEL

Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
abort	*	*		*	*	
activate	*	*		*	*	
add	*	*		*	*	
addform	*	*		*	*	
after	*	*		*	*	
all	*	*		*	*	
alter	*	*		*	*	
and	*	*		*	*	
any	*	*		*	*	
append	*	*		*	*	
array	*	*		*	*	
as	*	*		*	*	
asc	*	*		*	*	
at	*	*		*	*	
authorization	*	*		*	*	
avg	*	*		*	*	
avgu	*	*		*	*	
before	*	*		*	*	
begin	*	*		*	*	
between	*	*		*	*	
breakdisplay	*	*		*	*	
by	*	*		*	*	
byref	*	*		*	*	
call	*	*		*	*	
callframe	*	*		*	*	
callproc	*	*		*	*	
cascade	*	*		*	*	
case	*	*				
cast	*					
check	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
clear	*	*		*	*	
clearrow	*	*		*	*	
close	*	*		*	*	
coalesce	*					
column	*	*		*	*	
command	*	*		*	*	
commit	*	*		*	*	
committed	*	*		*	*	
connect	*	*		*	*	
constraint	*	*		*	*	
continue	*	*		*	*	
copy	*	*		*	*	
copy_from	*					
copy_into	*					
count	*	*		*	*	
countu	*	*		*	*	
create	*	*		*	*	
current	*	*		*	*	
current_user	*	*		*	*	
curval	*	*				
cursor	*	*		*	*	
cycle	*	*				
datahandler	*	*		*	*	
dbms_password		*				
declare	*	*		*	*	
default	*	*		*	*	
define	*	*		*	*	
delete	*	*		*	*	
deleterow	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
desc	*	*		*	*	
describe	*	*		*	*	
descriptor	*	*		*	*	
destroy	*	*		*	*	
direct	*	*		*	*	
disconnect	*	*		*	*	
display	*	*		*	*	
distinct	*	*		*	*	
distribute	*	*		*	*	
do	*	*		*	*	
down	*	*		*	*	
drop	*	*		*	*	
else	*	*		*	*	
elseif	*	*		*	*	
enable	*	*		*	*	
end	*	*		*	*	
end-exec	*	*		*	*	
enddata	*	*		*	*	
enddisplay	*	*		*	*	
endfor	*					
endforms	*	*		*	*	
endif	*	*		*	*	
endloop	*	*		*	*	
endrepeat	*					
endretrieve	*	*		*	*	
endselect	*	*		*	*	
endwhile	*	*		*	*	
escape	*	*		*	*	
except	*					

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
exclude	*	*		*	*	
excluding	*	*		*	*	
execute	*	*		*	*	
exists	*	*		*	*	
exit	*	*		*	*	
fetch	*	*		*	*	
field	*	*		*	*	
finalize	*	*		*	*	
first	*	*				
for	*	*		*	*	
foreign	*	*		*	*	
formdata	*	*		*	*	
forminit	*	*		*	*	
forms	*	*		*	*	
from	*	*		*	*	
full	*	*		*	*	
get	*	*		*	*	
getform	*	*		*	*	
getoper	*	*		*	*	
getrow	*	*		*	*	
global	*	*		*	*	
goto	*	*		*	*	
grant	*	*		*	*	
granted		*				
group	*	*		*	*	
having	*	*		*	*	
help	*	*		*	*	
help_forms	*	*		*	*	
help_frs	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
helpfile	*	*		*	*	
identified	*	*		*	*	
if	*	*		*	*	
iimessage	*	*		*	*	
iiprintf	*	*		*	*	
iiprompt	*	*		*	*	
iistatement	*	*		*	*	
immediate	*	*		*	*	
import	*	*		*	*	
in	*	*		*	*	
include	*	*		*	*	
increment	*	*				
index	*	*		*	*	
indicator	*	*		*	*	
ingres	*	*		*	*	
initial_user	*	*		*	*	
initialize	*	*		*	*	
inittable	*	*		*	*	
inquire_equel	*	*		*	*	
inquire_forms	*	*		*	*	
inquire_frs	*	*		*	*	
inquire_ingres	*	*		*	*	
inquire_sql	*	*		*	*	
insert	*	*		*	*	
insertrow	*	*		*	*	
integrity	*	*		*	*	
intersect	*					
into	*	*		*	*	
is	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
isolation	*	*		*	*	
join	*	*		*	*	
key	*	*		*	*	
leave	*					
left	*	*		*	*	
level	*	*		*	*	
like	*	*		*	*	
loadtable	*	*		*	*	
local	*	*		*	*	
max	*	*		*	*	
maxvalue	*	*				
menuitem	*	*		*	*	
message	*	*		*	*	
min	*	*		*	*	
minvalue	*	*				
mode	*	*		*	*	
modify	*	*		*	*	
module	*	*		*	*	
move	*	*		*	*	
natural	*	*		*	*	
next	*	*		*	*	
nextval	*	*				
nocache	*	*				
nocycle	*	*				
noecho	*	*		*	*	
nomaxvalue	*	*				
nominvalue	*	*				
noorder	*	*				
not	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
notrim	*	*		*	*	
null	*	*		*	*	
nullif	*					
of	*	*		*	*	
on	*	*		*	*	
only	*	*		*	*	
open	*	*		*	*	
option	*	*		*	*	
or	*	*		*	*	
order	*	*		*	*	
out	*	*		*	*	
outer	*	*		*	*	
param	*	*		*	*	
partition		*				
permit	*	*		*	*	
prepare	*	*		*	*	
preserve	*	*		*	*	
primary	*	*		*	*	
print	*	*		*	*	
printscreen	*	*		*	*	
privileges	*	*		*	*	
procedure	*	*		*	*	
prompt	*	*		*	*	
public	*	*		*	*	
purgetable		*			*	
putform	*	*		*	*	
putoper	*	*		*	*	
putrow	*	*		*	*	
qualification	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
raise	*	*		*	*	
range	*	*		*	*	
rawpct	*	*				
read	*	*		*	*	
redisplay	*	*		*	*	
references	*	*		*	*	
referencing	*	*		*	*	
register	*	*		*	*	
relocate	*	*		*	*	
remove	*	*		*	*	
rename	*	*		*	*	
repeat	*	*		*	*	
repeatable	*	*		*	*	
repeated	*	*		*	*	
replace	*	*		*	*	
replicate	*	*		*	*	
restart	*	*				
restrict	*	*		*	*	
result	*	*				
resume	*	*		*	*	
retrieve	*	*		*	*	
return	*	*		*	*	
revoke	*	*		*	*	
right	*	*		*	*	
roll		*				
rollback	*	*		*	*	
row	*	*				
rows	*	*		*	*	
run	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
save	*	*		*	*	
savepoint	*	*		*	*	
schema	*	*		*	*	
screen	*	*		*	*	
scroll	*	*		*	*	
scrolldown	*	*		*	*	
scrollup	*	*		*	*	
section	*	*		*	*	
select	*	*		*	*	
serializable	*	*		*	*	
session	*	*		*	*	
session_user	*	*		*	*	
set	*	*		*	*	
set_4gl	*	*		*	*	
set_equel	*	*		*	*	
set_forms	*	*		*	*	
set_frs	*	*		*	*	
set_ingres	*	*		*	*	
set_sql	*	*		*	*	
sleep	*	*		*	*	
some	*	*		*	*	
sort	*	*		*	*	
sql	*	*		*	*	
start	*	*				
stop	*	*		*	*	
submenu	*	*		*	*	
substring	*	*				
sum	*	*		*	*	
sumu	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
system	*	*		*	*	
system_ maintained	*	*		*	*	
system_user	*	*		*	*	
table	*	*		*	*	
tabledata	*	*		*	*	
temporary	*	*		*	*	
then	*	*		*	*	
to	*	*		*	*	
type	*	*		*	*	
uncommitted	*	*				
union	*	*		*	*	
unique	*	*		*	*	
unloadtable	*	*		*	*	
until	*	*		*	*	
up	*	*		*	*	
update	*	*		*	*	
user	*	*		*	*	
using	*	*		*	*	
validate	*	*		*	*	
validrow	*	*		*	*	
values	*	*		*	*	
view	*	*		*	*	
when	*	*		*	*	
whenever	*	*		*	*	
where	*	*		*	*	
while	*	*		*	*	
with	*	*		*	*	
work	*	*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
write	*	*				

Reserved Double Keywords

The following table lists OpenSQL double keywords:

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
after field	*	*		*	*	
alter default		*				
alter group	*	*		*	*	
alter location	*	*		*	*	
alter profile	*					
alter role	*	*		*	*	
alter security_audit	*	*		*	*	
alter_sequence	*	*				
alter table	*	*		*	*	
alter user	*	*		*	*	
array of	*	*		*	*	
base table structure	*					
before field	*	*		*	*	
begin transaction	*	*		*	*	
by role	*					
by user	*	*		*	*	
call on	*	*		*	*	
call procedure	*	*		*	*	
class of	*	*		*	*	
clear array		*				
close cursor	*	*		*	*	

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
comment on	*	*		*	*	
connect to	*	*		*	*	
copy table	*	*		*	*	
create dbevent	*	*		*	*	
create domain		*				
create group	*	*		*	*	
create integrity	*	*		*	*	
create link	*	*		*	*	
create location	*	*		*	*	
create permit	*	*		*	*	
create procedure	*	*		*	*	
create profile	*	*	*			
create role	*	*		*	*	
create rule	*	*		*	*	
create security_alarm	*	*		*	*	
create sequence	*	*				
create synonym	*	*		*	*	
create user	*	*		*	*	
create view	*	*		*	*	
cross join	*	*				
curr value	*					
current installation	*	*		*	*	
current value	*	*				
define cursor	*	*		*	*	
declare cursor	*	*		*	*	
define integrity	*	*		*	*	
define link	*	*		*	*	
define location	*	*		*	*	
define permit	*	*		*	*	

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
define qry	*	*		*	*	
define query	*	*		*	*	
define view	*	*		*	*	
delete cursor	*	*		*	*	
describe form		*				
destroy integrity	*	*		*	*	
destroy link	*	*		*	*	
destroy permit	*	*		*	*	
destroy table	*	*		*	*	
destroy view	*	*		*	*	
direct connect	*	*		*	*	
direct disconnect	*	*		*	*	
direct execute	*	*		*	*	
disable security_audit	*	*		*	*	
disconnect current	*	*		*	*	
display submenu	*	*		*	*	
drop dbevent	*	*		*	*	
drop domain		*				
drop group	*	*		*	*	
drop integrity	*	*		*	*	
drop link	*	*		*	*	
drop location	*	*		*	*	
drop permit	*	*		*	*	
drop procedure	*	*		*	*	
drop profile	*	*				
drop role	*	*		*	*	
drop rule	*	*		*	*	
drop security_alarm	*	*		*	*	
drop sequence	*	*				

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
drop synonym	*	*		*	*	
drop user	*	*		*	*	
drop view	*	*		*	*	
each row		*				
each statement		*				
enable security_audit	*	*		*	*	
end exclude		*				
end transaction	*	*		*	*	
exec sql	*	*		*	*	
execute immediate	*	*		*	*	
execute on	*	*		*	*	
execute procedure	*	*		*	*	
foreign key	*	*		*	*	
for deferred	*	*		*	*	
for direct	*	*		*	*	
for readonly	*	*		*	*	
for retrieve	*	*		*	*	
for update	*	*		*	*	
from group	*	*		*	*	
from role	*	*		*	*	
from user	*	*		*	*	
full join	*	*		*	*	
full outer	*					
get attribute		*				
get data		*		*	*	
get dbevent		*		*	*	
get global		*				
global temporary	*	*		*	*	
help all		*				

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
help comment	*	*		*	*	
help integrity	*	*		*	*	
help permit	*	*		*	*	
help table	*	*		*	*	
help view	*	*		*	*	
identified by	*	*		*	*	
inner join	*	*		*	*	
is null	*	*		*	*	
isolation level		*		*		
left join	*	*		*	*	
left outer	*		*			
modify table	*	*		*	*	
next value	*	*				
no cache	*	*				
no cycle	*	*				
no maxvalue	*	*				
no minvalue	*	*				
no order	*					
not like	*	*		*	*	
not null	*	*		*	*	
on commit	*	*		*	*	
on current	*	*		*	*	
on database	*	*		*	*	
on dbevent	*	*		*	*	
on location	*	*		*	*	
on procedure	*	*		*	*	
on sequence	*					
only where	*	*		*	*	
open cursor	*	*		*	*	

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
order by	*	*		*	*	
primary key	*	*		*	*	
procedure returning	*	*		*	*	
put data	*	*		*	*	
raise dbevent	*	*		*	*	
raise error	*	*		*	*	
read only		*				
read write		*				
register dbevent	*	*		*	*	
register table	*	*		*	*	
register view	*	*		*	*	
remote system_password		*				
remote system_user		*				
remove dbevent	*	*		*	*	
remove table	*	*		*	*	
remove view	*	*		*	*	
replace cursor	*	*		*	*	
result row	*	*				
resume entry	*	*		*	*	
resume menu	*	*		*	*	
resume next	*	*		*	*	
retrieve cursor	*	*		*	*	
right join	*	*		*	*	
run submenu	*	*		*	*	
session group	*	*		*	*	
session role	*	*		*	*	
session user	*	*		*	*	
set aggregate	*	*		*	*	

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set autocommit	*	*		*	*	
set connection		*				
set cpufactor	*	*		*	*	
set date_format	*	*		*	*	
set ddl_concurrency	*	*		*	*	
set decimal	*	*		*	*	
set flatten	*			*		
set global		*				
set hash	*			*		
set io_trace	*	*		*	*	
set jcpufactor	*	*		*	*	
set joinop	*	*		*	*	
set journaling	*	*		*	*	
set lock_trace	*	*		*	*	
set lockmode	*	*		*	*	
set log_trace	*	*		*	*	
set logdbevents	*	*		*	*	
set logging	*	*		*	*	
set maxconnect	*			*		
set maxcost	*	*		*	*	
set maxcpu	*	*		*	*	
set maxidle	*			*		
set maxio	*	*		*	*	
set maxpage	*	*		*	*	
set maxquery	*	*		*	*	
set maxrow	*	*		*	*	
set money_format	*	*		*	*	
set money_prec	*	*		*	*	
set noflatten	*			*		

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set nohash	*					
set noio_trace	*	*		*	*	
set nojoinop	*	*		*	*	
set nojournaling	*	*		*	*	
set nolock_trace	*	*		*	*	
set nolog_trace	*	*		*	*	
set nologdbevents	*	*		*	*	
set nologging	*	*		*	*	
set nomaxconnect	*			*		
set nomaxcost	*	*		*	*	
set nomaxcpu	*	*		*	*	
set nomaxidle	*			*		
set nomaxio	*	*		*	*	
set nomaxpage	*	*		*	*	
set nomaxquery	*	*		*	*	
set nomaxrow	*	*		*	*	
set noojflatten	*					
set nooptimizeonly	*	*		*	*	
set noparallel	*					
set noprintdbevents	*	*		*	*	
set noprintqry	*	*		*	*	
set noprintrules	*	*		*	*	
set noqep	*	*		*	*	
set norules	*	*		*	*	
set nosql	*	*		*	*	
set nostatistics	*	*		*	*	
set notrace	*	*		*	*	
set nounicode_substitution	*					

	SQL			QUEL		
Double Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
set ojflatten	*					
set optimizeonly	*	*		*	*	
set parallel	*					
set printdbevents	*	*		*	*	
set printqry	*	*		*	*	
set printrules	*					
set qep	*	*		*	*	
set random_seed	*			*		
set result_structure	*	*		*	*	
set ret_into				*	*	
set role	*					
set rules	*	*		*	*	
set session	*	*		*	*	
set sql	*	*		*	*	
set statistics	*	*		*	*	
set trace	*	*		*	*	
set unicode_substitution	*					
set update_rowcount	*			*		
set work	*	*		*	*	
system user	*	*		*	*	
to group	*	*		*	*	
to role	*	*		*	*	
to user	*	*		*	*	
user authorization	*	*		*	*	
with null	*	*		*	*	
with short_remark	*	*		*	*	

ANSI/ISO SQL Keywords

The following keywords are ANSI/ISO standard keywords that are not reserved in Ingres SQL or Ingres Embedded SQL. You may want to treat these as reserved words to ensure compatibility with other implementations of SQL.

absolute

action

allocate

alter

are

asc

asensitive

assertion

atomic

atomic

bit

bit_length

both

called

cardinality

cascaded

case

cast

catalog

char

char_length

character

character_length

coalesce

collate

collation

collect

condition
connection
constraints
convert
corr
corresponding
cross
cube
current_date
current_default_transform_group
current_path
current_role
current_time
current_timestamp

date
day
deallocate
dec
decimal
deferrable
deferred
deref
desc
deterministic
diagnostics
domain
double
dynamic

each
element
else

every
except
exception
exec
external
extract

false
filter
first
float
found
free
function
fusion

get
go
grouping

hold
hour

identity
initially
inout
input
insensitive
int
integer
intersect
intersection
intersects

interval
isolation

language
large
last
lateral
leading
level
ln
localtime
localtimestamp
lower

match
member
merge
method
minute
mod
modifies
module
month
multiset

names
national
nchar
new
no
none
normalize
nullif

numeric

octet_length
old
only
option
outer
output
over
overlaps
overlay

pad
parameter
partial
partition
position
precision
prior
privileges

rank
read
reads
real
recursive
ref
relative
release
returns
rollup
row_number

scope
search
second
sensitive
similar
size
smallint
space
specific
specifictype
sql
sqlcode
sqlerror
sqlexception
sqlstate
sqlwarning
static
submultiset
substring
symmetric

tablesample
then
time
timestamp
timezone_hour
timezone_minute
trailing
transaction
translate
translation
treat
trigger

trim
true

uescape
unknown
unnest
upper
usage

value
varchar
varying

width_bucket
window
within
without
work
write

year
zone

Appendix C: SQLSTATE Values and Generic Error Codes

This section contains the following topics:

How Error Code Mapping Works

Error code mapping works as follows:

- Proprietary error codes - Each host DBMS returns a set of proprietary error codes. These error codes are unique to the DBMS and therefore not useful for developing portable applications. Enterprise Access products map proprietary error codes to generic error codes. This is a many-to-one mapping: many proprietary error codes may map to a single generic error code. For details about proprietary error codes, see your host DBMS documentation.
- Generic error codes - Enterprise Access products return a consistent set of errors. To enable your application to interact with different host DBMS (through Enterprise Access products), your applications should check generic error codes.
- SQLSTATE - SQLSTATE is the ANSI standard error variable for returning errors to applications. If you are developing ANSI-compliant applications, your application should check SQLSTATE. The mapping of generic errors to the SQLSTATE is many-to-one: many generic errors may map to a single SQLSTATE value.

SQLSTATE Values

SQLSTATE is the ANSI/ISO Entry SQL-92-compliant method for returning errors to applications. The following table lists the values returned in SQLSTATE. An asterisk in the Ingres Only column indicates a value that is specified by ANSI as vendor-defined.

Note: The first two characters of the SQLSTATE are a class of errors and the last three a subclass. The codes that end in 000 are the names of the class.

SQLSTATE	Ingres Only	Description
00000		Successful completion
01000		Warning
01001		Cursor operation conflict
01002		Disconnect error
01003		Null value eliminated in set function
01004		String data, right truncation
01005		Insufficient item descriptor areas
01006		Privilege not revoked
01007		Privilege not granted
01008		Implicit zero-bit padding
01009		Search condition too long for information schema
0100A		Query expression too long for information schema
01500	*	LDB table not dropped
01501	*	DSQL update or delete affects entire table
02000		No data
07000		Dynamic SQL error
07001		Using clause does not match dynamic parameter specification
07002		Using clause does not match target specification
07003		Cursor specification cannot be executed
07004		Using clause required for dynamic parameters

SQLSTATE	Ingres Only	Description
07005		Prepared statement not a cursor specification
07006		Restricted data type attribute violation
07007		Using clause required for result fields
07008		Invalid descriptor count
07009		Invalid descriptor index
07500	*	Context mismatch
08000		Connection exception
08001		SQL-client unable to establish SQL-connection
08002		Connection name in use
08003		Connection does not exist
08004		SQL-server rejected establishment of SQL-connection
08006		Connection failure
08007		Transaction resolution unknown
08500	*	LDB is unavailable
0A000		Feature not supported
0A001		Multiple server transactions
0A500	*	Invalid query language
21000		Cardinality violation
22001		String data, right truncation
22002		Null value, no indicator parameter
22003		Numeric value out of range
22005		Error in assignment
22007		Invalid datetime format
22008		Datetime field overflow
22009		Invalid time zone displacement value
22011		Substring error
22012		Division by zero
22015		Interval field overflow
22018		Invalid character value for cast

SQLSTATE	Ingres Only	Description
22019		Invalid escape character
22021		Character not in repertoire
22022		Indicator overflow
22023		Invalid parameter value
22024		Unterminated C string
22025		Invalid escape sequence
22026		String data, length mismatch
22027		Trim error
22500	*	Invalid data type
23000		Integrity constraint violation
24000		Invalid cursor state
25000		Invalid transaction state
26000		Invalid SQL statement name
27000		Triggered data change violation
28000		Invalid authorization specification
2A000		Syntax error or access rule violation in direct SQL statement
2A500	*	Table not found
2A501	*	Column not found
2A502	*	Duplicate object name
2A503	*	Insufficient privilege
2A504	*	Cursor not found
2A505	*	Object not found
2A506	*	Invalid identifier
2A507	*	Reserved identifier
2B000		Dependent privilege descriptors still exist
2C000		Invalid character set name
2D000		Invalid transaction termination
2E000		Invalid connection name
33000		Invalid SQL descriptor name

SQLSTATE	Ingres Only	Description
34000		Invalid cursor name
35000		Invalid condition number
37000		Syntax error or access rule violation in SQL dynamic statement
37500	*	Table not found
37501	*	Column not found
37502	*	Duplicate object name
37503	*	Insufficient privilege
37504	*	Cursor not found
37505	*	Object not found
37506	*	Invalid identifier
37507	*	Reserved identifier
3C000		Ambiguous cursor name
3D000		Invalid catalog name
3F000		Invalid schema name
40000		Transaction rollback
40001		Serialization failure
40002		Integrity constraint violation
40003		Statement completion unknown
42000		Syntax error or access rule violation
42500	*	Table not found
42501	*	Column not found
42502	*	Duplicate object name
42503	*	Insufficient privilege
42504	*	Cursor not found
42505	*	Object not found
42506	*	Invalid identifier
42507	*	Reserved identifier
44000		With check option violation
50000	*	Miscellaneous Ingres-specific errors

SQLSTATE	Ingres Only	Description
50001	*	Invalid duplicate row
50002	*	Limit has been exceeded
50003	*	Resource exhausted
50004	*	System configuration error
50005	*	Enterprise Access product-related error
50006	*	Fatal error
50007	*	Invalid SQL statement id
50008	*	Unsupported statement
50009	*	Database procedure error raised
5000A	*	Query error
5000B	*	Internal error
5000D	*	Invalid cursor name
5000E	*	Duplicate SQL statement id
5000F	*	Textual information
5000G	*	Database procedure message
5000H	*	Unknown/unavailable resource
5000I	*	Unexpected LDB schema change
5000J	*	Inconsistent DBMS catalog
5000K	*	SQLSTATE status code unavailable
5000L	*	Protocol error
5000M	*	IPC error
HZ000		Remote Database Access

Generic Error Codes

Generic error codes are error codes that map to DBMS-specific errors returned by Ingres and by the DBMS that you access through Enterprise Access products. If your application interacts with more than one type of DBMS, it should check generic errors in order to remain portable.

The following table describes generic error codes:

Error Code	Message	Explanation
+00050	Warning message	The request was successfully completed, but a warning was issued.
+00100	No more data	A request for data was processed, but either no data or no more data fitting the requested characteristics was found.
00000	Successful completion	The request completed normally with no errors or unexpected conditions occurring.
-30100	Table not found	A table referenced in a statement doesn't exist or is owned by another user. This error can also be returned concerning an index or a view.
-30110	Column not known or not in table	A column referenced in a statement is not found.
-30120	Unknown cursor	An invalid or unopened cursor name or identifier was specified or referenced in a statement.
-30130	Other database object not found	A database object other than a table, view, index, column or cursor was specified or referenced in a statement, but is not identified or located. This applies to a database procedure, a grant or permission, a rule, or other object.
-30140	Other unknown or unavailable resource	A resource, of a type other than one mentioned above, is either not known or unavailable for the request.
-30200	Duplicate resource definition	An attempt to define a database object (such as a table) was made, but the object already exists.

Error Code	Message	Explanation
-30210	Invalid attempt to insert duplicate row	A request to insert a row was refused; the table does not accept duplicates, or there is a unique index defined on the table.
-31000	Statement syntax error	The statement just processed had a syntax error.
-31100	Invalid identifier	An identifier, such as a table name, cursor name or identifier, procedure name, was invalid because it contained incorrect characters or been too long.
-31200	Unsupported query language	A request to use an unrecognized or unsupported query language was made.
-32000	Inconsistent or incorrect query specification	A query, while syntactically correct, was logically inconsistent, conflicting or otherwise incorrect.
-33000	Runtime logical error	An error occurred at runtime. An incorrect specification was made, an incorrect host variable value or type was specified or some other error not detected until runtime was found.
-34000	Not privileged/ restricted operation	An operation was rejected because the user did not have appropriate permission or privileges to perform the operation, or the operation was restricted (for example, to a certain time of day) and the operation was requested at the wrong time or in the wrong mode.
-36000	System limit exceeded	A system limit was exceeded during query processing, for example, number of columns, size of a table, row length, or number of tables in a query.
-36100	Out of needed resource	The system exhausted, or did not have enough of, a resource such as memory or temporary disk space required to complete the query.
-36200	System configuration error	An error in the configuration of the system was detected.

Error Code	Message	Explanation
-37000	Communication/ transmission error	The connection between the DBMS and the client failed.
-38000	Error within an Enterprise Access product	An error occurred in an Enterprise Access product or DBMS interface.
-38100	Host system error	An error occurred in the host system.
-39000	Fatal error - session terminated	A severe error occurred which has terminated the session with the DBMS or the client.
-39100	Unmappable error	An error occurred which is not mapped to a generic error.
-40100	Cardinality violation	A request tried to return more or fewer rows than allowed. This usually occurs when a singleton select request returns more than one row, or when a nested subquery returns an incorrect number of rows.
-402*dd*	Data exception	A data handling error occurred. The subcode *dd* defines the type of error.
-40300	Constraint violation	A DBMS constraint, such as a referential integrity or the check option on a view was violated. The request was rejected.
-40400	Invalid cursor state	An invalid cursor operation was requested; for example, an update request was issued for a read-only cursor.
-40500	Invalid transaction state	A request was made that was invalid in the current transaction state. For example, an update request was issued in a read-only transaction, or a request was issued improperly in or out of a transaction.
-40600	Invalid SQL statement identifier	An identifier for an SQL statement, such as a repeat query name, was invalid.
-40700	Triggered data change violation	A change requested by a cascaded referential integrity change was invalid.

Error Code	Message	Explanation
-41000	Invalid user authorization identifier	An authorization identifier, usually a user name, was invalid.
-41200	Invalid SQL statement	Unlike generic error -31000 (statement syntax error), this was a recognized statement that is either currently invalid or unsupported.
-41500	Duplicate SQL statement identifier	An identifier for an SQL statement, such as a repeat query name, was already active or known.
-49900	Serialization failure (Deadlock)	An error occurred (for example, deadlock, timeout, forced abort, log file full) that caused the query to be rejected. If the transaction is rejected, it is rolled back except in the case of a timeout. (Check SQLWARN6 in the SQLCA structure.) The query or transaction can be resubmitted.

Generic Error Data Exception Subcodes

The following table lists subcodes returned with generic error -402 (generic errors -40200 through -40299):

Subcode	Description
00	No subcode
01	Character data truncated from right
02	Null value, no indicator variable specified
03	Exact numeric data, loss of significance (decimal overflow)
04	Error in assignment
05	Fetch orientation has value of zero
06	Invalid date or time format
07	Date/time field overflow
08	Reserved
09	Invalid indicator variable value
10	Invalid cursor name

Subcode	Description
15	Invalid data type
20	Fixed-point overflow
21	Exponent overflow
22	Fixed-point divide
23	Floating point divide
24	Decimal divide
25	Fixed-point underflow
26	Floating point underflow
27	Decimal underflow
28	Other unspecified math exception
99	Maximum legal subcode

SQLSTATE and Equivalent Generic Errors

The following table lists the correspondence between SQLSTATE values and Ingres generic errors:

SQLSTATE	Generic Error
00000	E_GE0000_OK
01000	E_GE0032_WARNING
01001	E_GE0032_WARNING
01002	E_GE0032_WARNING
01003	E_GE0032_WARNING
01004	E_GE0032_WARNING
01005	E_GE0032_WARNING
01006	E_GE0032_WARNING
01007	E_GE0032_WARNING
01008	E_GE0032_WARNING
01009	E_GE0032_WARNING
0100A	E_GE0032_WARNING
01500	E_GE0032_WARNING

SQLSTATE	Generic Error
01501	E_GE0032_WARNING
02000	E_GE0064_NO_MORE_DATA
07000	E_GE7D00_QUERY_ERROR
07001	E_GE7D00_QUERY_ERROR
07002	E_GE7D00_QUERY_ERROR
07003	E_GE7D00_QUERY_ERROR
07004	E_GE7D00_QUERY_ERROR
07005	E_GE7D00_QUERY_ERROR
07006	E_GE7D00_QUERY_ERROR
07007	E_GE7D00_QUERY_ERROR
07008	E_GE7D00_QUERY_ERROR
07009	E_GE7D00_QUERY_ERROR
07500	E_GE98BC_OTHER_ERROR
08000	E_GE98BC_OTHER_ERROR
08001	E_GE98BC_OTHER_ERROR
08002	E_GE80E8_LOGICAL_ERROR
08003	E_GE80E8_LOGICAL_ERROR
08004	E_GE94D4_HOST_ERROR
08006	E_GE9088_COMM_ERROR
08007	E_GE9088_COMM_ERROR
08500	E_GE75BC_UNKNOWN_OBJECT
0A000	E_GE98BC_OTHER_ERROR
0A001	E_GE98BC_OTHER_ERROR
0A500	E_GE79E0_UNSUP_LANGUAGE
21000	E_GE9CA4_CARDINALITY
22000	E_GE9D08_DATAEX_NOSUB
22001	E_GE9D09_DATAEX_TRUNC
22002	E_GE9D0A_DATAEX_NEED_IND
22003	E_GE9D0B_DATAEX_NUMOVR
22003	E_GE9D1C_DATAEX_FIXOVR

SQLSTATE	Generic Error
22003	E_GE9D1D_DATAEX_EXPOVR
22003	E_GE9D21_DATAEX_FXPUNF
22003	E_GE9D22_DATAEX_EPUNF
22003	E_GE9D23_DATAEX_DECUNF
22003	E_GE9D24_DATAEX_OTHER
22005	E_GE9D0C_DATAEX_AGN
22007	E_GE9D0F_DATAEX_DATEOVR
22008	E_GE9D0E_DATAEX_DTINV
22009	E_GE9D0F_DATAEX_DATEOVR
22011	E_GE80E8_LOGICAL_ERROR
22012	E_GE9D1E_DATAEX_FPDIV
22012	E_GE9D1F_DATAEX_FLTDIV
22012	E_GE9D20_DATAEX_DCDIV
22012	E_GE9D24_DATAEX_OTHER
22015	E_GE9D0F_DATAEX_DATEOVR
22018	E_GE7918_SYNTAX_ERROR
22019	E_GE7918_SYNTAX_ERROR
22021	E_GE9D08_DATAEX_NOSUB
22022	E_GE9D11_DATAEX_INVIND
22023	E_GE9D08_DATAEX_NOSUB
22024	E_GE98BC_OTHER_ERROR
22025	E_GE7918_SYNTAX_ERROR
22026	E_GE9D08_DATAEX_NOSUB
22027	E_GE7918_SYNTAX_ERROR
22500	E_GE9D17_DATAEX_TYPEINV
23000	E_GE9D6C_CONSTR_VIO
24000	E_GE9DD0_CUR_STATE_INV
25000	E_GE9E34_TRAN_STATE_INV
26000	E_GE75B2_NOT_FOUND
27000	E_GE9EFC_TRIGGER_DATA

SQLSTATE	Generic Error
28000	E_GEA028_USER_ID_INV
2A000	E_GE7918_SYNTAX_ERROR
2A500	E_GE7594_TABLE_NOT_FOUND
2A501	E_GE759E_COLUMN_UNKNOWN
2A502	E_GE75F8_DEF_RESOURCE
2A503	E_GE84D0_NO_PRIVILEGE
2A504	E_GE75A8_CURSOR_UNKNOWN
2A505	E_GE75B2_NOT_FOUND
2A506	E_GE797C_INVALID_IDENT
2A507	E_GE797C_INVALID_IDENT
2B000	E_GE7D00_QUERY_ERROR
2C000	E_GE7918_SYNTAX_ERROR
2D000	E_GE9E34_TRAN_STATE_INV
2E000	E_GE797C_INVALID_IDENT
33000	E_GE75BC_UNKNOWN_OBJECT
34000	E_GE75A8_CURSOR_UNKNOWN
35000	E_GE7D00_QUERY_ERROR
37000	E_GE7918_SYNTAX_ERROR
37500	E_GE7594_TABLE_NOT_FOUND
37501	E_GE759E_COLUMN_UNKNOWN
37502	E_GE75F8_DEF_RESOURCE
37503	E_GE84D0_NO_PRIVILEGE
37504	E_GE75A8_CURSOR_UNKNOWN
37505	E_GE75B2_NOT_FOUND
37506	E_GE797C_INVALID_IDENT
37507	E_GE797C_INVALID_IDENT
3C000	E_GE9DD0_CUR_STATE_INV
3D000	E_GE98BC_OTHER_ERROR
3F000	E_GE797C_INVALID_IDENT
40000	E_GE98BC_OTHER_ERROR

SQLSTATE	Generic Error
40001	E_GEC2EC_SERIALIZATION
40002	E_GE9D6C_CONSTR_VIO
40003	E_GE9088_COMM_ERROR
42000	E_GE7918_SYNTAX_ERROR
42500	E_GE7594_TABLE_NOT_FOUND
42501	E_GE759E_COLUMN_UNKNOWN
42502	E_GE75F8_DEF_RESOURCE
42503	E_GE84D0_NO_PRIVILEGE
42504	E_GE75A8_CURSOR_UNKNOWN
42505	E_GE75B2_NOT_FOUND
42506	E_GE797C_INVALID_IDENT
42507	E_GE797C_INVALID_IDENT
44000	E_GE7D00_QUERY_ERROR
50000	E_GE98BC_OTHER_ERROR
50001	E_GE7602_INS_DUP_ROW
50002	E_GE8CA0_SYSTEM_LIMIT
50003	E_GE8D04_NO_RESOURCE
50004	E_GE8D68_CONFIG_ERROR
50005	E_GE9470_GATEWAY_ERROR
50006	E_GE9858_FATAL_ERROR
50007	E_GE9E98_INV_SQL_STMT_ID
50008	E_GEA0F0_SQL_STMT_INV
50009	E_GEA154_RAISE_ERROR
5000A	E_GE7D00_QUERY_ERROR
5000B	E_GE98BC_OTHER_ERROR
5000C	E_GE9D0D_DATAEX_FETCH0
5000D	E_GE9D12_DATAEX_CURSINV
5000E	E_GEA21C_DUP_SQL_STMT_ID
5000F	E_GE98BC_OTHER_ERROR
5000H	E_GE75BC_UNKNOWN_OBJECT

SQLSTATE	Generic Error
5000I	E_GE98BC_OTHER_ERROR
5000J	E_GE98BC_OTHER_ERROR
5000K	E_GE98BC_OTHER_ERROR
5000L	E_GE9088_COMM_ERROR
5000M	E_GE9088_COMM_ERROR
HZ000	E_GE9088_COMM_ERROR

Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y