Ingres® 2006 Release 3

SQL Reference Guide

INGRES®

October 2007

Contents

Chapter 4: Understanding the Elements of SQL Statements   91

Chapter 5: Working with Embedded SQL   161

Chapter 8: SQL Statements   289

Appendix A: Keywords   719

Appendix B: Terminal Monitors   747

Index 789

Chapter 1: Introducing the SQL Reference Guide

The *SQL Reference Guide* provides the following information:

- Detailed descriptions of all SQL statements

- Examples of the correct use of SQL statements and features

- Detailed discussion on performing transactions and handling errors

- Detailed descriptions about database procedures, sessions, and events

Audience

This guide is intended for programmers and users who have an understanding of the SQL language, and a basic understanding of Ingres® and relational database systems. In addition, you must have a basic understanding of your operating system. This guide is also intended as a reference for the database administrator.

Enterprise Access Compatibility

If your installation includes one or more Enterprise Access products, check your OpenSQL documentation for information about syntax that differs from the syntax described in this guide. Areas that differ include:

- Varchar data type length

- Legal row size

- Command usage

- Name length

- Table size

System-specific Text in This Guide

Generally, Ingres operates the same on all systems. When necessary, however, this guide provides information specific to your operating system. For example:

UNIX: Information is specific to the UNIX environment.

VMS: Information is specific to VMS environment.

Windows: Information is specific to the Windows environment.

When necessary for clarity, the symbol ▆ is used to indicate the end of system-specific text.

For sections that pertain to one system only, the system is indicated in the section title.

Terminology Used in This Guide

This guide uses the following terminology:

command

A *command* is an operation that you execute at the operating system level. An extended operation invoked by a command is often referred to as a utility.

statement

A *statement* is an operation that you embed within a program or execute interactively from a terminal monitor.

Note: A statement can be written in Ingres 4GL, a host programming language (such as C), or a database query language (SQL or QUEL).

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).

Chapter 2: Introducing SQL

This chapter provides an overview of SQL, including an introduction to interactive and embedded SQL statements. This chapter also describes the features and extensions of SQL and the database management system (DBMS).

SQL Functionality

SQL statements enable you to:

- **Manipulate database objects**—Create, modify, and destroy a variety of database objects, such as tables, views, indexes, and database procedures.

- **Manipulate data**—Select, insert, update, and delete data in database tables.

- **Manage groups of statements as transactions**—Process a group of database statements as a single transaction. Transaction management includes the ability to undo (roll back) a transaction, either in whole or in part.

- **Perform other database management functions**—Set runtime options, copy data between tables and files, modify the characteristics of a database table, and perform many other database management functions.

Types of SQL Statements

SQL statements are categorized according to the task performed:

Data Definition Language (DDL)

Creates or deletes objects such as tables, indexes, and database procedures.

Data Manipulation Language (DML)

Allows data manipulation in tables.

SQL Releases

SQL statements come in two releases:

Interactive SQL

SQL statements are entered from a terminal and query results display on the terminal screen.

Embedded SQL

SQL statements can be included in programming languages such as C or Fortran.

Interactive SQL

Interactive SQL statements are entered through the Terminal Monitor.

Line-Based Terminal Monitors

The line-based Terminal Monitor accepts SQL statements in a line-oriented style. The line-based Terminal Monitor is invoked by typing **sql** at the operating system prompt.

For a complete discussion of the line-based Terminal Monitor, see the appendix "Terminal Monitors."

The Help SQL statement displays information about SQL statements and about tables, views, and other database objects. A complete list of help options is provided in the chapter "Using SQL Statements."

Forms Based Terminal Monitor

The forms-based Terminal Monitor accepts SQL statements in a screen-oriented style. The forms based Terminal Monitor is invoked by typing **isql** at the operating system prompt.

Embedded SQL

Embedded SQL statements can be embedded in a procedural (3GL) programming language. The procedural language is referred to as the *host* language.

Embedded SQL Support

Embedded SQL is supported in the following host languages:

Windows:

- C
- C++
- COBOL
- Fortran

UNIX:

- C
- C++
- COBOL
- Fortran
- Verdix Ada

VMS:

- C
- C++
- BASIC
- COBOL
- Fortran
- Pascal
- Ada

How Embedded SQL Differs From Interactive SQL

Embedded SQL statements can be mixed with the full range of host language statements and provide your applications with full access to Ingres databases. The statements available in embedded SQL include those available in interactive SQL; embedded SQL, however, differs from interactive SQL in the following ways:

- **Use of Host Language Variables** - Embedded SQL allows host variables to be used in place of many syntactic elements.

- **Error and Status Handling** - In interactive SQL, error and status messages are sent directly to the terminal screen. Embedded SQL stores error and status information in a data structure called the SQL Communications Area (SQLCA).

- **Cursors** - To enable an application to process the result of a query one row at a time, embedded SQL provides cursor versions of the data manipulation statements Select, Update, and Delete. A database cursor points to the row currently being processed by the application.

- **Forms Statement** - Embedded SQL allows the creation of applications based on forms that have been created through Visual-Forms-Editor (VIFRED). Using forms statements, your application can:

  - Display VIFRED forms

  - Transfer data from the form to the database, and vice-versa

  - Respond to user actions (such as menu selections, control keys, and function keys)

  - Validate user entries

  - Display help screens

- **Dynamic Programming** - Embedded SQL allows you to create and execute statements dynamically, specifying portions of SQL statements in program variables at runtime.

  The dynamic programming feature of embedded SQL allows you to specify tables, columns, and queries at runtime. Dynamic programming allows generic applications to be written that can be used with any table. Details about dynamic programming, can be found in Dynamic Programming in the chapter "Embedded SQL."

- **Multiple Sessions** - An embedded SQL application can use multiple sessions to connect to different databases or to establish multiple connections to the same database.

- **Additional Database Access Statements** - Embedded SQL includes several statements not available in interactive SQL. For example, there are embedded statements that enable your application to connect to a particular database, and to manipulate cursors.

SQL Naming and Statement Rules

This section briefly describes the SQL naming and statement rules, as well as the additional features and extensions of SQL and the database management system (DBMS).

The following statements and features enable the control of:

- Access to information in the database - Enhancements to the Grant SQL statement allow you to specify which users can view, add, change, or delete data from a table. In addition, table access for groups of users and for individual applications can be controlled.

- Access to computing resources - The Grant SQL statement allows control of user consumption of computing resources. For example, the amount of I/O a user can perform can be limited and the approximate maximum number of rows that can be returned by a query issued by the user to prevent a runaway query.

- Referential integrity - There are two mechanisms for ensuring that the data in various tables maintain the relationships required for your business purposes: rules and referential constraints. Both allow relationships between specific columns of different tables to be specified, and to specify actions to be performed when a change to a table violates the relationships you require.

Object Naming Rules

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

- Names can contain only alphanumeric characters and must begin with an alphabetic character or an underscore (_). Database names must begin with an alphabetic character, and cannot begin with an underscore.

- Case significance (upper or lower) is determined by the settings for the database in which the object is created (Ingres or ANSI/ISO Entry SQL-92-compliant) and differs for delimited and non-delimited identifiers.

  For details about delimited identifiers, see Regular and Delimited Identifiers in this chapter.

- Names can contain (but cannot begin with) the following special characters: 0 through 9, #, @, and $. Names specified as delimited identifiers (in double quotes) can contain additional special characters.

  For details about delimited identifiers, see Regular and Delimited Identifiers in this chapter.

- Database objects (such as tables, columns, views, and database procedures) cannot begin with the letters, ii. This name is reserved for use by the DBMS Server.

- The maximum length of an object name is 32 characters. Database names must be unique to 24 characters (or the maximum file name length imposed by your operating system, if less than 24).

  The following are examples of objects managed by Ingres tools (such as VIFRED or Vision):

  - Forms
  - JoinDefs
  - QBFNames
  - Graphs
  - Reports

- Avoid assigning reserved words as object names. A list of reserved words can be found in the appendix "Keywords."

Regular and Delimited Identifiers

Identifiers in SQL statements specify names for the following objects:

- Authorization identifier (user, group, or role)
- Column
- Constraint
- Correlation name
- Cursor
- Database event
- Database procedure
- Database procedure label
- Database procedure parameter
- Database procedure variable
- Index
- Location
- Prepared query
- Rule
- Savepoint
- Schema
- Synonym
- Table
- View

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

- Table name in a Select SQL statement specified using a regular identifier:

  ```
  select * from employees
  ```

- Table name in a Select SQL statement specified using a delimited identifier:

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

Delimited identifiers enable you to embed special characters in object names. The use of special characters in regular identifiers is restricted.

Note: Case sensitivity for delimited identifiers is specified when a database is created. For compliance with ANSI/ISO Entry SQL-92, delimited identifiers must be case sensitive.

Restrictions on Identifiers

The following table lists the restrictions for regular and delimited identifiers (the names assigned to database objects):

Restrictions	Regular Identifiers	Delimited Identifiers
Quotes	Specific without quotes	Specified in double quotes
Keywords	Cannot be a keyword	Can be a keyword
Valid special characters	"At" sign (@) (not ANSI/ISO) Crosshatch (#) (not ANSI/ISO) Dollar sign ($) (not ANSI/ISO) Underscore (_)	Ampersand (&) Asterisk (*) "At" sign (@) Colon (:) Comma (,) Crosshatch (#) Dollar sign ($) 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 (|) Backslash (\) Caret (^) Curly braces ({ }) Exclamation point (!) Left quote (ASCII 96 or X'60') Tilde (~)

Note: The maximum length of an identifier is 32 characters. For ANSI/ISO Entry SQL-92 compliance, identifiers must be no longer than 18 characters.

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

```
DEL (ASCII 127 or X'7F')
```

To specify double quotes in a delimited identifier, repeat the quotes.

For example:

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

is interpreted as:

```
"Identifier"Name"
```

Trailing spaces are deleted from object names specified using delimited identifiers.

For example:

```
create table "space test " (scolumn int);
```

creates a table named, space test, with no trailing blanks (leading blanks are retained).

If an object name composed entirely of spaces is specified, the object is assigned a name consisting of a single blank. For example, the following creates a table named " ".

```
create table "      " (scolumn int);
```

Case Sensitivity of Identifiers

Case sensitivity for regular and delimited identifiers is specified at the time a database is created. By default, delimited identifiers are not case sensitive. For compliance with ANSI/ISO Entry SQL-92, however, delimited identifiers must be case sensitive.

The DBMS Server treats database, user, group, role, cursor, and location names without regard to case, and mixed-case database or location names cannot be created.

Comment Delimiters

To indicate comments in interactive SQL, 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 */
```

The delimiter, --, indicates that the rest of the line is a comment. A comment delimited by -- cannot be continued to another line.-- (left side only).

For example:

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

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

--, with the same usage rules as interactive SQL. Host language comment delimiters.

For information about comment delimiters, see the *Embedded SQL Companion Guide*.

Statement Terminators

Statement terminators separate one SQL statement from another. In interactive SQL, the statement terminator is the semicolon (;). Terminate statements with a semicolon when entering two or more SQL statements before issuing the **go** command (\g), selecting the Go menu item, or issuing some other terminal monitor command.

In the following example, semicolons terminate the first and second statements. The third statement does not need to 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 SQL 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 or to abbreviate long table names. 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 Rules

Correlation names can be specified in a Select, Delete, Update, Create Integrity, or Create Rule SQL statement. The rules of using correlation names are as follows:

- A single query can reference a maximum of 126 table names (including all base tables referenced by views specified in the query).

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

```
delete from employee
    where salary > 100000;
```

the DBMS Server assumes the correlation name of employee for the salary column and interprets the preceding query as:

```
delete from employee
    where employee.salary > 100000;
```

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

```
/*wrong*/
delete 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:

```
/*wrong*/
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, the DBMS Server resolves unqualified column names by checking the tables specified in the nearest from clause, then the from clause at the next highest 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, the DBMS Server performs the following steps to determine to which table the columns belong:

dno

> The DBMS Server checks the table specified in the nearest from clause (the deptsal table). The dno column does belong to the deptsal table; the DBMS interprets the column specification as deptsal.dno

dept

> The DBMS Server checks the table specified in the nearest from clause (deptsal). The dept column does not belong to the deptsal table.
>
> The DBMS Server checks the table specified in the from clause at the next highest level (the employee table). The dept column does belong to the employee table; the column specification is interpreted as employee.dept.

■ The DBMS Server 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);
```

The DBMS Server checks the description of the sales_departments table for the mgrno and manager columns; if they are not found, it checks the employee table next, but does not check the mktg_departments table. Similarly, the DBMS Server first checks the mktg_departments table for the mgrno and manager_id columns. If they are not found, it checks the employee table, but never checks the sales_departments table.

Database Procedures

Database procedures are compiled, stored, and managed by the DBMS Server. Database procedures can be used in conjunction with rules to enforce database integrities, or to perform frequently repeated operations. When the procedure is created, its execution plan is saved, reducing the execution overhead.

Database Procedure Creation

Database procedures can be created interactively or in an embedded program. A database procedure can be executed in a host language program, in terminal monitor, in another database procedure, or in a 4GL program. Database procedures can also be invoked by rules. For more information, see Database Procedures and Rules in the chapter "Understanding Database Procedures, Sessions, and Events."

Determine Settings for a Database

To determine the settings for the database to which a session is connected, use dbmsinfo(db_name_case) and dbmsinfo(db_delim_case).

For details about dbmsinfo, see Status Information (see page 233)

Object Management Extension

The Object Management Extension allows data types to be created in addition to the standard SQL data types. Using the Object Management Extension, you can define operators and a function to manipulate your data types, and integrate the new data types, operators, and functions into the DBMS Server.

ANSI Compliance

Ingres is compliant with ANSI/ISO Entry SQL-92. In addition, Ingres contains numerous vendor extensions. For embedded SQL applications, the ESQL preprocessor can be directed to flag statements in your program that are not compliant with entry-level ANSI/ISO SQL-92. For details, see the *Embedded SQL Companion Guide*.

Information about the settings required to operate in compliance with ANSI/ISO Entry SQL-92, can be found in the appendix "ANSI Compliance Settings."

OpenSQL

OpenSQL is the subset of SQL statements that can be used to access non-Ingres databases through Enterprise Access products.

Security Levels

Basic Ingres installations can be administered in compliance with the C2 security standard. The following statements are of particular interest to C2 security administrators and DBAs:

- Create/drop/help security_alarm

- Enable/disable security_audit

- Create/alter/drop user

- Create/alter/drop role

- Create/alter/drop group

- Create/alter/drop location

- Register/remove table

- Dbmsinfo(security_priv)

- Dbmsinfo(security_audit_log)

For details about administering a C2 site, see the *Database Administrator Guide.*

Chapter 3: Understanding SQL Data Types

This chapter describes SQL data types, along with data type storage formats, literals, and SQL constants.

SQL Data Types

The following table lists the SQL data types:

Class	Category	Data Type
Character	Fixed length	c
		char (character)
	Varying length	text
		varchar (character varying)
		long varchar (clob, character large object, char large object)
Unicode	Fixed length	nchar
	Varying length	nvarchar
		long nvarchar (clob, nclob, nchar large object, national character large object)
Numeric	Exact numeric	integer (integer4)
		smallint (integer2)
		bigint (integer8)
		tinyint (integer1)
		decimal
	Approximate numeric	float (float8, double precision)
		float4 (real)
Date/time	Date	date
		ansidate
		ingresdate

Class	Category	Data Type
	Time	time (time with time zone, time without time zone, time with local time zone)
	Timestamp	timestamp (timestamp with time zone, timestamp without time zone, timestamp with local time zone)
	Interval	interval (interval year to month, interval day to second)
Abstract	(none)	money
	Logical key	object_key
		table_key
Binary		byte
		byte varying
		long byte (blob, binary large object)

Character Data Types

Character data types are strings of characters. Upper and lower case alphabetic characters are accepted literally. There are two fixed-length character data types: char and c, and three variable-length character data types: varchar, long varchar, and text.

The maximum row length is dependent on the default_page_size setting (a DBMS Server configuration parameter), and can be set to a maximum of 32,767 bytes. For further information on 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 cannot exceed 32,000. Long varchar columns are an exception: the maximum length of these columns is 2 GB.

C Data Types

Fixed-length c data types accept only printing characters. Non-printing characters, such as control characters, are converted into blanks.

Blanks are ignored when c strings are compared. For example, this c string:

```
'the house is around the corner'
```

is considered equal to:

```
'thehouseisaroundthecorner'
```

Note: Char is the preferred fixed length character type. C is supported for backward compatibility.

Char Data Types

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

```
'ABC  '
```

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

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

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

```
'ABC '
```

the wildcard specification must also contain trailing blanks:

```
'____'
```

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

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

Note: Character is a synonym for char.

Text Data Types

All ASCII characters except the null character (\0) are allowed within text strings. Null characters are converted to blanks.

Blanks are not ignored when text strings are compared. Unlike varchar, if the strings are unequal in length, blanks are not added to the shorter string. For example, assume that the following text strings are being compared:

`'abcd'`

and

`'abcd '`

The string 'abcd ' is considered greater than the string 'abcd' because it is longer.

Note: Varchar is the preferred varying length character type. Text is supported for backward compatibility.

Varchar Data Types

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

```
'03ABCxx'
```

where:

03 is a 2-byte length specifier

ABC is three bytes of data

xx represents two bytes containing unknown (and irrelevant) data.

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

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

```
'03ABC'
```

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, the following two varchar 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 varchar 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 Varchar Data Types

The long varchar data type has the same characteristics as the varchar data type, but can accommodate strings up to 2 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 more information on data handlers, see Data Handlers for Large Objects (see page 207) 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 SQL statement.

- They can be included in a select list with the "distinct" qualifier, but duplicate values will not be eliminated.

- 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

  - Byteextract

  - Left

  - Right

  - Substring

  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 nchar, nvarchar and long nvarchar are used to store Unicode data. They behave similar to char, varchar and long varchar character types respectively, except that each character in Unicode types typically uses 16 bits. Similar to their local character counterparts, nchar types are of fixed length and nvarchar and long nvarchar are of variable length.

Ingres represents Unicode data in UTF-16 encoding form and internally stores them in Normalization Form D (NFD) or Normalization Form C (NFC) depending upon the createdb flag (-n or –i) used for creating the database. Each character of a Unicode value is typically stored in a 2-byte code point (some complex characters require more). The maximum length of a Unicode column is limited by the maximum row width configured, but cannot exceed 16,000 characters for nchar and 15,999 for nvarchar. Long nvarchar columns can have a maximum length of 2 GB.

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.).

Embedded programs use wchar_t data type to store and process Unicode values.

Note: No matter what size the compilation platform uses for the data type wchar_t, Ingres initializes only the low 16 bits with UTF-16 data. When Ingres reads values from wchar_t variables, the values are coerced to 16 bits and stored in the NFD or NFC canonical form. Applications that make use of any available bits beyond the lower 16 to represent information, for example for UTF-32, will not be able to store that information directly in Ingres. It is the responsibility of the application to convert UTF-32 encoded Unicode to UTF-16 encoded Unicode for use with the Ingres Unicode data types.

For details on Unicode Normalization Forms, go to http://www.unicode.org.

Numeric Data Types

There are two categories of numeric data types: exact and approximate. Exact data types include integer data types and decimal data types. Approximate data types include floating point data types.

Integer Data Types

Exact numeric data types includes the following 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 Types

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 is an example of precision and scale in decimal values:

```
Precision=10 ┐
          ┌───────────┐
          │12345.67890│
          └───────────┘
Scale=5 ──┘
```
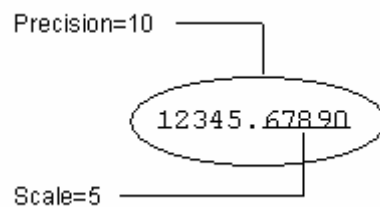
The minimum precision for a decimal value is 1 and the maximum precision is 39. The scale of a decimal value cannot exceed its precision. Scale can be 0 (no digits to the right of the decimal point).

Note: The decimal data type is suitable for storing currency data where the required range of values or number of digits to the right of the decimal point exceeds the capacities of the money data type. Note that, for display purposes, a currency sign cannot be specified for decimal values.

Decimal Data Type Syntax

Specify the decimal data type using the following syntax:

```
decimal(p,s)
```

p

Defines the precision.

s

Defines the scale.

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

Floating Point Data Types

A floating point value is represented either as whole plus fractional digits (like decimal values) or as a mantissa plus an exponent. The following is an example of the mantissa and exponent parts of floating point values:
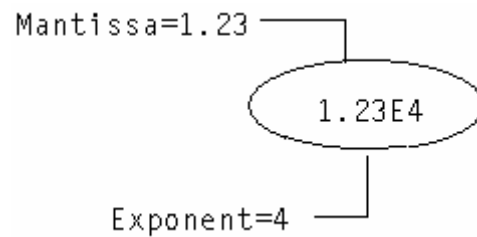
There are two floating point data types:

- float4 (4-byte)
- float (8-byte)

Note: Real is a synonym for float4, while float8 and double precision are synonyms for float.

Floating point numbers are stored in four or eight bytes. Internally, Ingres rounds eight byte numbers to 15 decimal digits. The precision of four byte numbers is processor dependent.

You can specify the binary 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. Storage is allocated according to the precision that is specified, as follows:

Range of Binary Precision	Storage Allocated
0 to 23	4-byte float
24 to 53	8-byte float

Float Point Limitations

Users must consider the effects of data type conversions when numeric values are combined or compared. This is especially true when dealing with floating point values.

Exact matches on floating point numbers are discouraged, because float and float4 data types are approximate numeric values. Integer and decimal data types, on the other hand, are exact numeric values.

Date/Time Data Types

Date/time data types include the following:

- Date data types
- Time data types
- Timestamp data types
- Interval data types

Date/Time Input Formats

Date/time data values have the following data type input formats:

- Ansidate
- Ingresdate
- Time (without time zone)
- Time with time zone
- Time with local time zone
- Timestamp (without time zone)
- Timestamp with time zone
- Timestamp with local time zone
- Interval year to month
- Interval day to second

Date Data Type

The declaration format of the date type can be one of the following:

```
DATE
```

```
ANSIDATE
```

```
INGRESDATE
```

Examples:

Date Format	Example
DATE	The keyword DATE used for a column data type is an alias, which can be configured to either ANSIDATE or INGRESDATE (the default) by setting the configuration parameter date_alias.
ANSIDATE	2006-05-16
INGRESDATE	For more information, see Ingresdate Data Types (see page 65).

Time Data Types

The TIME data types include the following:

- Time or time without time zone
- Time with time zone
- Time with local time zone

The format is as follows:

```
TIME [time_precision] [time_zone_spec]
```

time_precision

(Optional) Indicates the number of digits of precision in the fractions of seconds, as an integer value from 0 to 9. When no time precision is supplied, the value of time_precision is set to 0 by default.

time_zone_spec

(Optional) Specifies a time zone as one of the following:

- WITH TIME ZONE
- WITHOUT TIME ZONE
- WITH LOCAL TIME ZONE

Example:

Time Format	Example
TIME(5) WITH TIME ZONE	12:30:55.12345-05:00
TIME(4) WITHOUT TIME ZONE	12:30:55.1234
TIME(9) WITH LOCAL TIME ZONE	12:30:55.123456789

Note: When a value is entered that exceeds the specified precision, the value is truncated at the specified precision.

Note: The time value in TIME WITH TIME ZONE data type indicates the time at the specified time zone.

Timestamp Data Types

The TIMESTAMP data type consists of a date and time.

The TIMESTAMP data types include the following:

- Timestamp or timestamp without time zone
- Timestamp with time zone
- Timestamp with local time zone

The format is as follows:

```
timestamp [(timestamp_precision)] [time_zone_spec]
```

timestamp_precision

(Optional) Indicates the number of digits of precision in the fractions of seconds, as an integer value from 0 to 9. When no time precision is supplied, the value of time_precision is set to 6 by default.

time_zone_spec

(Optional) Specifies a time zone as one of the following:

- WITH TIME ZONE
- WITHOUT TIME ZONE
- WITH LOCAL TIME ZONE

Example:

Timestamp Format	Example
TIMESTAMP(5) WITH TIME ZONE	2006-12-15 9:30:55.12345-08:00
TIMESTAMP(4) WITHOUT TIME ZONE	2006-12-15 12:30:55.1234
TIMESTAMP(9) WITH LOCAL TIME ZONE	2006-12-15 12:30:55.123456789

Note: Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the database. For example:

TIMESTAMP '2006-12-15 9:30:55 -8:00'

is the same as

TIMESTAMP '2006-12-15 12:30:55 -5:00'

Note: When a value is entered that exceeds the specified precision, the value is truncated at the specified precision.

Note: The time value in TIMESTAMP WITH TIME ZONE data type indicates the time at the specified time zone.

Interval Data Types

The INTERVAL data types include the following:

- Interval year to month
- Interval day to second

The format is as follows:

```
interval interval_qualifier
```

interval_qualifier

Defines an interval column as one of the following:

- YEAR TO MONTH
- DAY TO SECOND [(*second precision*)]

where *second precision* is the number of digits in the fractions of seconds field.

Example:

Interval Format	Example	Explanation
INTERVAL DAY TO SECOND(3)	7 6:54:32.123	An interval of 7 days, 6 hours, 54 minutes, 32 seconds and 123 thousandths of a second
INTERVAL YEAR TO MONTH	123-04	An interval of 123 years, 4 months.

Note: When a value is entered that exceeds the specified precision, the value is truncated at the specified precision.

Summary of ANSI Date/Time Data Types

The following table summarizes valid input and output formats for ANSI date/time data types:

Data Type	Input and Output Format	Example
ANSIDATE	yyyy-mm-dd	2007-02-28
TIME WITH TIME ZONE	hh:mm:ss.ffff... [+|-]th:tm	12:45:12.23456 - 05:00
TIME or TIME WITHOUT TIME ZONE	hh:mm:ss.ffff...	12:45:12.23456
TIME WITH LOCAL TIME ZONE	hh:mm:ss.ffff...	12:45:12.23456
TIMESTAMP WITH TIME ZONE	yyyy-mm-dd hh:mm:ss.ffff... [+|-]th:tm	12:45:12.23456 - 05:00
TIMESTAMP or TIMESTAMP WITHOUT TIME ZONE	yyyy-mm-dd hh:mm:ss.ffff... [+|-]th:tm	12:45:12.23456 - 05:00
TIMESTAMP WITH LOCAL TIME ZONE	yyyy-mm-dd hh:mm:ss.ffff... [+|-]th:tm	12:45:12.23456 - 05:00
INTERVAL YEAR TO MONTH	[+|-]Years-MM	55-4
INTERVAL DAY TO SECOND	[+|-] Days HH:MM:SS.ffff...	-18 12:02:23.12345

where:

yyyy

Is a four-digit year value. All four digits are required.

mm

Is a two-digit month value between 01 to 12.

dd

Is a two-digit day value between 01 to 31.

hh

Is a two-digit hour value between 00 to 23.

mm

Is a two-digit minute value between 00 to 59.

th

Is a two-digit hour value between -12 to +14.

tm

Is a two-digit minute value between 00 to 59.

Years

Is the number of years.

Days

Is the number of days.

SS.fffff…

Is a two-digit seconds value, which can be followed by 0 to 9 digits of fractional seconds.

Ingresdate Data Types

The ingresdate data type is an abstract data type. Ingresdate values can contain either absolute dates and times or time intervals. The ingresdate data type input formats are as follows:

- Absolute date input

- Absolute time input

- Combined date and time input

- Date interval

- Time interval

Absolute Date Input

Ingresdate values are specified as quoted character strings. A date can be entered by itself or together with a time value. If a date is entered without specifying the time, no time is shown when the data displays (see page 71).

II_DATE_FORMAT for Ingresdate

The legal formats for absolute ingresdate values are determined by the II_DATE_FORMAT setting, summarized in the following table. If it 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.*

The following table lists date input formats:

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* and all US formats except *mm/dd/yyyy*	*dd/mm/yy*
MULTINATIONAL4	*dd/mm/yyyy* and all US formats	*dd/mm/yyyy*
ISO	*yymmdd* *ymmdd* *yyyymmdd* *mmdd* *mdd* and all US input formats except *mmddyy*	*yymmdd*
ISO4	*yymmdd* *ymmdd* *yyyymmdd* *mmdd* *mdd* and all US input formats except *mmddyy*	*yyyymmdd*
SWEDEN or FINLAND	*yyyy-mm-dd* all US input formats except *mm-dd-yyyy*	*yyyy-mm-dd*

II_DATE_FORMAT Setting	Valid Input Formats	Output
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* *yymmd*d *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 this date is entered:

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

using the format *mm/dd/yyyy*, the DBMS Server assumes that you are referring to March 21, 2003.

In three-character month formats, for example, *dd-mmm-yy*, specify three-letter abbreviations for the month (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 date and time, use the constant, now.

II_DATE_CENTURY_BOUNDARY

The II_DATE_CENTURY_BOUNDARY variable, which can be set to an integer in the 0< n <=100 range, dictates the implied century for an ingresdate value when only the last two digits of the year are entered.

For example, if II_DATE_CENTURY_BOUNDARY is 50 and the current year is 1999, an input date of 3/17/51 is treated as March 17, 1951, but a date of 03/17/49 is treated as March 17, 2049.

If the II_DATE_CENTURY_BOUNDARY variable is not set or if it is set to 0 or 100, the current century is used. If the user enters the full four digits for the year in a four-digit year field in the application, the year is accepted as entered, regardless of the II_DATE_CENTURY_BOUNDARY setting.

Absolute Time Input

The legal format for inputting an absolute time into an ingresdate value is:

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

Input formats for absolute times are assumed to be on a 24-hour clock. If a time with the designation am or pm is entered, the time is converted to a 24-hour internal and displayed representation.

If *timezone* is omitted, the local time zone designation is assumed. Times are stored as Greenwich Mean Time (GMT) and displayed using the time zone adjustment specified by II_TIMEZONE_NAME.

If an absolute time without a date is entered, the date defaults to 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 in an ingresdate. The following table shows examples of valid date and time entries, using the US absolute date input formats:

Format	Example
mm/dd/yy hh:mm:ss	11/15/03 10:30:00
dd-mmm-yy hh:mm:ss	15-nov-03 10:30:00
mm/dd/yy hh:mm:ss	11/15/03 10:30:00
dd-mmm-yy hh:mm:ss gmt	15-nov-03 10:30:00 gmt
dd-mmm-yy hh:mm:ss [am|pm]	15-nov-03 10:30:00 am
mm/dd/yy hh:mm	11/15/03 10:30
dd-mmm-yy hh:mm	15-nov-03 10:30
mm/dd/yy hh:mm	11/15/03 10:30
dd-mmm-yy hh:mm	15-nov-03 10:30

Date Interval

Ingresdate interval values, like absolute date values, are entered as quoted character strings. Date intervals can be specified in terms of years, months, days, or combinations of these. Years and months can be abbreviated to yrs and mos, respectively. For example:

```
'5 years'
'8 months'
'14 days'
'5 yrs 8 mos 14 days'
'5 years 8 months'
'5 years 14 days'
'8 months 14 days'
```

The following table lists valid ranges for date intervals:

Date Interval	Range
Years	-9999 to +9999
Months	-119988 to +119988
Days	-3652047 to +3652047

Time Interval

Time intervals can be specified as hours, minutes, seconds, or combinations of these units. They can be abbreviated to hrs, mins, or secs. For example:

```
'23 hours'
'38 minutes'
'53 seconds'
'23 hrs 38 mins 53 secs'
'23 hrs 53 seconds'
'28 hrs 38 mins'
'38 mins 53 secs'
'23:38 hours'
'23:38:53 hours'
```

All values in an interval must be in the range -2,147,483,639 to +2,147,483,639. If a time interval greater than 1 day is entered, the interval is converted to a date and time interval.

For example:

```
'26 hours'
```

is converted to:

```
'1 day 2 hours'
```

Date and Time Display

Ingresdate values display as strings of 25 characters with trailing blanks inserted.

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

```
hh:mm:ss
```

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

If seconds are not entered when entering a time, zeros display in the seconds place.

For a time interval, Ingres displays the most significant portions of the interval that fit in the 25-character string. If necessary, trailing blanks are appended to fill out the string. The format appears as:

yy yrs *mm* mos *dd* days *hh* hrs *mm* mins *ss* secs

Significance is a function of the size of any component of the time interval. For instance, if the following time interval is entered:

```
5 yrs 4 mos 3 days 12 hrs 32 min 14 secs
```

the entry is displayed as:

```
5 yrs 4 mos 3 days 12 hrs
```

Truncating the least significant portion of the time (the minutes and seconds) to fit the result into 25 characters.

Coercion Between Date/Time Data Types

The rules governing coercion between the various date/time data types are as follows:

1. Ansidate cannot be converted to any of the time types, nor can the time types be converted to ansidate. Doing so results in error E_AD5066_DATE_COERCION.

2. When converting from a data type that does not have time zone information to a data type with time zone value (for example, ansidate to a timestamp with time zone, time without time zone to a time with time zone), the time zone is set to the current session time zone.

   Example: In Eastern Standard Time (EST) time zone (that is, -05:00), the following statements insert a value of 2007-02-08 16:41:00-05:00 in the database.

   ```
   create table tab (col1 timestamp with time zone);
   insert into tab values (TIMESTAMP  '2007-02-08 16:41:00');
   ```

3. Ingresdate, time with local time zone, and timestamp with local time zone, store date/time values in UTC. When converting from other data types like ansidate, time with/without time zone, timestamp with/without time zone to these data types, the session time zone displacement is subtracted from the date/time value. On the reverse operation, when converting from ansidate, time with/without time zone, timestamp with/without time zone to ingresdate, time with local time zone and timestamp with local time zone, the session time zone displacement is added to the date/time value in the database to make it in local time zone.

   Example: In EST time zone (with time zone displacement of -05 :00), the following query stores a value of 2007-02-18 15:04:12 in the database:

   ```
   create table tab (col1 timestamp with local time zone);
   insert into tab values (TIMESTAMP '2007-02-18 10:04:12');
   ```

   If this value was selected in PST time zone (with time zone displacement of -08:00), the session time zone value is added to the value stored in database and the value in local time zone is displayed, that is:

   ```
   2007-02-18 07:04:12
   ```

4. When a time value is converted to a timestamp, date, or time/timestamp with local time zone types, the year, month, and day fields are filled with the current year, month, and day value.

   Example: If current date is 08 Feb 2007 then the following statements insert a value of 2007-02-08 17:01:00 in the database:

   ```
   create table tab (col1 timestamp);
   insert into tab values (TIME '17:01:00');
   ```

5. When converting from time without time zone to time with local time zone, the following procedure is used:

   a. Current date is added to the time value to make a timestamp.

   b. Time zone displacement is then applied to the time value.

   c. The date part is removed from the result.

   Example: If current date is 08 Feb 2007 and the session time zone is -05 :00 (EST), the following query stores a value of 22:01:00 in the database:

   ```
   create table tab (col1 time with local time zone);
   insert into tab values (TIME '17:01:00');
   ```

6. INTERVAL types cannot be converted to any other types except themselves and ingresdates.

7. When a time/timestamp with time zone is converted to ingresdate, time with local time zone, or timestamp with local time zone, the time value is converted to UTC by applying the time zone information in the value.

   Example: In any time zone, the following query will insert a value of 2007-02-18 03:04:12 in the database:

   ```
   create table tab (col1 timestamp with local time zone);
   insert into tab values (TIMESTAMP '2007-02-18 10:04:12-07:00');
   ```

Abstract Data Types

Abstract data types include the following:

- Money data types
- Logical key data types

Money Data Types

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 parameters), that the comma is followed by a space. For example:

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

Logical Key Data Types

The logical key data type allows the DBMS Server or your application to assign a unique key value to each row in a table. Logical keys are useful when an application requires a table to have a unique key, and the columns of the table do not comprise a unique key.

Types of Logical Keys

There are two types of logical keys:

- **System_maintained**—The DBMS Server automatically assigns a unique value to the column when a row is appended to the table. Users or applications cannot change system_maintained columns. When the column is created, the DBMS Server assigns it the option, not null with default. An error is returned if any option other than not null with default is specified.

- **Not system_maintained**—The DBMS Server does not assign a value to the column when a row is appended. Your application must maintain the contents of the column; users and application programs can change logical key columns that are not system_maintained. The default for logical key columns is not system_maintained.

Specify the scope of uniqueness for system_maintained logical key columns using the following options:

- **Table_key**—Values are unique within the table.
- **Object_key**—Values are unique within the entire database.

If two or more system_maintained logical key columns of the same type (object_key or table_key) are created within the same table, the same value is assigned to all columns of the same type in a given row. Different values are assigned to object and table key columns in the same row, as shown in the following diagram:
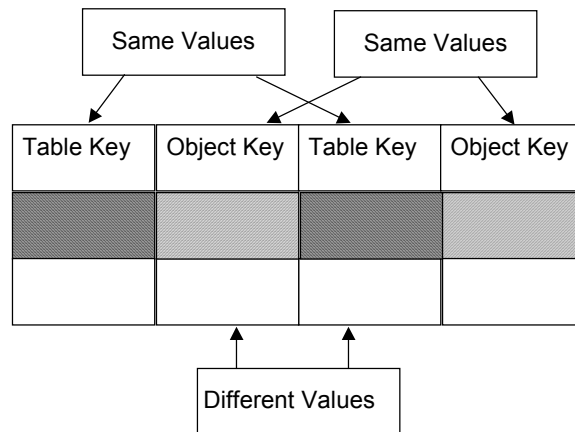
Table_key values are returned to embedded SQL programs as 8-byte char strings, and object_key values as 16-byte char strings. Values can be assigned to logical keys that are not system_maintained using string literals. For example:

```
insert into keytable(table_key_column)
       values('12345678');
```

Values assigned to table_keys must be 8-character strings; values assigned to object_keys must be 16-character strings.

Restrictions on Logical Keys

When working with logical keys, be aware of the following restrictions:

- A system_maintained logical key column cannot be created using the CREATE TABLE...AS SELECT statement. A not system_maintained data type is assigned to the resulting column.

- The COPY statement cannot be used to load values from a file into a system_maintained column.

Binary Data Types

There are three binary data types:

- Byte
- Byte varying
- Long byte

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

Byte Data Types

The byte data type is a fixed length binary data type. If the length of the data assigned to a byte column is less than the declared length of the column, the value is padded with zeros to the declared length when it is stored in a table. The minimum length of a byte column is 1 byte, and the maximum length is limited by the maximum row width configured but not exceeding 32,000.

Byte Varying Data Types

The byte varying data type is a variable length data type. The actual length of the binary data is stored with the binary data, and, unlike the byte data type, the data is not padded to its declared length. The minimum length of a byte varying column is 1 byte, and the maximum length is limited by the maximum row width configured but not exceeding 32,000.

Long Byte Data Types

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 (see page 207) and the *Embedded SQL Companion Guide*.

Restrictions on Long Byte Data Types

The following restrictions apply to long byte 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 of 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 byte columns:

  – Locate

  – Pad

  – Shift

  – Squeeze

  – Trim

  – Notrim

  – Charextract

  – Byteextract

  – Left

  – Right

  – Substring

- Long byte columns cannot be directly compared to other data types. To compare a long byte column to another data type, apply a coercion function.

- A literal of more than 2000 bytes cannot be assigned to a long byte column. For details about assigning long values to long byte columns, see the description of data handlers in the *Embedded SQL Companion Guide*, Dynamic Programming in the chapter "Embedded SQL" or the *OpenAPI User Guide*.

Storage Formats of Data Types

The following table lists storage formats for SQL data types:

Data Type	Description	Range
char	character	A string of 1 to maximum configured row size but not exceeding 32,000 (16,000 in a UTF8 instance) characters
c	character	A string of 1 to maximum configured row size but not exceeding 32,000 (16,000 in a UTF8 instance) characters
varchar	character	A string of 1 to maximum configured row size but not exceeding 32,000 (16,000 in a UTF8 instance) characters
long varchar	character	A string of 1 to 2 GB characters
text	character	A string of 1 to maximum configured row size but not exceeding 32,000 (16,000 in a UTF8 instance) 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 bytes in length)
tinyint	1-byte integer	-128 to +127
smallint	2-byte integer	-32,768 to +32,767
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	fixed-point exact numeric	Depends on precision and scale. Default is (5,0): -99999 to +99999. Maximum number of digits is 39.
float4	4-byte floating	-1.0e+38 to +1.0e+38 (7 digit precision)
float	8-byte floating	-1.0e+38 to +1.0e+38
ansidate	4-byte binary	0001-01-01 to 9999-12-31

Data Type	Description	Range
time	10-byte binary	00:00:00 to 24:00:00
timestamp	14-byte binary	0001-01-01 00:00:00 to 9999-12-31 24:00:00
interval year to month	3-byte binary	-9999-0 to 9999-11
interval day to second	12-byte binary	-3652047 00:00:00 to 3652047 23:59:59
ingresdate	ingresdate (12 bytes)	1-jan-0001 to 30-dec-9999 (for absolute dates) and -9999 years to +9999 years (for time intervals)
money	money (8 bytes)	$-999,999,999,999.99 to $999,999,999,999.99
table_key	logical key	No range: stored as 8 bytes
object_key	logical key	No range: stored as 16 bytes
byte	binary	Fixed length binary data, 1 to maximum configured row size
byte varying	binary	Variable length binary data, 1 to maximum configured row size
long byte	binary	1 to 2 GB of binary data

Nullable columns require one additional byte to store a null indicator.

Note: If your hardware supports the IEEE standard for floating point numbers, the float type is accurate to 14 decimal precision (-ddddddddddddd.dd to +ddddddddddddd.dd) and ranges from -10**308 to +10**308. The money type is accurate to 14 decimal precision with or without IEEE support.

Literals

A literal is an explicit representation of a value. There are three types of literals:

- String
- Numeric
- Date/time

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 money or date data type 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 causes unexpected results if the non-character data type contains the 'NULL (0) value.

For example, to compare the function X string literal that returns a varchar data type to a byte data type, cast the result of the X function as follows:

```
SELECT * FROM uid_table
       WHERE uid = BYTE(X'010000000000000000000000000000000')
```

or

```
SELECT * FROM uid_table
       WHERE HEX(uid) = '010000000000000000000000000000000'
```

Hexadecimal Representation

To specify a non-printing character in terminal monitor, use a hex (hexadecimal) constant. Hex constants are specified by an X followed by a single-quoted string composed of (an even number of) alphanumeric characters. For example, the following represents the ASCII string ABC<carriage return>:

```
X'4142430D'
```

```
A = X'41', B = X'42', C = X'43', and carriage return = X'0D'.
```

Quotes within Strings

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

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

which is evaluated as:

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

Unicode Literals

To specify a Unicode literal value within a non-Unicode command string (for example, in a query entered into the terminal monitor), the Unicode literal notation can be used. A Unicode literal is a sequence of ASCII characters intermixed with escaped sequence of hex digits, all enclosed in quotes and preceded by U&. The escape character \ precedes sets of 4 hex digits that are treated as 2-byte Unicode codepoints and the escape sequence \+ precedes sequences of 6 hex digits that are treated as 4-byte Unicode codepoints with a leading byte of 0. For example:

```
U&'Hello\1234world\+123456'
```

In this string, Hello is converted to the equivalent Unicode codepoints, the hex digits 1234 are treated as a 2-byte Unicode codepoint, **world** is also converted to the equivalent Unicode codepoints, and the hex digits 123456 are treated as a 4-byte Unicode codepoint with a leading byte of 0. The resulting literal is the concatenation of the converted Unicode components.

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. The literal is automatically converted to the appropriate data type, if necessary.

By default, the period (.) is displayed to indicate the decimal point. This default can be changed by setting II_DECIMAL. For information about setting II_DECIMAL, see the *System Administrator Guide*.

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:
```

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, it is represented as an integer. Values that exceed the range of integers are represented as decimals.

You can specify integers 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 exponential notation as follows:

```
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.

Date/Time Literals

Date/time literals specify ANSI compliant date/time values. There are four types of date/time literals:

- date
- time
- timestamp
- interval

Date/time literals can be assigned to the corresponding date/time data type without using an explicit conversion function. The value is coded as a quoted string, but is automatically converted to the appropriate internal value.

Date Literals

Literals of the ANSI date type have the following format:

```
DATE 'date_value'
```

date_value

Defines a date in the format yyyy-mm-dd.

Note: The II_DATE_FORMAT setting has no impact on the processing of date literals.

Examples:

```
date '2006-05-29'
date '1998-10-08'
date '2000-11-29'
```

Time Literals

Literals of the ANSI time type have the following format:

```
TIME 'time_value'
```

time_value

Defines a time value, which must be in the format hh:mm:ss, optionally followed by .fff (fractions of seconds) and also optionally followed by ±hh:mm, the time zone offset.

Examples:

```
time '11:11:00'
time '18:05:23.425364'
time '5:23:00-5:00'
```

Timestamp Literals

Literals of the ANSI timestamp type have the following format:

```
TIMESTAMP 'timestamp_value'
```

timestamp_value

Consists of a date value and a time value separated by a single space. The time values can contain optional fractions of seconds and/or time zone offsets.

Examples:

```
timestamp '2006-05-29 10:30:00.000-04:00'
timestamp '1918-11-11 11:11:00'
```

Interval Literals

Literals of the ANSI interval type have the following format:

```
INTERVAL [sign] 'interval_value' interval_qualifier
```

sign

Indicates a positive (+) or negative (-).

interval_value

Consists of a year to month interval value or a day to second interval value.

Format of year to month interval value: year-mm (for example: '25-7')

Format of day to second interval value: dddd... hh:mm:ss[.fffffffff] (for example: '15 5:10:27.4325')

interval_qualifier

Qualifies the interval as one of the following:

- YEAR TO MONTH
- DAY TO SECOND

Interval qualifier has the following format:

```
leading field [TO trailing field]
```

Valid values for leading field and trailing field in order of precedence are:

YEAR

MONTH

DAY

HOUR

MINUTE

SECOND [(p)].

The leading field cannot have lower precedence than the trailing field.

The precision value on the SECOND field indicates the number of digits allowed in the fractional part of the seconds field.

Examples:
```
interval'5-7' year to month
interval '-0-11' year to month
interval'+24 12:10:5.1234' day to second
interval '124' year
interval '12' month
interval '18' day
interval '10' hour
interval '34' minute
```

```
interval '20.23456789' second (9)
interval '8-11'  year to month
interval '12 10' day to hour
interval '12 10:20' day to minute
interval '121 10:15:23.123456' day to second(6)
```

SQL Constants

The following constants can be used in queries:

Special Constant	Meaning
Now	Current date and time. This constant must be specified in quotes.
	Note: This constant only works when used within the SQL date() function.
	Current date (as ANSI date type)
Null	Indicates a missing or unknown value in a table.
Today	Current date. This constant must be specified in quotes.
	Note: This constant only works when used within the SQL date() function.
current_date	Current date (as ANSI date type)
current_time	Current time
current_timestamp	Current date and time
local_time	Current time without time zone
local_timestamp	Current date and time without time zone
User	Effective user of the session (the Ingres user identifier, not the operating system user identifier).
current_user	Same as user.
system_user	Operating system user identifier of the user who started the session.
initial_user	Ingres user identifier in effect at the start of the session.
session_user	Same as user.

These constants can be used in queries and expressions. They can be used any number of times in a query, but the value is only materialized once per query execution. So a constant such as CURRENT_TIME can be referenced in an INSERT statement that inserts many rows, or an UPDATE statement that alters many rows, and the same time value will be used for each.

For example:

```
select date('now');

insert into sales_order
        (item_number, clerk, billing_date)
        values ('123', user, date('today')+date('7 days'));

update employee set e_sal = e_sal * 1.05, e_udp_time = CURRENT_TIMESTAMP where
e_stat = 'x';
```

To specify the effective user at the start of a session, use the Ingres -u flag (for operating system commands) or the identified by clause of the SQL connect statement.

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 section Create Table in the chapter "Using SQL Statements."

The ifnull function and the is null predicate allow nulls in queries to be handled. For details, see ifNull function and isNull in the chapter "Elements of SQL Statements."

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.

Nulls and Aggregate Functions

If an aggregate function against a column that contains nulls is executed, the function ignores the nulls. This prevents unknown or inapplicable values from affecting the result of the aggregate. For example, if the aggregate function, avg(), is applied to a column that holds the ages of your employees, be sure that any ages that have not been entered in the table are not treated as zeros by the function. This distorts the true average age. If a null is assigned to any missing ages, the aggregate returns a correct result: the average of all known employee ages.

Aggregate functions, except count(), return null for an aggregate that has an argument that evaluates to an empty set. (Count() returns 0 for an empty set.) In the following example, the select returns null, because there are no rows in the table named test.

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

In the above example, use the ifnull function to return a zero (0) instead of a null:

```
select ifnull(max(coll),0) as x from test;
```

For more information, see ifNull function in the chapter "Elements of SQL Statements."

When specifying a column that contains nulls as a grouping column (that is, in the group by clause) for an aggregate function, nulls in the column are treated 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 chapter "Using SQL Statements".

Nulls and Integrity Constraints

When creating a table with nullable columns and subsequently creating integrities on those columns (using the CREATE INTEGRITY statement), the constraint must include the OR...IS NULL clause to ensure that nulls are allowed in that column.

For example, if the following create table statement is issued:

```
create table test (a int, b int not null);
/* "a" is nullable */
```

and the following integrity constraint is defined on the test table:

```
create integrity on test is a > 10;
```

the comparison, a >10, is not true whenever a is null. For this reason, the table does not allow nulls in column a, even though the column is defined as a nullable column.

Similarly, the following INSERT statements fails:

```
insert into test (b) values (5);
insert into test values (null, 5);
```

Both of these INSERT statements are acceptable if the integrity had not been defined on column a. To allow nulls in column a, define the integrity as:

```
create integrity on test is a > 10 or a is null;
```

Note: If an integrity on a nullable column is created without specifying the OR...IS NULL clause and the column contains nulls, the DBMS Server issues an error and the integrity is not created.

Chapter 4: Understanding the Elements of SQL Statements

This chapter describes the following elements of SQL statements:

- Operators, functions and predicates

- Arithmetic operations, assignments, and other basic operations

- Expressions and search conditions in queries

This chapter identifies the differences in syntax between embedded and interactive SQL (where applicable). If the embedded syntax is dependent on a host language, see the *Embedded SQL Companion Guide*.

SQL Operators

There are three types of SQL operators:

- Arithmetic

- Comparison

- Logical

Arithmetic Operators

Arithmetic operators are used to combine numeric expressions arithmetically to form other numeric expressions.

The following are the valid arithmetic operators (in descending order of precedence):

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

Unary operators group from right to left, while binary operators group from left to right. Use the unary minus (-) to reverse the algebraic sign of a value.

To force a desired order of evaluation, use parentheses. 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. SQL includes the following comparison operators:

Comparison Operator	Description
=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

In addition, you can specify the comparison operator "not equal to" (<>) using "!=" or "^=".

All comparison operators are of equal precedence.

The equal sign (=) also serves as the assignment operator in assignment operations. For a discussion of assignment operations, see Assignment Operations (see page 95).

Logical Operators

There are three logical operators in SQL:

- Not

- And

- Or

The not operator has the highest precedence, followed by the and operator, with the or operator having the least precedence. Use parentheses to change this behavior. For example, the following expression:

```
exprA or exprB and exprC
```

is evaluated as:

```
exprA or (exprB and exprC)
```

To change the order of evaluation, use parentheses:

```
(exprA or exprB) and exprC
```

When parentheses are used as shown, the DBMS Server evaluates (*exprA* **or** *exprB*) first, then uses the and operator for the result with *exprC*.

Parentheses can also be used to change the default evaluation order of a series of expressions combined with the same logical operator. For example, the following expression:

```
exprA and exprB and exprC
```

is evaluated as:

```
(exprA and exprB) and exprC
```

To change this default left-to-right grouping, use parentheses as follows:

```
exprA and (exprB and exprC)
```

The parentheses direct the DBMS Server to use the and operator for *exprB* and *exprC,* then use the and operator for that result with *exprA*.

SQL Operations

This basic SQL operations supported include:

- String concatenation operations
- Assignment operations
- Arithmetic operations
- Ingresdate operations

String Concatenation Operations

To concatenate strings, use either the operator **+** or **||**.

For example:

```
'This ' + 'is ' + 'a ' + 'test.'
```

gives the value:

```
'This is a test.'
```

Likewise:

```
'Hello' || 'World'
```

gives the value:

```
'Hello World'
```

Also, the concat function can be used to concatenate strings. For details, see String Functions (see page 118).

Assignment Operations

An assignment operation 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. Assignments can also occur within a database procedure.

When an assignment operation occurs, the data types of the assigned value and the receiving column or variable must either be the same or comparable.

- If the data types are not the same, comparable data types are converted.

- If the data types are not comparable, convert the assignment value into a type that is the same or comparable with the receiving column or variable.

For information about the type conversion functions, see Default Type Conversion (see page 100).

Character String Assignment

All character types are comparable with one another and with integer and float types. Any character string can be assigned to any column or variable of character data type. A character string can also be assigned to a column or variable of integer or float type, as long as the content of the string is a valid numeric value. The result of the assignment depends on the types of the assignment string and the receiving column or variable.

Assigned String	Receiving Column or Variable	Description
Fixed-length (c or char)	Fixed-length	The assigned string is truncated or padded with spaces if the receiving column or variable is not the same length as the fixed length string. If the assigned string is truncated to fit into a host variable, a warning condition is indicated in SQLWARN. For a discussion of the SQLWARN indicators, see SQL Communications Area (SQLCA) (see page 241).

Assigned String	Receiving Column or Variable	Description
Fixed-length	Variable-length (varchar, long varchar, or text)	The trailing spaces are trimmed. If the receiving column or variable is shorter than the fixed length string, the fixed length string is truncated from the right side. If the assignment was to a variable, a warning condition is indicated in SQLWARN. For a discussion of the SQLWARN indicators, see SQL Communications Area (SQLCA) (see page 241). **Note:** If a long varchar value over is assigned to another character data type, the result is truncated at the maximum row size configured but not exceeding 32,000 (16,000 in a UTF8 instance).
Variable-length (varchar, long varchar, or text)	Fixed-length	The assigned string is truncated or padded with spaces if the receiving column or variable is not the same length as the variable length string. **Note:** If a long varchar value over is assigned to another character data type, the result is truncated at the maximum row size configured but not exceeding 32,000 (16,000 in a UTF8 instance).
Variable-length	Variable-length	The variable length string is truncated if the receiving column or variable is not long enough.

String Truncation

If an attempt is made to insert a string value into a table column that is too short to contain the value, the string is truncated.

String truncation can occur as a result of the following statements:

- COPY
- CREATE TABLE...SELECT
- INSERT
- UPDATE

-string_truncation Flag—Specify Error Handling for String Truncation

To specify error handling for string truncation, use the -string_truncation=*option* flag on the CONNECT statement, specified when a session connects to a database.

The *option* can be one of the following:

ignore

(Default) Truncates and inserts the string. No error or warning is issued.

fail

Does not insert the string, issues an error, and aborts the statement.

This flag can also be specified on the command line for Ingres operating system commands that accept SQL option flags. For details about SQL option flags, see the sql command description in the *Command Reference Guide*.

Numeric Assignments

All numeric types are compatible with one another and with character types. Money is compatible with all of the numeric and string data types.

Numeric assignments follow these rules:

- The DBMS Server can truncate leading zeros, or all or part of the fractional part of a number if necessary. If truncation of the non-fractional part of a value (other than leading zeros) is necessary, an overflow error results. These errors are reported only if the -numeric_overflow flag is set to warn or fail. For information about the -numeric_overflow flag, see the sql command description in the *Command Reference Guide.*

- If the receiving column or variable specifies more digits to the right of the decimal point than is present in the assignment value, the assignment value is padded with trailing zeros.

- When a float, float4, decimal, or money value is assigned to an integer column or variable, the fractional part is truncated.

- When a decimal value with a scale greater than two is assigned to a money column or variable, the fractional value is rounded.

ANSI Date/Time

The ANSI date/time data types are compatible between themselves and string columns, and follow the rules of coercion described in Coercion Between Date/Time Data Types (see page 72).

The date/time values of all date/time types can be assigned to string columns. The result is a display version of each value. String values can also be assigned to date/time columns, as long as the string values correspond to the acceptable input format for the particular date/time type.

Note: When a date/time value is retrieved from a database, it is always presented to the user in the appropriate display format in a string variable.

Ingresdate

The ingresdate data type is compatible with string data types if the value in the string is a valid representation of a date or time input format.

Absolute date or interval column values can be assigned to an ingresdate column. In addition, a string literal, a character string host variable, or a character string column value can be assigned to an ingresdate column if its value conforms to the valid input formats for Ingres dates or times.

When a date value is assigned to a character string, the DBMS Server converts the date to the display format. For more information about date display formats, see Date and Time Display (see page 71).

Types of Logical Keys

There are two types of logical keys:

Table_key—This is comparable only with another table_key or a char that has a length of 8 bytes (char(8)).

Object_key—This is comparable only with another object_key or a char that has a length of 16 bytes (char(16)).

If a logical key column is declared as system_maintained, the DBMS Server assigns the values to that column. System_maintained logical key columns cannot be updated. If a logical key column is declared as not system_maintained, values must be assigned to the column.

In embedded SQL programs, if values are assigned using host variables, the host variables must be char(8)-comparable for table_key columns, and char(16)-comparable variables for object_key columns.

Values can be assigned to logical keys, not system_maintained, using a hex constant or a string literal. For information about the format of a hex constant, see String Literals.

Values assigned to table_keys must be 8 bytes long. Values assigned to object_keys must be 16 bytes long. The following example inserts the value 1 into a table_key column using a hex constant:

```
insert into test (tablekey) values (table_key(X'0000000000000001'));
```

The previous statement inserts 7 bytes containing 0, followed by 1 byte containing 1. The value is explicitly converted to a table key using the table_key conversion function.

The following example assigns the value 'abc' (padded to 8 characters for data type compatibility) to a logical key column:

```
insert into test (tablekey) values (table_key('abc'));
```

Null Value 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 "Working with Embedded SQL."

To ensure that a null is not assigned to a column, use the Ifnull Function.

Arithmetic Operations

An arithmetic operation combines two or more numeric expressions using the arithmetic operators to form a resulting numeric expression. For details about arithmetic operators, see Arithmetic Operators (see page 91).

Before performing any arithmetic operation, the participating expressions are converted to identical data types. After the arithmetic operation is performed, the resulting expression too has that storage format.

Default Type Conversion

When two numeric expressions are combined, the data types of the expressions are converted to be identical; this conversion determines data type of the result. The expression having the data type of lower precedence is converted to the data type of the higher. The order of precedence among the numeric data types is, in highest-to-lowest order:

- money
- float4
- float
- decimal
- integer8 (bigint)
- integer
- smallint
- integer1 (tinyint)

For example, in an operation that combines an integer and a floating point number, the integer is converted to a floating point number. If the operands are two integers of different sizes, the smaller is converted to the size of the larger. The conversions are done before the operation is performed.

The following table lists the data types that result from combining numeric data types in expressions:

	integer1	smallint	integer	integer8	decimal	float	float4	money
integer1	integer	integer	integer	integer8	decimal	float	float4	money
smallint	integer	integer	integer	integer8	decimal	float	float4	money
integer	integer	integer	integer	integer8	decimal	float	float4	money
integer8	integer8	integer8	integer8	integer8	decimal	float	float4	money
decimal	decimal	decimal	decimal	decimal	decimal	float	float4	money
float	float	float	float	float	float	float	float4	money
float4	float4	float4	float4	float4	float4	float4	float4	money
money	money	money	money	money	money	money	money	money

For example, for the expression:

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

the first operator (+) combines a float4 expression (job.lowsal) with a smallint constant (1000). The result is float4. The second operator (*) combines the float4 expression with a smallint constant (12), resulting in a float4 expression.

To convert one data type to another, use data type conversion functions. For details, see Data Type Conversion in this chapter.

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	Largest 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 (largest number of fractional digits) + 3 (largest number of non-fractional digits) + 1 = 7

Scale = 3

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

Result:

0569.124

If exponentiation is performed on a decimal value, the resulting data type is float.

Specify Error Handling for Arithmetic Errors

To specify error handling for numeric overflow, underflow, and division by zero, use the CONNECT statement -numeric_overflow=*option* flag. Error-handling options are:

Option	Description
ignore	No error is issued.
warn	A warning message is issued.
fail (default setting)	An error message is issued and the statement that caused the error is aborted. To obtain ANSI-compliant behavior, specify this option (or omit the numeric_overflow flag).

This flag can also be specified on the command line for Ingres operating system commands that accept SQL option flags. For details about SQL option flags, see the sql command description in the *Command Reference Guide*.

Date/Time Arithmetic

The following arithmetic is supported for date/time data types. Any combinations that do not conform result in errors E_AD5068_ANSIDATE_DATESUB and E_AD5069_ANSIDATE_DATEADD.

ansidate	-	ansidate	=	interval day to second
time	-	time	=	interval day to second
timestamp	-	timestamp	=	interval day to second
ingresdate	+/-	ingresdate	=	ingresdate

Note: The second operand cannot be an absolute date.

ansidate	+/-	interval type	=	ansidate
time	+/-	interval type	=	time
timestamp	+/-	interval type	=	timestamp
interval year to month	+/-	interval year to month	=	interval year to month
interval day to second	+/-	interval day to second	=	interval day to second

ANSI Date/Time Arithmetic

The following arithmetic operations can be coded for the ANSI date/time types. These operations allow the computation of intervals from date/time values and new date/time values from existing date/time values and intervals.

date/time	-	date/time	=	interval
date/time	+	interval	=	date/time
interval	+	date/time	=	date/time
date/time	-	interval	=	date/time
interval	+	interval	=	interval
interval	-	interval	=	interval
interval	*	numeric expression	=	interval
numeric expression	*	interval	=	interval
interval	/	numeric expression	=	interval

The operands must be compatible. For example, you can subtract a date value from another date value or from a timestamp, but you cannot subtract a date value from a time value, or vice versa. Likewise, only intervals of the same type (YEAR TO MONTH or DAY TO SECOND) can be combined in an operation and a YEAR TO MONTH interval cannot be added to or subtracted from a TIME value.

ANSI Date/Time Comparisons

ANSI date/time values can be compared only to values of the same type. When the timezone setting of a time or timestamp value does not match that of its comparand, time or timestamp values WITH TIME ZONE or WITH LOCAL TIME ZONE are considered to be stored in GMT and can be compared directly. If one comparand is time or timestamp WITH TIME ZONE or WITH LOCAL TIME ZONE and the other comparand is WITHOUT TIME ZONE, the value of the WITHOUT TIME ZONE comparand is assumed to be in local time of the session default time zone. It is converted to GMT based on that assumption and the comparison proceeds.

Ingresdate Arithmetic

Ingresdate values can be added and subtracted, but cannot be multiplied or divided. The following lists the results of date arithmetic:

Addition:

interval	+	interval	=	interval
interval	+	absolute	=	absolute

Subtraction:

interval   -   interval   =   interval

absolute   -   absolute   =   interval

absolute   -   interval   =   absolute

When adding intervals, each of the units is added.

For example:

```
date('6 days') + date('5 hours')
```

yields, 6 days 5 hours, while:

```
date('4 years 20 minutes') + date('6 months 80 minutes')
```

yields, 4 years 6 months 1 hour 40 minutes.

In the above example, 20 minutes and 80 minutes are added and the result is 1 hour 40 minutes. 20 minutes plus 80 minutes equals 100 minutes, but this result overflows the minute time unit because there are 60 minutes in an hour. Overflows are propagated upward except when intervals are added. In the above example, the result is 1 hour 40 minutes. However, days are not propagated to months. For example, if 25 days is added to 23 days, the result is 48 days.

When intervals or absolute dates are subtracted, the result is returned in appropriate time units. For example, if the following subtraction is performed:

```
date('2 days') - date('4 hours')
```

the result is 1 day 20 hours.

Date constants can be converted into numbers of days relative to an absolute date. For example, to convert today's date to the number of days since January 1, 1900:

```
num_days = int4(interval('days', 'today' - date('1/1/00')))
```

To convert the interval back to a date:

```
(date('1/1/00') + concat(char(num_days), ' days'))
```

where num_days is the number of days added to the date constant.

Adding a month to a date always yields the same date in the next month. For example:

```
date('1-feb-98') + '1 month'
```

yields March 1.

If the result month has fewer days, the resulting date is the last day of the next month. For instance, adding a month to May 31 yields June 30, instead of June 31, which does not exist. Similar rules hold for subtracting a month and for adding and subtracting years.

Dates that are stored without time values can acquire time values as a result of date arithmetic. For example, the following SQL statements create a table with one date column and store today's date (with no time) in the column:

```
create table dtest (dcolumn date);
insert into dtest (dcolumn) values (date('today'));
```

If the contents of the date column is selected using the following query:

```
select dcolumn from dtest;
```

a date with no time is returned. For example:

```
09-aug-2001
```

If date arithmetic is used to adjust the value of the date column, the values in the column acquire a time. For example:

```
update dtest set dcolumn=dcolumn-date('1 hour');
select dcolumn from dtest;
```

returns the value:

```
08-aug-1998 23:00:00
```

Comparing Ingresdates

In comparisons, a blank (default) ingresdate is less than any interval ingresdate. All interval ingresdates are less than all absolute ingresdates. Intervals are converted to comparable units before they are compared. For instance, before comparing ingresdate('5 hours') and ingresdate('200 minutes'), both the hours and minutes are converted to milliseconds internally before comparing the values. Ingresdates are stored in Greenwich Mean Time (GMT). For this reason, 5:00 PM Pacific Standard Time is equal to 8:00 PM Eastern Standard Time.

Note: Support of blank date values and the ability to compare absolute and interval values are available only with the ingresdate type and not with the ANSI date/time data types.

SQL Functions

Functions can be used in the following SQL statements:

- SELECT
- INSERT
- UPDATE
- DELETE
- WHILE
- IF

Scalar functions take single-valued expressions as their argument. *Aggregate* functions take a set of values (for example, the contents of a column in a table) as their argument. Aggregate functions cannot be used in IF or WHILE statements.

Scalar Functions

The scalar functions require either one or two single-value arguments. Scalar functions can be nested to any level.

The types of scalar functions are as follows:

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

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:
```

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.) Type conversions can also be specified using the CAST expression (see page 148).

Name	Operand Type	Result Type	Description
ansidate(*expr*)	c, text, char, varchar, ingresdate, ansidate, timestamp with time zone, timestamp without time zone, timestamp with local time zone	ansidate	Converts a c, char, varchar text, ingresdate, ansidate, or timestamp type to internal ansidate representation.

Name	Operand Type	Result Type	Description
byte(*expr* [, *len*]) or binary(*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* bytes. *Len* must be a positive integer value that does not exceed the length of the *expr* string.
date(*expr*) or ingresdate(*expr*)	c, text, char, varchar	ingresdate	Converts a c, char, varchar or text string to internal date representation.

Name	Operand Type	Result Type	Description
decimal(*expr* [,*precision*[,*scale*]]) or numeric(*expr* [,*precision*[,*scale*]])	any except ingresdate and ANSI date/time	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.

Name	Operand Type	Result Type	Description
dow(*expr*)	ingresdate	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*)	any	varchar	Returns the hexadecimal representation of the internal Ingres form of the argument expression. The length of the result is twice the length of the argument, because the hexadecimal equivalent of each byte of the argument requires two bytes. For example, hex('ABC') returns '414243' (ASCII) or 'C1C2C3' (EBCDIC). Also, hex(int4(125)) returns '0000007D', the hexadecimal equivalent of the 4 byte binary integer 125.
int1(*expr*) or tinyint(*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 occurs if the integer portion of a floating point or decimal value is too large to be returned in the requested format.
int2(*expr*) or smallint(*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 occurs if the integer portion of a floating point or decimal value is too large to be returned in the requested format.
int4(*expr*) or int(*expr*) or integer(*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 occurs if the integer portion of a floating point or decimal value is too large to be returned in the requested format.
int8(*expr*) or bigint(*expr*)	c, char, varchar, text, float, money, decimal, integer1, smallint, integer	integer8	Converts the specified expression to integer. Decimal and floating point values are truncated. Numeric overflow occurs if the integer portion of a floating point or decimal value is too large to be returned in the requested format.

Name	Operand Type	Result Type	Description
interval_dtos(*expr*)	c, text, char, varchar, ingresdate, interval day to second	interval day to second	Converts c, text, char, varchar, ingresdate or interval year to month types to internal interval day to second representation.
interval_ytom(*expr*)	c, text, char, varchar, ingresdate, interval year to month	interval year to month	Converts c, text, char, varchar, ingresdate or interval year to month types to internal interval year to month representation.
long_byte(*expr*) or long_binary(*expr*)	any	long byte	Converts the expression to long byte binary data.
long_varchar (*expr*)	c, char, varchar, text, long varchar, long byte	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.
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.
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.

Name	Operand Type	Result Type	Description
text(*expr* [, *len*])	any	text	Converts argument to text string. 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* string.
time_local(*expr*)	c, text, char, varchar, ingresdate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	time with local timezone	Converts a c, char, varchar text, ingresdate, time, or timestamp type to internal time with local time zone representation.
time_with_tz(*expr*)	c, text, char, varchar, ingresdate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	time with time zone	Converts a c, char, varchar text, ingresdate, time, or timestamp types to internal time with time zone representation.
time(*expr*) or time_wo_tz(*expr*)	c, text, char, varchar, ingresdate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	time without timezone	Converts a c, char, varchar text, ingresdate, time, or timestamp type to internal time without time zone representation.

Name	Operand Type	Result Type	Description
timestamp_local(*expr*)	c, text, char, varchar, ingresdate, ansidate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	timestamp with local time zone	Converts a c, char, varchar text, ingresdate, ansidate, time, or timestamp type to internal time with local time zone representation.
timestamp_with_tz(*expr*)	c, text, char, varchar, ingresdate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	timestamp with timezone	Converts a c, char, varchar text, ingresdate, ansidate, time, or timestamp type to internal timestamp with time zone representation.
timestamp(*expr*) or timestamp_wo_tz(*expr*)	c, text, char, varchar, ingresdate, ansidate, time without time zone, time with time zone, time with local time zone, timestamp without time zone, timestamp with time zone, timestamp with local time zone	timestamp without time zone	Converts a c, char, varchar text, ingresdate, ansidate, time, or timestamp type to internal timestamp without time zone representation.

Name	Operand Type	Result Type	Description
unhex(*expr*)	varchar, c, text	varchar	Returns the opposite of the hex function. For example, unhex('61626320') returns 'abc' and unhex('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: Normally 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, the value is converted to a backslash (\), followed by the numeric value of the hex digit pair as a three-digit octal value.
varbyte(*expr* [, *len*]) or varbinary(*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* bytes. *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 of Argument	Result Length
byte	Length of operand
byte varying	Length of operand
c	Length of operand

Data Type of Argument	Result Length
char	Length of operand
decimal	Depends on precision and scale of column
float & float4	11 characters; 12 characters on IEEE computers
ingresdate	25 characters
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

SQL 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.
acos(n)	all numeric types	float	Arccosine of cosine value n
asin(n)	all numeric types	float	Arcsine value of sine value n
atan(n)	all numeric types	float	Arctangent of n; returns a value from (-pi/2) to pi/2.
atan2 (x, y)	all numeric types	float	Arctangent of angle defined by coordinate pair (x, y)
ceil(n) ceiling(n)	all numeric types	decimal	Returns the smallest integer greater than or equal to the specified numeric expression.
cos(n)	all numeric types	float	Cosine of n; returns a value from -1 to 1.

Name	Operand Type	Result Type	Description
exp(*n*)	all numeric types and money	float	Exponential of *n*.
floor(*n*)	all numeric types	decimal	Returns the largest integer less than, or equal to, the specified numeric expression.
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.
pi	None	float	Value of pi (ratio of the circumference of a circle to its diameter)
power(*x*,*y*)	all numeric types	float	*x* to the power of *y* (identical to x ** y).
round(*n*,*i*)	all numeric types	decimal	Rounds value at the i'th place right or left of the decimal, depending on whether *i* is greater or less than 0.
sign(*n*)	all numeric types	integer	-1 if *n* < 0, 0 if *n* = 0, +1 if *n* > 0
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*.
tan(*n*)	all numeric types	float	Tangent value of angle *n*
trunc(*x* [,*y*]) truncate(*x* [,*y*])	all numeric types	decimal	Truncates x at the decimal point, or at y places to the right or left of the decimal, depending on whether y is greater or less than 0.

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

radians = degrees / 360 * 2 * pi()

The functions acos, asin, atan and atan2 return a value in radians.

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 concatenates x.firstname with the first concatenation result.

The **+** or **||** operator can also be used to concatenate strings:

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

String Functions That Do Not Accept Long Varchar or Long Byte Types

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

- Charextract
- Byteextract
- Locate
- Pad
- Shift
- Squeeze
- Trim
- ANSI Trim
- Notrim
- Left
- Right
- Substring

To apply any of these 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.

String Functions and the UTF8 Character Set

Note: For the UTF8 character set, the character data is multibyte string and the actual number of bytes for the data could be more than the number of characters. If the output buffer for any string operation is not sufficiently large to hold the multibyte string, the result will be truncated at a character boundary.

String Functions Supported

The following table lists the string functions supported in SQL.

The expressions *c1* and *c2*, representing function arguments, can be any of the string types (char, varchar, long varchar, c, text, byte, varbyte, long varbyte) or any of the Unicode types (nchar, nvarchar, long nvarchar), except where noted. The expressions *len*, *n, n1*, *n2* or *nshift*, representing function arguments, are the integer type. For string functions operating on one of the string types, the integer arguments represent character (or 8-bit octet) counts or offsets. For string functions operating on one of the Unicode types, the integer arguments represent "code point" (or 16-bit Unicode characters) counts or offsets.

Name	Result Type	Description
ascii(*v1*)	any character type	Returns the character equivalent of the value *v1*, which is an expression of either character or numeric type.
byteextract(*c1*,*n*)	char	Returns the *n*th byte of *c1*. If *n* is larger than the length of the string, the result is a blank character. It does not support long varchar or long nvarchar arguments.
charextract(*c1*,*n*)	varchar or nchar	Returns the *n*th character or code point of *c1*. If *n* is larger than the length of the string, the result is a blank character. It does not support long varchar or long nvarchar arguments.
character_length(*c1*)	integer	Returns the number of characters in *c1* without trimming blanks, as is done by the length() function.
chr(*n*)	character	Converts integer into corresponding ASCII code. If n is greater than 255, the conversion is performed on n mod 256.
collation_weight(*c1* [,*n1*])	varbyte	Returns the collation weight of any char, c, varchar, text, nchar, or nvarchar value *c1*. *n1* is an optional collation ID when the collation weight is desired relative to a specific collation.

Name	Result Type	Description
concat(*c1*,*c2*)	any character or Unicode 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.
		This function does not support long nvarchar arguments.
ii_ipaddr(*n*)	byte	Converts an IPv6 address (*n*) to a byte data type of length 4. Each digit of the IP address is converted into a single byte in the hex result.
		For example: 255.255.0.0 produces 'ffff0000'; 10.20.30.40 produces '0a141e28'.
left(*c1*,*len*)	any character or Unicode 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*.
		This function does not support long nvarchar arguments.
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 *c1* is a variable-length string, returns the number of characters actually in *c1*.

Name	Result Type	Description
locate(*c1*,*c2*)	smallint	Returns the location of the first occurrence of *c2* within *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.
		This function does not support long varchar or long nvarchar arguments.
lowercase(*c1*) or lower(*c1*)	any character or Unicode data type	Converts all upper case characters in *c1* to lower case.
lpad(expr1, n [, expr2])	any character data type	Returns character expression of length n in which expr1 is prepended by n-m blanks (where m is length(expr1)) or, if expr2 is coded, enough copies of expr2 to fill n-m positions at the start of the result string.
ltrim(*expr*)	any character data type	Returns character expression with leading blanks removed.
octet_length(*c1*)	integer	Returns the number of 8-bit octets (bytes) in *c1* without trimming blanks, as is done by the length() function.
position (*c1* IN *c2*)	smallint	ANSI compliant version of locate function.
replace(expr1, expr2, expr3)	any character data type	Returns character expression derived from expr1 in which all instances of expr2 have been replaced by expr3.

Name	Result Type	Description
right(*c1*,*len*)	any character or Unicode 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*. This function does not support long nvarchar arguments.
rpad(*expr1*, *n* [, *expr2*])	any character data type	Returns character expression of length n in which expr1 is appended by n-m blanks (where m is length(expr1)) or, if expr2 is coded, enough copies of expr2 to fill n-m positions at the end of the result string.
rtrim(*expr*)	any character data type	Returns character expression with trailing blanks removed.
shift(*c1*,*nshift*)	any character or Unicode 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*. This function does not support long varchar or long nvarchar arguments.
size(*c1*)	smallint	Returns the *declared* size of *c1* without removal of trailing blanks.

Name	Result Type	Description
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.
		This function does not support long varchar or any Unicode arguments.
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.
		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.
		This function does not support long varchar or long nvarchar arguments.
substring(*c1* from *loc*[FOR l*en*])	varchar or nvarchar	Returns part of *c1* starting at the *loc* position and either extending to the end of the string or for the number of characters/code points in the *len* operand. The result format is a varchar or nvarchar the size of *c1*.

Name	Result Type	Description
trim(*c1*)	text or varchar	Returns *c1* without trailing blanks. The result has the same length as *c1*. This function does not support long varchar or long nvarchar arguments.
trim([[*trim_spec*] [*c1*] FROM] *c2*)	any character string variable	ANSI compliant version of the trim function. The *trim_spec* is LEADING, TRAILING, or BOTH, indicating that the result will return *c2* with all occurrences of *c1* removed from the beginning, end, or both. In the absence of *trim_spec*, BOTH is assumed. In the absence of *c1*, the space is assumed.
notrim(*c1*)	any character string variable	Retains trailing blanks when placing a value in a varchar column. This function can be used only in an embedded SQL program. For more information, see the *Embedded SQL Companion Guide*.
uppercase(*c1*) or upper(*c1*)	any character 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

| 1st String | 2nd String | Trim Blanks | | Result Type |
		from 1st?	from 2nd?	
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
nchar	nchar	No	No	nchar
nchar	nvarchar	No	No	nchar
nvarchar	nchar	No	No	nchar
nvarchar	nvarchar	No	No	nvarchar
Byte	Byte	No	No	Byte
Byte	Varbyte	No	No	Byte
Varbyte	Byte	No	No	Byte
Varbyte	Varbyte	No	No	Varbyte
Byte	Longbyte	No	No	Longbyte
Varbyte	Longbyte	No	No	Longbyte
Longbyte	Longbyte	No	No	Longbyte

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

The following date functions extract the specified portion of a date, time, or timestamp. They can be used with the ansidate and ingresdate data types.

year()

Extracts the year portion of a date or timestamp.

quarter()

Extracts the quarter corresponding to the date or timestamp. Quarters are numbered 1 through 4.

month()

Extracts the month portion of a date or timestamp.

week()

Extracts the number of the week of the year that the date or timestamp refers to. 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. Weeks are numbered 1 to 53.

week_iso()

Extracts the number of the week of the year that the date or timestamp refers to, and conforms to ISO 8601 definition for number of the week. Week_iso begin on Monday, but the first week is the week that has the first Thursday of the year. 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.

day()

Extracts the day portion of a date or timestamp.

hour()

Extracts the hour portion of a time or timestamp.

minute()

Extracts the minute portion of a time or timestamp.

second()

Extracts the second portion of a time or timestamp.

microsecond()

Extracts the fractions of seconds portion of a time or timestamp as microseconds.

nanosecond()

Extracts the fractions of seconds portion of a time or timestamp as nanoseconds.

Examples:

day('2006-12-15 12:30:55.1234') returns 15

second('2006-12-15 12:30:55.1234') returns 55.1234

Date Functions for Ingresdate Data Type

Note: The functions described in this section apply only to the ingresdate data type and not to the ANSI date/time data types (date, time, timestamp, interval). The effect of the ingresdate functions can be derived by first coercing the ANSI date/time value to ingresdate using the ingresdate() function, then executing the desired ingresdate function.

SQL 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 (see page 108).

Some date functions require specifying of 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, ses, 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)* 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

Name	Format (Result)	Description
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
isdst(*date*)	integer	Returns 1 if *date* falls within Daylight Saving Time for the session timezone, else 0.
_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 Note that this function formats a leading space for day values less than 10.
_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

Name	Format (Result)	Description
_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

Truncate Dates using date_trunc Function

Use the date_trunc function to group all the dates within 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 falls into the previous year.

Using Date_part

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

Extract Function—Extract a Field from a Date/Time Value

The EXTRACT function can be used in an SQL statement to extract a particular field from date/time value.

The syntax for EXTRACT is as follows:

```
EXTRACT (component FROM extract_source)
```

component

Is the field you want to extract. Valid values are:

YEAR

MONTH

DAY

HOUR

MINUTE

SECOND

TIMEZONE_HOUR

TIMEZONE_MINUTE

extract_source

Is any date/time or interval value expression.

Examples:

```
SELECT EXTRACT (YEAR FROM datecol) FROM datetable;
SELECT EXTRACT (MONTH FROM CURRENT_TIME);
SELECT EXTRACT (HOUR FROM timecol) FROM tab1 \g
SELECT * FROM tx WHERE EXTRACT(HOUR FROM tab_time) = 17 \g
```

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.

There are six external bit-wise functions:

- **bit_add**—The logical "add" of two byte operands; any overflow is disregarded.

- **bit_and**—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**—The logical "not" of two byte operands.

- **bit_or**—The logical "or" of two byte operands. For example, if either or both bits are 1, the answer is 1.

- **bit_xor**—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 Functions

The hash 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: Because 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.

Random Number Functions

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 enhance the "randomness" of the returned value. Note that the seed value can be any integer.

If you omit the value, 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 within the specified range (that is, l >= x <= h).

- randomf(l,h)— Passing two integer values generates an integer result within the specified range; passing two floats generates a float within the specified range; passing an int and a float causes them to be coerced to an int and generates an integer result within the specified range (that is, l >= x <= h).

Aggregate Functions

Aggregate functions take a set of values as their argument.

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;
```

SQL Aggregate Functions

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).

Name	Result Data Type	Description
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 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 Count(*) Function (see page 140).

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 (indep_parm, dep_parm)	integer	Count of rows with non-null values for both dependent and independent variables.

Name	Result Data Type	Description
covar_pop (indep_parm, dep_parm)	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).
covar_samp (indep_parm, dep_parm)	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 (indep_parm, dep_parm)	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 (indep_parm, dep_parm)	float	Square of the correlation coefficient.
regr_slope (indep_parm, dep_parm)	float	Slope of the least-squares-fit linear equation determined by the (independent variable, dependent variable) pairs.
regr_intercept (indep_parm, dep_parm)	float	Y-intercept of the least-squares-fit linear equation determined by the (independent variable, dependent variable) pairs.
regr_sxx (indep_parm, dep_parm)	float	Sum of the squares of the independent variable.
regr_syy (indep_parm, dep_parm)	float	Sum of the squares of the dependent variable.
regr_sxy (indep_parm, dep_parm)	float	Sum of the product of the independent variable and the dependent variable.

Name	Result Data Type	Description
regr_avgx (indep_parm, dep_parm)	float	Average of the independent variables.
regr_avgy (indep_parm, dep_parm)	float	Average of the dependent variables.

Count(*) Function

The count function 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

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: 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 Types

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. For a description of comparable data types, see Assignment.

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 are decimal, the data type of the result returned by ifnull 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.

Benefits of Using a UUID

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

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

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.

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 basic format of a unique 128-bits (16 octets) UUID:

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.

Field	Data Type	Octet Number	Note
node	unsigned 48 bit integer	10-15	The spatially unique node identifier.

SQL Functions for UUID Implementation

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

- uuid_create ( );
- uuid_from_char (c);
- uuid_compare(uuid1, uuid2);

uuid_compare(uuid1, uuid2) Function

The uuid_compare(uuid1, uuid2) function, upon completion, returns an integer value of:

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

```
* select uuid_compare(u1,u2) from uuidtable\g

P,,,,T
.col1
5,,,,6
.     1.
F,,,,G
```

uuid_from_char(c) Function

The uuid_from_char(c) function converts a generated UUID from character representation into byte representation:

```
//
// Inserts a generated UUID in character format.
//

* insert into uuidtochar values ();\g

* select * from  uuidtochar;\g

P,,,,,,,,,,,,,T
.c1
5,,,,,,,,,,,,,6
.f703c440-b35c-01d5-8637-00805fc13ce5.
F,,,,,,,,,,,,,G

//
// converts  UUID into byte representation
//
* select uuid_from_char (u1) from uuidtochar;\g

P,,,,,,T
.col1
5,,,,,,6
.œ\003Ä@³\\\001Õ\2067\221\0134¡\221\013.
F,,,,,,G
```

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)); \g
* insert into uuidtable values (uuid_create(), uuid_create())\g
//
// verify length in byte format
//
* select length(u1) from uuidtable;\g

P,,,,T
.col1
5,,,,6
.     16.
F,,,,G
```

Length returned equals 16 bytes.

Expressions in SQL

Expressions are composed of various operators and operands that evaluate to a single value or a set of values. Some expressions do not use operators; for example, a column name is an expression. 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 > 75000
```

In the preceding 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.

Expressions that contain aggregate functions can appear only in SELECT and HAVING clauses, and aggregate functions cannot be nested.

An expression can be enclosed in parentheses without affecting its value.

Case Expressions

Case expressions provide a decoding capability that allows one expression to be transformed into another. Case expressions can appear anywhere that other forms of expressions can be used. There are two forms of case expressions:

- Simple
- Searched

A simple case expression looks like this:

```
case expr when expr1 then expr2 when expr3 then expr4... [else exprn] end
```

The initial case expression is compared in turn to the expressions in each WHEN clause. The result of the case is the expression from the THEN clause corresponding to the first WHEN clause whose expression is equal to the case expression. If none of the WHEN expressions match the case expression, the result is the value of the expression from the ELSE clause. If there is no ELSE clause, the result is the null value.

The searched case expression syntax looks like this:

```
case when search_conditon1 then expr1 when search_expression2 then expr2...[else
exprn] end
```

The search conditions of each when clause are evaluated in turn. The result of the case is the expression from the THEN clause corresponding to the first WHEN clause whose search condition evaluates to true. If none of the WHEN clause search conditions evaluate as true, the result is the value of the expression from the ELSE clause. If there is no ELSE clause, the result is the null value.

NULLIF, COALESCE Functions

NULLIF and COALESCE are derivative functions of the case expression and can be defined in terms of the case expression. NULLIF returns the null value if its two parameters are equal; otherwise it returns the first parameter value. It can be defined as follows:

```
nullif(expr1, expr2) = case when expr1 = expr2 then null else expr1 end
```

COALESCE simply returns the first non-null value of an arbitrary list of parameters. It can be defined as follows:

```
coalesce(expr1, expr2) = case when expr1 is not null then expr1 else expr2 end
```

```
coalesce(expr1, expr2, ..., exprn) = case when expr1 is not null then expr1
                            else coalesce(expr2, ..., exprn)
```

Cast Expressions

An alternative to the conversions outlined above is the cast expression. Cast expressions can appear anywhere that other forms of expression can and they coerce the source expression into the indicated data type.

They are specified as follows:

```
cast (expr as datatype)
```

where *expr* is the source value expression and *datatype* is any supported Ingres data type.

Sequence Expressions

Sequence expressions return values from defined database sequences. A sequence expression consists of one of two different operators:

- **next value for** [*schema*.]*sequence* or [*schema*.]*sequence*.**nextval**

- **current value for** [*schema.*]*sequence* or [*schema.*]*sequence.***currval**

The next value operator returns the next available value from the referenced sequence. The current value operator returns the previous value returned from the sequence to the executing application. Note that the current value operator cannot be used in a transaction until a next value operator is executed on the same sequence. This prevents transactions from seeing sequence values returned to other executing applications.

Sequence expressions are typically used in INSERT or UPDATE statements to maintain columns that reflect some ordinal relationship with the creation of their containing rows. For example:

```
INSERT INTO T1 VALUES (:hv1, NEXT VALUE FOR mydb.t1seq, …)
```

or

```
INSERT INTO T2 SELECT col1, col2, t2seq.NEXTVAL, …FROM …
```

Sequence expressions can also be used in the select list of a SELECT statement, but not in a where clause, on clause, group by, or having clause.

A "next value" or "current value" expression on a particular sequence is evaluated once per row inserted by an INSERT statement, updated by an UPDATE statement, or added to the result set of a SELECT statement. If several occurrences of a "next value" or "current value" expression on the same sequence are coded in a single statement, only one value is computed for each row touched by the statement. If a "next value" expression and a "current value" expression are coded on the same sequence in the same statement, the "next value" expression is evaluated first, then the "current value" expression (assuring they return the same value), regardless of their order in the statement syntax.

Locking and Sequences

In applications, sequences use logical locks that allow multiple transactions to retrieve and update the sequence value while preventing changes to the underlying sequence definition. The logical lock is held until the end of the transaction.

Predicates in SQL

Predicates are keywords that specify a relationship between two expressions. SQL supports the following types of predicates:

- Comparison
- Like
- Between
- In
- Any-or-All
- Exists
- Is null

These predicates are described in the following sections.

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

Comparison Predicate

The syntax for comparison predicates is as follows:

expression_1 comparison_operator *expression_2*

In a comparison predicate, *expression2* can be a subquery. If *expression2* is a subquery and does not return any rows, the comparison predicate evaluates to false. For information about subqueries, see Subqueries (see page 157). For details about comparison operators, see Comparison Operators (see page 92).

Like Predicate

The like predicate performs pattern matching for the character data types (char, varchar, c, and text) and Unicode data types (nchar and nvarchar). The like predicate has the following syntax:

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

The *expression* can be a column name or an expression containing 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 ignore trailing blanks. If you are matching a char value (that is padded with blanks when it is inserted) or if the value has trailing blanks that were entered by the user, include these trailing blanks in your pattern. For example, if searching a char(10) column for rows that contain the name harold, specify the trailing blanks in the pattern to be matched:

```
name like 'harold    '
```

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

Because blanks are not significant when performing comparisons of c data types, the like predicate returns a correct result whether or not 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.
- Square brackets [ ]. Within escaped square brackets ([and]), a series of individual characters or a range of characters separated by a dash (-) can be specified.

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 that start with S and end with h, disregarding any leading or trailing spaces:

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

To detect a single quote, repeat the quote:

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

Between Predicate

The following table explains the operators between and not between:

Operator	Meaning
y between [asymmetric] *x* and *z*	*x* < = *y* and *y* < = *z*
y not between [asymmetric] *x* and *z*	not (*y* between *x* and *z*)
y between symmetric *x* and *z*	(*x* < = *y* and *y* < = *z*) or (*z* < = *y* and *y* < = *x*)
y not between asymmetric *x* and *z*	not (*y* between symmetric *x* and *z*)

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

In Predicate

The following table explains the operators in and not in:

Operator	Meaning
y in (*x*, ..., *z*)	The in predicate returns true if *y* is equal to one of the values in the list (*x*, ..., *z*).
	(*x*, ..., *z*) represents a list of expressions, each of which must evaluate to a single value. If there is only one expression in the list, the parentheses are optional. None of the expressions (*y*, *x*, or z) can be subqueries.
y not in (*x*, ..., *z*)	Returns true if *y* is not equal to any of the values in the list (*x*, ..., *z*).
	(*x*, ..., *z*) is a list of expressions, each of which must evaluate to a single value. If there is only one expression in the list, the parentheses are optional. None of the expressions (*y*, *x*, or z) can be subqueries.
y in (subquery)	Returns true if *y* is equal to one of the values returned by the *subquery*. The subquery must be parenthesized and can reference only one column in its SELECT clause.
y not in (*subquery*)	Returns true if *y* is not equal to any of the values returned by the *subquery*. The subquery must be specified in parentheses and can reference only one column in its SELECT clause.

Any-or-All Predicate

The any-or-all predicate takes the following 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

The != (instead of <>) can also be used to specify not equal. Include a space between the comparison operator and the keyword any or all.

A predicate that includes the any operator is true if the specified comparison is true for at least one value *y* in the set of values returned by the subquery. If the subquery returns no rows, the any comparison is false.

A predicate that includes the all operator is true if the specified comparison is true for all values *y* in the set of values returned by the subquery. If the subquery returns no rows, the all comparison is 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 name
from employee
where dept = some
        (select dno
        from dept
        where floor = 3);
```

Exists Predicate

The exists predicate takes the following form:

```
[NOT] EXISTS (subquery)
```

It evaluates to true if the set returned by the subquery is not 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 following form:

```
IS [NOT] NULL
```

For example:

```
x is null
```

is true if *x* is a null. Use this predicate to determine whether an expression is null, because you cannot test for null by using the = comparison operator.

Search Conditions in SQL Statements

Search conditions are used in WHERE and HAVING clauses to qualify the selection of data. Search conditions are composed of predicates of various kinds, optionally combined using parentheses and logical operators (and, or, and not). The following are examples of legal 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_%

Description	Example
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 (that is, and, or, not) 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, not unknown. Unknown values are considered false. For more information about predicates and logical operators, see Logical Operators (see page 93).

Subqueries

A *subquery* is a SELECT statement nested within another SELECT statement.

Subqueries in the Where Clause

A subquery in a WHERE clause can be used to qualify a column against a set of rows.

For example, the following subquery returns the department numbers for departments on the third floor. The outer query retrieves the names of employees who work on the third floor.

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

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 in SQL (see page 150).

The syntax of the subquery is identical to that of the subselect, except the SELECT clause must contain only one element. A subquery can see 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. Otherwise the dept column is assumed to be implicitly qualified by empy. The overall query is evaluated by assigning empx each of its values from the employee table and evaluating the subquery for each value of empx.

Note: Although aggregate functions cannot appear directly in a WHERE clause, they can appear in the SELECT clause or the HAVING clause of a subselect, which itself appears within a WHERE clause.

Subqueries in the From Clause (Derived Tables)

Derived tables let you create or simplify complex queries. Useful in data warehousing applications, they provide a way to isolate complex portions of query syntax from the rest of a query.

A *derived table* is the coding of a SELECT in the FROM clause of a SELECT statement.

For example:

```
select relid, x.attname
from (select attrelid, attrelidx, attname,
      attfrml from iiattribute) x, iirelation
where reltid = attrelid and reltidx = x.attrelidx
```

The derived table behaves like an inline view; the rows of the result set of the derived table are read and joined to the rest of the query. If possible, the derived table is flattened into the containing query to permit the query compiler to better optimize the query as a whole.

Some complex queries cannot be implemented without using either pre-defined views or derived tables. The derived table approach is more concise than pre-defined views, and avoids having to define persistent objects, such as views, that may be used for a single query only.

For example, consider a query that joins information to some aggregate expressions. Derived tables allow the aggregates to be defined and joined to non-aggregated rows of some other tables all in the same query. Without derived tables, a persistent view would have to be defined to compute the aggregates. Then a query would have to be coded to join the aggregate view to the non-aggregated data.

Derived Table Syntax

The SELECT in the FROM clause must be enclosed in parentheses and must include a "correlation name."

Following the correlation name, the derived table can include an override list of column names in parentheses, or these column names can be coded with AS clauses in the SELECT list of the derived table.

Columns in the derived table can be referenced in SELECT, ON, WHERE, GROUP BY, and HAVING clauses of the containing query, qualified by the correlation name, if necessary.

Example Queries Using Derived Tables

```
select e.ename from employee e,
    (select avg(e1.salary), e1.dno from employee e1
        group by e1.dno) e2 (avgsal, dno)
    where e.dno = e2.dno and e.salary > e2.salary
```

Changing columns names with AS clause:

```
select e.ename from employee e,
    (select avg(e1.salary) as avgsal, e1.dno from employee e1
        group by e1.dno) e2
    where e.dno = e2.dno and e.salary > e2.salary
```

Chapter 5: Working with Embedded SQL

This chapter discusses the techniques of, and how to work with, Embedded SQL. Topics covered include:

- General syntax and rules of an embedded SQL statement
- Structure of embedded SQL programs
- Host language variables
- Data manipulation with cursors
- Dynamic programming
- Data handlers for large objects

Embedded SQL Statements

Embedded SQL statements refer to SQL statements embedded in a host language such as C or Fortran. Embedded SQL statements include most interactive SQL statements and statements that fulfill the additional requirements of an embedded program.

How Embedded SQL Statements are Processed

Embedded SQL statements are processed by an embedded SQL (ESQL) preprocessor, which converts the ESQL statements into host language source code statements. The resulting statements are calls to a runtime library that provides the interface to Ingres (host language statements are not altered by the ESQL preprocessor). After the program has been preprocessed, it must be compiled and linked according to the requirements of the host language. For details about compiling and linking an embedded SQL program, see the *Embedded SQL Companion Guide*.

General Syntax and Rules of an Embedded SQL Statement

An embedded SQL statement has the following format:

```
[margin] EXEC SQL SQL_statement [terminator]
```

Note: To create forms-based applications, use forms statements. For details, see the *Forms-based Application Development Tools User Guide.*

When creating embedded SQL (ESQL) programs, remember the following points:

- 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 SQL statement. EXEC SQL indicates to the embedded SQL preprocessor that the statement is an embedded SQL statement.

- The terminator, which indicates the end of the statement, is specific to the host language. Different host languages require different terminators and some, such as Fortran, do not require any.

- Embedded SQL statements can be continued across multiple lines according to the host language's rules for line continuation.

- A label can precede an embedded statement if a host language statement in the same place can be preceded by a label. Nothing can be placed between the label and the EXEC SQL keywords.

- Host language comments must follow the rules for the host language.

Some host languages allow the placement of a line number in the margin. For information about language-dependent syntax, see the *Embedded SQL Companion Guide*.

Syntax Conventions Used in this Chapter

In the examples in this chapter, host language program elements are indicated by pseudocode in italics. All of the examples use the semicolon (;) as the statement terminator. In an actual program, however, the statement terminator is determined by the host language.

Structure of an Embedded SQL Program

In general, SQL statements can be embedded anywhere in a program that host language statements are allowed.

The following example shows a simple embedded SQL program that retrieves an employee 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 SQL statements. The sequence of statements in this example illustrates a pattern common to most embedded SQL programs.

```
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;
exec sql select ename, sal
        into :name, :salary
        from employee
        where eno = 23;
print name, salary;
exec sql disconnect;
end program
```

Each program statement is described here:

```
exec sql include sqlca;
```

The INCLUDE statement incorporates the SQL error and status handling mechanism—the SQL Communications Area (SQLCA)—into the program. The SQLCA is used by the WHENEVER statement, which appears later in the program.

```
exec sql begin declare section;
```

Next is an SQL declaration section. Host language variables must be declared to SQL prior to their use in any embedded SQL statements. Host language variables are described in detail in the next section.

```
exec sql whenever sqlerror stop;
```

The WHENEVER statement that follows uses information from the SQLCA to control program execution under error or exception conditions. In general, an error handling mechanism must precede all executable embedded SQL statements in a program. For details about error handling, see Error Handling in the chapter "Working with Transactions and Handling Errors."

```
exec sql connect personnel;
```

Next is a series of SQL and host language statements. The first statement initiates access to the personnel database. A CONNECT statement must precede any references to a database.

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

Next is the familiar SELECT statement, containing a clause that begins with the keyword INTO. The INTO clause associates values retrieved by the SELECT statement with host language variables in the program. Following the INTO keyword are the two host language variables previously declared to SQL: name and salary.

```
print name, salary;
```

This host language statement prints the values contained in the variables.

```
exec sql disconnect;
```

The last SQL statement in the program severs the connection of the program to the database.

Host Language Variables in Embedded SQL

Embedded SQL allows the use of host language variables for many elements of embedded SQL statements. Host language variables can be used to transfer data between the database and the program or to specify the search condition in a WHERE clause.

In embedded SQL statements, host language variables can be used to specify the following elements:

- Database expressions. Variables can generally be used wherever expressions are allowed in embedded SQL statements, such as in target lists and predicates. Variables must denote constant values and cannot represent names of database columns or include any operators.

- 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.

- Miscellaneous statement arguments. Many embedded SQL statement arguments can be specified using host language variables. For more information, see the chapter "Using SQL Statements".

A host language variable can be a simple variable or a structure. All host language variables must be declared to embedded SQL before using them in embedded SQL statements.

For a full discussion of the use of host language variables in embedded SQL, see the *Embedded SQL Companion Guide*.

Variable Declaration

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

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

A program can contain multiple declaration sections. The preprocessor treats variables declared in each declaration section as global to the embedded SQL program from the point of declaration onward, even if the host language considers the declaration to be in local scope.

The variable declarations are identical to any variable declarations in the host language, however, the data types of the declared variables must belong to a subset of host language data types that embedded SQL understands.

The DBMS Server automatically handles the conversion between host language numeric types and SQL numeric types, as well as the conversion between host language character string types and SQL character string types. To convert data between numeric and character types, use one of the explicit conversion functions described in Default Type Conversion in the chapter "Understanding the Elements of SQL Statements." For a list of the data types acceptable to embedded SQL and a discussion of data type conversion, see the *Embedded SQL Companion Guide*.

Note: Host language variables that are not declared to SQL are not processed by the ESQL preprocessor and therefore can include data types that the preprocessor does not understand.

Include Statement

The embedded SQL INCLUDE statement allows external files to be included in your source code. The syntax of the INCLUDE statement is:

```
EXEC SQL INCLUDE filename
```

This statement is commonly used to include an external file containing variable declarations.

For example, assuming you have a file called, myvars.dec, that contains a group of variable declarations, 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.

Variable Usage

After host language variables are declared, use them in your embedded statements. In embedded SQL 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 language variable can have the same name as a database object, such as a column. The preceding colon distinguishes the variable from an object of the same name.

If no value is returned (for example, no rows qualified in a query), the contents of the variable are not modified.

Variable Structures

To simplify data transfer in and out of database tables, embedded SQL allows the usage of variable structures in the SELECT, FETCH, UPDATE, and INSERT statements. Structures must be specified according to the rules of the host language and must be declared in an embedded SQL declare section. For structures to be used in SELECT, INSERT, and UPDATE statements, the number, data type, and order of the structure elements must correspond to the number, data type, and order of the table columns in the statement.

For example, if you have a database table, employee, with the columns, ename (char(20)) and eno (integer), you can declare the following variable structure:

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

and issue the following SELECT statement:

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

It is also legal to specify only the structure name in the statement. If this is done, each variable structure must correspond to the table specified in the statement. The number, data type, and order of the structure elements must correspond to the number, data type, and order of the table columns in the statement.

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

The embedded SQL preprocessor expands the structure name into the names of the individual members. Placing a structure name in the INTO clause has the same effect as enumerating all members of the structure in the order in which they were declared.

A structure can also be used 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*.

Dclgen Utility—Generate Structure

The *dclgen utility* (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.

The dclgen utility can be invoked from the operating system level by executing the following command:

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

language

Defines the embedded SQL host language.

dbname

Defines the name of the database containing the table.

tablename

Defines the name of the database table.

filename

Defines the output file generated by dclgen containing the structure declaration.

structurename

Defines the name of the generated host language structure.

This command creates the declaration file, *filename*, containing a structure corresponding to the database table. The file also includes a DECLARE TABLE statement that serves as a comment and identifies the database table and columns from which the structure was generated. Once the file has been generated, use an embedded SQL INCLUDE statement to incorporate it into the variable declaration section. For details, see Declare Table (see page 469).

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

Indicator Variables

An *indicator variable* is a 2-byte integer variable associated with a host language variable in an embedded SQL statement. Indicator variables enable an application to:

- Detect when a null value is retrieved

- Assign a null value to a table column, form field, or table field column

- Detect character string truncation

Indicator Variable Declaration

Like other host language variables, an indicator variable must be declared to embedded SQL in a declare section.

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

```
host_variable:indicator_variable
```

Or you can use the optional keyword indicator in the syntax:

```
host_variable indicator :indicator_variable
```

When used to detect or assign a null, indicator variables are commonly termed *null indicator variables*.

Specify indicator variables in association with host language variables that contain the following data:

- Database column value

- Constant database expression

- Form field value

- Table field column value

For example, the following example associates null indicators with variables representing column values:

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

Null Indicators and Data Retrieval

When a null value is retrieved into a host language variable that has an associated indicator variable, the indicator variable is set to -1 and the value of the host language variable is not changed. If the value retrieved is not a null, the indicator variable is set to 0 and the value is assigned to the host language 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 variable lists

- Data handlers for long varchar and long byte values

- Database procedure parameters passed by reference

The following example illustrates the use of a null indicator when retrieving data. This example issues a FETCH statement and updates a roster while checking for null phone numbers indicated by the variable, phone_null.

```
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

Use an indicator variable with a host language variable to specify the assignment of a null to a database column. (An indicator variable can also be used to assign a null to a form field or a table field column.) When the DBMS Server assigns the value from a host language variable to one of these objects, it checks the value of the host language variable's associated indicator variable.

If the indicator variable value is -1, a null is assigned to the object and ignores the value in the host language variable. If the indicator variable is any value other than -1, the value of the host language variable is assigned to the object.

If the indicator value is -1 and the object type is not nullable (such as a column created with the NOT NULL clause), an error results.

The following example demonstrates the use of an indicator variable and the null constant with the INSERT statement. For a description of the null constant, see Nulls in the chapter "SQL Data Types."

```
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 VALUES clause of the INSERT statement

- The SET clause of the UPDATE statement

- EXECUTE PROCEDURE statement parameters

All constant expressions in the above clauses can include the keyword NULL. Wherever an indicator variable can be used to assign a null, you can use the keyword NULL.

Indicator Variables and Character Data Retrieval

When a character string is retrieved into a host language variable too small to hold the string, the data is truncated to fit. (If the data was retrieved from the database, the sqlwarn1 field of the SQLCA is set to 'W'.)

If the host language 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 language 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 (two gigabytes) is too large a value to fit in an indicator variable.

Null Indicator Arrays and Host Structures

You can use host structures with the SELECT, FETCH, and INSERT statements wherever these statements allow host language variables to be used. An array of indicator variables can be used in conjunction with a host structure to detect whether a null has been assigned to an element of the host structure.

An indicator array is an array of 2-byte integers, and is typically declared in the same declare section as its associated host language variable structure. Each element of the indicator array acts as an indicator variable to the correspondingly ordered member of the host structure.

The following example declares a host language variable structure, emprec, and an associated indicator array, empind.

```
emprec
    ename       character(20),
    eid         integer,
    esal        float;

empind array(3)of short_integer;
```

The following example illustrates the use of a host structure and indicator array in an embedded statement:

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

In the preceding example, if the value of the employee id column is null, a value of -1 is assigned to the second element of the empind array.

Data Manipulation with Cursors

Cursors enable embedded SQL programs to process, one at a time, the result rows returned by a SELECT statement. 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 SQL statements to perform the following steps:

1. Declare a cursor that selects a set of rows for processing.

2. Open the cursor, thereby selecting the data.

3. Fetch each row from the result set and move the data from the row into host language variables.

4. Update or delete the current row.

5. Close the cursor and terminate processing.

Example: Cursor Processing

The following is an example of cursor processing:

```
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;

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 less than 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 using a cursor in an application, the cursor 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 be used to retrieve data. A cursor is always associated with a SELECT statement. The select is executed when the cursor is opened. Updates can be performed only if the cursor SELECT statement refers to a single table (or updateable view) and does not include any of the following elements:

- Aggregate functions

- 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. The cursor name cannot exceed 32 characters, and can be assigned dynamically. For details, see Summary of Cursor Positioning (see page 181).

Open Cursors

To open a cursor, use the OPEN statement:

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

Opening a cursor executes the associated SELECT statement. The rows returned by the SELECT statement are stored in a temporary result set. The cursor is positioned before the first row in the result table.

Note: If a cursor is closed and reopened, the cursor is repositioned to the beginning of the result table, and does not resume the position it had before it was closed.

Readonly Cursors

Readonly cursors specify that the data does not intend to be updated. The FOR READONLY clause can be specified even if the cursor was declared for update; if this is done, updates on the data cannot be performed.

To improve performance, the DBMS Server pre-fetches (buffers) rows for readonly cursors. Use the SET_SQL(prefetchrows) statement to disable prefetching or to specify the number of rows to prefetch. To determine the number of rows that is prefetched for a particular readonly cursor, open the cursor, issue the INQUIRE_SQL(prefetchrows) statement.

By default the DBMS Server calculates the number of rows it can prefetch, taking into consideration the size of the row being fetched and the dimensions of internal buffers. If, using SET_SQL(prefetchrows), a value larger than the maximum number of rows the DBMS Server can prefetch is specified, prefetchrows is reset to its calculated maximum.

Note: Prefetchrows cannot be set for readonly cursors that return long varchar or long byte columns.

Open Cursors and Transaction Processing

Cursors affect transaction processing as follows:

- Cursors cannot remain open across transactions. The COMMIT statement closes all open cursors, even if a CLOSE CURSOR statement was not issued.

- A savepoint cannot be declared when a cursor is open.

- If an error occurs while a cursor is open, the DBMS Server can roll back the entire transaction and close the cursor.

Fetch Data From Cursor

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 set, and loads the values indicated in the SELECT clause of the DECLARE CURSOR statement 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;
```

Later in the program, the following FETCH statement appears:

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

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

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

You can detect when you have fetched the last row in the result table in the following ways:

- The sqlcode variable in the SQLCA is set to 100 if an attempt is made to fetch past the last row of the result table.

- 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.

- The SQLSTATE variable returns 02000 when the last row has been fetched.

Cursors can only move forward through a set of rows. To refetch a row, close and reopen a cursor.

Fetch Rows Inserted by Other Queries

While a cursor is open, your application can append rows using non-cursor INSERT statements. If rows are inserted after the current cursor position, the rows are or are 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 updateable 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 rows, specify the FOR UPDATE clause when declaring the cursor:

```
EXEC SQL DECLARE cursor_name CURSOR FOR
        select_statement
        FOR [DEFERRED | DIRECT] UPDATE FROM column {, column};
```

The FOR UPDATE clause must list any columns in the selected database table that are intended to be updated. Columns cannot be updated unless they have been declared for update. To delete rows, the cursor does not need to be declared for update.

The cursor UPDATE statement has the following syntax:

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

Cursor Modes

There are two cursor modes: direct and deferred. The default cursor mode is specified when the DBMS Server is started. The default for ANSI/ISO Entry SQL-92 compliance is direct mode.

Direct Mode for Update

Direct mode allows changes to be seen by the program before the cursor is closed. In direct mode, if a row is updated with a value that causes the row to move forward with respect to the current position of the cursor (for example, a key column is updated), the program sees this row again and takes appropriate steps to avoid reprocessing that row.

Deferred Mode for Update

In a Deferred mode, changes made to the current row of a cursor are not visible to the program that opened the cursor until the cursor is closed. Transaction behavior, such as the release of locks and external visibility to other programs, is not affected if deferred update is used. There can be only one cursor open for update in deferred mode at any given time.

Cursor Position for Updates

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

The cursor must be pointing to a row (a fetch has been executed) before a cursor update is performed. The UPDATE statement does not advance the cursor; a fetch is required to move the cursor forward one row. Two cursor updates not separated by a fetch generally cause the same row to be updated twice if the cursor was opened in direct mode, or cause an error if the cursor was opened in deferred mode.

Delete Data Using Cursors

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.

A cursor does not have to be declared for update to perform a cursor delete.

Example: Updating and Deleting with Cursors

The following 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;

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

To close a cursor, issue the CLOSE cursor statement:

```
EXEC SQL CLOSE cursor_name;
```

After the cursor is closed, no more processing can be performed with it unless another OPEN statement is issued. The same cursor can be opened and closed any number of times in a single program. A cursor must be closed before it can be reopened.

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 SQL, see the *Embedded SQL Companion Guide*.

Dynamically Specifying 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 tree structure is represented as a relationship between two columns. In an employee table, employees are assigned an ID number. One of the columns in the employee table contains the ID number of each employee's manager. The ID number column establishes the relationships between employees and managers.

To use dynamically specified cursor names to scan this kind of table, do the following:

- 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, which has the following format:

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

exec sql connect dbname;

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

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

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

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(32)
     ename character_string(32)
     title character_string(20);
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;
```

Cursors versus Select Loops

A select loop is a block of code associated with an embedded SELECT statement; the select loop processes the rows returned by the select. Select loops are typically used when the select returns more than one row.

The select loop is an enhancement of standard SQL. ANSI SQL does not allow more than one row to be retrieved by a SELECT statement. If multiple rows are to be retrieved, the ANSI standard requires the use of a cursor even if the rows are not updated.

Cursors enable an application to retrieve and manipulate individual rows without the restriction of the select loop. Within a select loop, statements cannot be issued that access the database. Use cursors in the following situations:

- When a program needs to scan a table to update or delete rows.

- When a program requires access to other tables while processing rows.

- When more than one table needs to be scanned simultaneously (parallel queries).

- When more than one table needs to be scanned in a nested fashion, for example, in a master-detail application.

The following two examples do the same thing. The first example uses a select loop and the second uses a cursor. Because there are no nested updates and only one result table is being processed, the select method is preferred.

```
 //Select Loop Version

exec sql select ename, eno, sal
    into :name, :empnum, :salary
    from employee
    order by ename;

    exec sql begin;
    print name, salary, empnum;
    exec sql end;



//Cursor Version
```

```
exec sql declare c1 cursor for
    select ename, eno, sal/* No INTO clause */
    from employee
    order by ename;

    exec sql open c1;
    exec sql whenever not found goto closec1;

    loop while more rows
    exec sql fetch c1 into :name, :salary, :empnum;
    print name, salary, empnum;
    end loop;

    closec1:
    exec sql close c1;
```

Dynamic Programming

Dynamic programming allows your applications to specify program elements (including queries, SQL statements, and form names) at runtime. In applications where table or column names are not known until runtime, or where queries must be based on the runtime environment of the application, hard-coded SQL statements are not sufficient.

To support applications such as these, use dynamic SQL. Using dynamic SQL, you can:

- Execute a statement that is stored in a buffer (using the execute immediate statement).

- Encode a statement stored in a buffer and execute it multiple times (using the prepare and execute statements).

- Obtain information about a table at runtime (using the prepare and describe statements).

Note: Dynamic FRS allows an application to transfer data between the form and the database using information specified at runtime.

SQLDA

The descriptor area, called the SQLDA (SQL Descriptor Area), is a host language structure used by dynamic SQL. Dynamic SQL uses the SQLDA to store information about each result column of the SELECT statement and to store descriptive information about program variables. Use the SQLDA when executing a DESCRIBE statement, a PREPARE statement, an EXECUTE statement, or EXECUTE IMMEDIATE statement.

Structure of the SQLDA

Typically, storage for the SQLDA structure is 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*.

The layout of the SQLDA is shown in the following table:

SQLDA Structure	Description
sqldaid	8-byte character array assigned the 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, form, or table field. 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, 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, the program must set sqld to the correct number before the SQLDA is used in a statement. When describing a dynamic SQL statement, if the value in sqld is zero, the described statement is not a SELECT statement.

SQLDA Structure	Description
sqlvar	A sqln-size array composed of the following elements:
	sqltype—2-byte integer indicating the length data type of the column, variable, or field.
	sqllen—2-byte integer indicating the length of the column, variable, or field.
	sqldata—Pointer.
	sqlind—Pointer to indicator variable associated with the host language variable. Your program must also allocate the memory to which this variable points.
	sqlname—The result column name (when a SELECT statement is described).

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

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

Describe Statement and SQLDA

The DESCRIBE statement loads descriptive information about a prepared SQL statement, a form, or the table field of a form into the SQLDA structure. There are three versions of this statement, one for each type of object (statement, form, table field) that can be described.

Dynamic SQL 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 statements that are not SELECT statements are described, a 0 is returned in the sqld field, indicating that the statement was not a SELECT statement. For a complete discussion of how to use describe in a dynamic SQL application, see Preparing and Describing Select Statement (see page 200).

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 a type code is negative, the column, variable, or field described by the sqlvar element is nullable.

Data Type Name	Data Type Code	Nullable
byte	23	No
	-23	Yes
byte varying	24	No
	-24	Yes
C	32	No
	-32	Yes
char	20	No
	-20	Yes
date	3	No
	-3	Yes
decimal	10	No
	-10	Yes
float	31	No
	-31	Yes
integer	30	No
	-30	Yes

Data Type Name	Data Type Code	Nullable
long byte	25	No
	-25	Yes
long varchar	22	No
	-22	Yes
money	5	No
	-5	Yes
text	37	No
	-37	Yes
varchar	21	No
	-21	Yes

Note: Logical keys are returned as char values.

For details about dynamic programming with long varchar and long byte columns, see Using Large Objects in Dynamic SQL (see page 208).

Using Clause

The USING clause directs the DBMS Server to use the variables pointed to by the sqlvar elements of the SQLDA (or other host language variables) when executing the statement.

The USING clause has the following syntax:

```
USING DESCRIPTOR descriptor_name
```

The keyword descriptor is optional in some statements that accept the USING clause.

The statements that accept the USING clause are:

- describe
- execute
- execute immediate
- execute procedure
- fetch
- open
- prepare

Dynamic SQL Statements

Dynamic SQL has the following statements that are used exclusively in a dynamic program:

- Execute immediate

- Prepare and Execute

- Describe

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

This section is an overview of the four statements used in dynamic programs. Detailed discussions on using these statements to execute dynamic statements can be found in Execute a Dynamic Non-select Statement (see page 193) and Execute a Dynamic Select Statement (see page 196). Information about the dynamic versions of the cursor statements is found in Data Manipulation with Cursors (see page 173). In addition, information about the dynamic version of the EXECUTE PROCEDURE statement is found in Execute Procedure of the chapter "Using SQL Statements."

Execute Immediate Statement

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

Use the EXECUTE IMMEDIATE statement to execute all SQL statements except for the following statements:

- call

- close

- connect

- declare

- describe

- disconnect

- enddata

- execute procedure

- execute

- fetch

- get data

- get dbevent

- include

- inquire_sql

- open

- prepare to commit

- prepare

- put data

- set_sql

- whenever

The syntax of EXECUTE IMMEDIATE is:

```
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 executing a dynamic SELECT statement.

Prepare and Execute Statements

The PREPARE statement tells the DBMS Server 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 within a transaction by issuing the EXECUTE statement and specifying the statement name. This method improves performance if your program must execute the same statement many times in a transaction. When a transaction is committed, all statements that were prepared during the transaction are discarded.

The following SQL statements cannot be prepared:

- call
- close
- connect
- declare
- disconnect
- enddata
- execute immediate
- execute fetch
- get data
- get dbevent
- include
- inquire_sql
- open
- prepare to commit
- prepare
- put data
- set
- set_sql
- whenever

The syntax of the PREPARE statement is:

```
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, with a maximum length of 32 characters. 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:

```
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 (?); these elements must be filled in (by the USING clause) when the statement is executed. For more information see Prepare in the chapter "Using SQL Statements."

Describe Statement

The DESCRIBE statement describes a prepared SQL statement into a program descriptor (SQLDA), which allows 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 syntax for the DESCRIBE statement is as follows:

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

For more information about the describe statement, see Describe Statement and SQLDA (see page 188) and Preparing and Describing Select Statement (see page 200).

Execute a Dynamic Non-select 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, execute it 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 indicate whether the statement was a select. For more information, see Executing a Dynamic Select Statement (see page 196).

Using Execute Immediate to Execute a Non-select Statement

EXECUTE IMMEDIATE executes an SQL statement specified using a string literal or host language variable. Use this statement to execute all but a few of the SQL statements; the exceptions are listed in Execute Immediate Statement (see page 190).

For non-select statements, the syntax of EXECUTE IMMEDIATE 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 table employee';
```

The following example reads SQL statements from a file into a host string variable 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 the statement parameters (such as an employee name or number) change at runtime, EXECUTE IMMEDIATE is not needed. A value can be replaced with a host language variable. For instance, the following example increases the salaries of employees whose employee numbers are read from a file:

```
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;
```

Preparing and Executing a Non-select Statement

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

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

```
read SQL 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 following example creates a table whose name is the same as the user name, and inserts a set of rows with fixed-typed parameters (the user's children) into the table:

```
get user name from terminal;
buffer = 'create table ' + user_name +
     '(child char(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;
```

A list of statements that cannot be executed using PREPARE and EXECUTE can be found in Prepare and Execute Statements (see page 192).

Execute a Dynamic Select Statement

To execute a dynamic SELECT statement, use one of the following methods:

- If your program knows the data types of the SELECT statement result columns, use the EXECUTE IMMEDIATE statement with the INTO clause to execute the select. EXECUTE IMMEDIATE defines a select loop to process the retrieved rows.

- If your program does not know the data types of the SELECT statement result columns, use the EXECUTE IMMEDIATE statement with the USING clause to execute the select.

- If your program does not know the data types of the SELECT statement result columns, declare a cursor for the prepared SELECT statement and use the cursor to retrieve the results.

The EXECUTE IMMEDIATE option allows you to define a select loop to process the results of the select. Select loops do not allow the program to issue any other SQL statements while the loop is open. If the program must access the database while processing rows, use the cursor option.

Details about these options are found in When Result Column Data Types Are Known (see page 198) and When Result Column Data Types Are Unknown (see page 199).

To determine whether a statement is a select, use the prepare and describe statements. A repeated SELECT statement cannot be prepared.

The following code demonstrates the use of the PREPARE and DESCRIBE statements to execute random statements and print results. This example uses cursors to retrieve rows if the statement is a select.

```
statement_buffer = ' ';
loop while reading statement_buffer from terminal
     exec sql prepare s1 from :statement_buffer;
     exec sql describe s1 into :rdescriptor;

     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
                    :rdescriptor;
               if (sqlca.sqlcode not equal 100) then
                    print the row using
                    rdescriptor;
               end if;

          end loop;

          free result variables from rdescriptor;

          exec sql close c1;

     end if;

     process sqlca for status;

end loop;
```

Unknown Result Column Data Types

For some dynamic SELECT statements, the program knows the data types of the resulting columns and, consequently, the data types of the result variables used to store the column values. If the program has this information, the program can use the EXECUTE IMMEDIATE statement with the into clause to execute the SELECT statement.

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 successfully enter two passwords before continuing. The first password entered is 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:

```
...
prompt for table_password and value_password
select_stmt = 'select column1 from ' +
    table_password;
exec sql execute immediate :select_stmt
    into :result_password;
if (sqlstate < 0) or (value_password <>
    result_password) then
    print     'Password authorization failure'
endif
...
```

Because the application developer knows the data type of the column in the password table (although not which password table is 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:

```
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 language 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 SQL statements for the duration of the loop.

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

How Unknown Result Column Data Types are Handled

In most instances, when a dynamically defined SELECT statement is executed, the program does not know in advance the number or types of result columns. To provide this information to the program, first prepare and 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 allocate (or reference) dynamically 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 results storage area (one row at a time), and close the cursor.

- The program can use EXECUTE IMMEDIATE. EXECUTE IMMEDIATE defines a select loop to process the returned rows. If the select returns only one row, it is not necessary to use a select loop.

Prepare and Describe Select Statements

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 192) in this chapter.

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; however, it is also possible to describe other statements. (When a non-select statement is described, the only information returned to the program is that the statement was not a SELECT statement.) The syntax of the DESCRIBE statement is:

```
exec sql describe statement_name into|using descriptor_name;
```

When a SELECT statement is described, information about each result column is returned to an sqlvar element. (For information about sqlvar elements, see Structure of the SQLDA (see page 186).) This is a one-to-one correspondence: the information in one sqlvar element corresponds to one result column. Therefore, 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 example shows a typical DESCRIBE statement and the surrounding host program code. The program assumes that 20 sqlvar elements are 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;
```

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). Your program must also allocate the space for these variables.

For example, if you create the table object as follows:

```
exec sql create table object
    (o_id           integer not null,
     o_desc         char(100) not null,
     o_price        money not null,
     o_sold         date);
```

and describe the following dynamic query:

```
exec sql prepare s1 from 'select * from object';
exec sql describe s1 into sqlda;
```

The SQLDA descriptor results are as follows:

sqld	4 (columns)		
sqlvar(1)	sqltype	=	30 (integer)
	sqllen	=	4
	sqlname	=	'o_id'
sqlvar(2)	sqltype	=	20 (char)
	sqllen	=	100
	sqlname	=	'o_desc'
sqlvar(3)	sqltype	=	5 (money)
	sqllen	=	0
	sqlname	=	'o_price'
sqlvar(4)	sqltype	=	-3 (nullable date)
	sqllen	=	0
	sqlname	=	'o_sold'

The value that the DESCRIBE statement returns in sqllen depends on the data type of the column being described, as listed in the following table:

Data Type	Contents of sqllen
char and varchar	Maximum length of the character string.
byte and byte varying	Maximum length of the binary data.
long varchar and long byte	Length of the string. If the length exceeds the maximum value of a 2-byte integer, sqllen is set to 0. Long varchar and long byte columns can contain up to 2 GB of data. To avoid buffer overflow, be sure to allocate a host language variable that is large enough to accommodate your data.
integer and float	Declared size of the numeric field.
date	0 (the program must use a 25-byte character string to retrieve or set date data).
money	0 (the program must use an 8-byte floating point variable to retrieve or set money data).
decimal	High byte contains precision, low byte contains scale.

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, modify sqltype and sqllen to be consistent with the program variables. After describing a SELECT statement, there is 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 contain the values in the result columns of the SELECT statement by pointing sqldata at a host language variable that 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 if the column is nullable), set sqltype to its positive value and sqlind to zero. Conversely, 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 preceding example, the program analyzes the results and modifies some of the types and lengths to correspond with the host language variables used by the program: the money data type is changed to float, and the date type to char. In addition, sqlind and sqldata are set to appropriate values. The values in the resulting sqlvar elements are:

sqlvar(1)	sqltype	=	30 (integer not nullable)
	sqllen	=	4
	sqldata	=	Address of 4-byte integer
	sqlind	=	0
	sqlname	=	'o_id'
sqlvar(2)	sqltype	=	20 (char not nullable)
	sqllen	=	100
	sqldata	=	Address of 100-byte character string
	sqlind	=	0
	sqlname	=	'o_desc'
sqlvar(3)	sqltype	=	31 (float not nullable, was money)
	sqllen	=	8 (was 0)
	sqldata	=	Address of 8-byte floating point
	sqlind	=	0
	sqlname	=	'o_price'
sqlvar(4)	sqltype	=	-20 (char nullable, was date)
	sqllen	=	25 (was 0)
	sqldata	=	Address of 25-byte character string
	sqlind	=	Address of 2-byte indicator variable
	sqlname	=	'o_sold'

Select Statement with Execute Immediate

A dynamic SELECT statement can be executed if the statement has been prepared and described with an EXECUTE IMMEDIATE statement that includes the USING clause. The USING clause tells the DBMS Server to place the values returned by the select into the variables pointed to by the elements of the SQLDA sqlvar array. If the select returns more than one row, a select loop can also be defined to process each row before another is returned.

The syntax of EXECUTE IMMEDIATE in this context is:

```
exec sql execute immediate select_statement
            using [descriptor] descriptor_name;
[exec sql begin;
            host_code
exec sql end;]
```

Within a select loop, no SQL statements other than an ENDSELECT can be issued. For non-looped selects, the DBMS Server expects the select to return a single row, and issues an error if more than one row is returned.

To illustrate this option, the following program example contains a dynamic select whose results 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 an error message
        ('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 Results Using Cursors

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, a cursor must be used 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 to fetch the result rows.

The syntax of the cursor declaration for a dynamically defined SELECT statement is:

```
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:

```
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, the result columns are copied into the result areas referenced by the descriptor.

The following example 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, it is executed. If the statement is a FETCH statement, 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 300 sqlvar elements;
sqlda.sqln = 300;

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 the results
                    (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 the sqlca and print the status;

     read statement_buffer from the terminal;

end loop;
```

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; for details about indicator variables, see Indicator Variables (see page 168). 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. 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;

}
```

Ingres 4GL Interface

Embedded SQL programs can be called from Ingres 4GL and OpenROAD. Using Ingres 4GL interface statements, 4GL data can be passed to the embedded SQL program, and the embedded SQL program can operate on 4GL objects. Ingres 4GL interface statements must be preceded with the keywords EXEC 4GL. For more information about these statements see the *Forms-based Application Development Tools User Guide* or the *OpenROAD Language Reference Guide*. The following statements are available:

- callframe
- callproc
- clear array
- describe
- get attribute
- get global constant
- get global variable
- getrow
- inquire_4gl
- insertrow
- removerow
- send userevent
- set attribute
- set global variable
- set_4gl
- setrow [deleted]

Chapter 6: Working with Transactions and Handling Errors

This chapter discusses the following SQL features:

- Transactions
- Two-phase commit
- Status information
- Error handling

Transactions

A *transaction* is one or more SQL statements processed as a single, indivisible database action.

If the transaction contains multiple statements, it is often called a *multi-statement transaction* (MST). By default, all transactions are multi-statement transactions.

How Transactions Work

A transaction can be performed by the SQL user, the program, or in some instances, by the DBMS Server itself.

The transaction performs the following actions:

- The transaction begins with the first SQL statement following a CONNECT, COMMIT, or ROLLBACK statement.
- 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 issues an implicit COMMIT statement.)

How Consistency is Maintained During Transactions

None of the database changes made by a transaction are visible to other users until the transaction is committed. In a multi-user environment, where more than one transaction is open concurrently, this behavior maintains database consistency. If two transactions are writing to the same database tables, the DBMS lock manager makes one transaction wait until the other is finished. A transaction that writes to the database locks pages in the tables that are affected, thus enforcing database consistency.

How Commit and Rollback Process Works

A transaction begins with the first statement after connection to the database or the first statement following a commit or rollback (including rollbacks performed by the DBMS). 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.

Individual Commits

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 DBMS 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 multi-statement transactions), issue the SET AUTOCOMMIT OFF statement.

How to Determine if You Are in a Transaction

To determine whether you are in a transaction, use the INQUIRE_SQL (see page 562) statement.

To find out if autocommit is on or off, use dbmsinfo.

Statements Used to Control Transactions

Three primary statements are used to control transactions, they are:

- COMMIT

- ROLLBACK

- SAVEPOINT

In some circumstances, the DBMS terminates a transaction with a rollback. For details, see Aborting Statements and Transactions (see page 224).

How Transaction Control Works

The COMMIT, ROLLBACK, and SAVEPOINT statements allow control of the effects of a transaction on the database as follows:

- The COMMIT statement makes the changes permanent.

- The ROLLBACK statement undoes all the changes made by the transaction.

- The ROLLBACK statement used with the SAVEPOINT statement allows a partial undo of the effects of a 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.

Savepoints on Multi Statement Transactions

In a multi-statement transaction, use ROLLBACK together with the SAVEPOINT statement to perform a partial transaction rollback. The SAVEPOINT statement establishes a marker in the transaction. If a rollback is subsequently issued, specify that the rollback only go back to the savepoint. All changes made prior to the savepoint are left in place; those made after the savepoint are undone. SAVEPOINT does not commit changes or release any locks, it simply establishes stopping points for use in partial rollbacks. For example:

```
...
insert into emp_table values (ename, edept);
update.....
savepoint first;
insert.....
delete.....
if error on delete
     rollback to first;
else if other errors
     rollback;
...
commit;
```

If an error occurs on the DELETE statement, the ROLLBACK TO FIRST statement directs the DBMS to back out all changes made after the savepoint was first created, in this case, only the changes made by the second INSERT statement. Processing resumes with the first statement that follows the ROLLBACK TO FIRST statement; the transaction is not terminated.

If an error occurs that makes it necessary to abort the entire transaction, the ROLLBACK statement that does not specify a savepoint causes the DBMS to back out the entire transaction. Depending on the design of the application, the program can either restart the transaction or continue with the next transaction.

An unlimited number of savepoints can be placed in a transaction. Roll back to the same savepoint is allowed any number of times within a transaction.

For a complete description of these statements, see the chapter "Using SQL Statements."

Interrupt Handling

When an operator interrupt occurs on the currently executing transaction, the Ingres transaction processing system response depends on the operating system used:

Windows: The Ingres transaction processing system recognizes the interrupt signal, Ctrl+C. When the user enters a Ctrl+C through a terminal monitor during transaction processing, the DBMS interrupts the current statement and rolls back any partial results of that statement. Additional use of Ctrl+C is ignored (unless an additional statement is added to the transaction). The transaction remains open until terminated by a COMMIT or ROLLBACK statement.

UNIX: The Ingres transaction processing system recognizes the interrupt signal Ctrl+C. When the user enters a Ctrl+C through a terminal monitor during transaction processing, the DBMS interrupts the current statement and rolls back any partial results of that statement. If there is no statement currently executing, Ctrl+C has no effect. Ctrl+C has no effect on the state of the transaction and does not cause any locks to be released.

VMS: The Ingres transaction processing system recognizes two interrupt signals, Ctrl+C and Ctrl+Y, when they are entered through a terminal monitor. When the user enters a Ctrl+C through a terminal monitor during transaction processing, the DBMS interrupts the current statement and rolls back any partial results of that statement. If there is no statement currently executing, Ctrl+C has no effect. Ctrl+C has no effect on the state of the transaction and does not cause any locks to be released.A Ctrl+Y character causes the DBMS to roll back a transaction in progress. The use of Ctrl+Y is strongly discouraged, as is the use of the VMS STOP command.

Abort Policy for Transactions and Statements

Transactions and statements can be aborted by any of the following entities:

- An application
- The DBMS

Applications can abort transactions or statements as a result of the following conditions:

- ROLLBACK statement
- Timeout (if set)

The DBMS aborts statements and transactions as a result of the following conditions:

- Deadlock
- Transaction log full
- Lock quota exceeded
- Error while executing a database statement

How to Direct the DBMS to Rollback an Entire Transaction or Statement

To direct the DBMS to rollback an entire transaction (or a single statement), use the SET SESSION WITH ON ERROR = ROLLBACK STATEMENT | TRANSACTION statement.

Note: The errors Deadlock, Transaction log full, and Lock quota exceeded always rollback the entire transaction regardless of the current on_error setting.

Effects of Aborted Transactions

When a statement or transaction is aborted (due to an application or the DBMS itself), the following occurs:

- 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.

When writing embedded SQL applications, your application must include logic for handling operator interrupts. By default, if the application is aborted during a transaction, the transaction is rolled back. This also applies to Ingres tools. For example, if you abort Query-By-Forms (QBF) while it is performing an update, the update is rolled back.

Two Phase Commit

Two phase commit is a mechanism that enables an application managing multiple connections to ensure that committal of a distributed transaction occurs in all concerned databases. This maintains database consistency and integrity.

Statements that Support Two Phase Commit

SQL provides the following two statements that support two phase commit functionality:

PREPARE TO COMMIT

The PREPARE TO COMMIT statement allows the coordinator application to poll each local DBMS to determine if the local DBMS is ready to commit the local transaction associated with the specified distributed transaction. Using this statement, the coordinator application can ensure that a distributed transaction is committed if and only if all of the local transactions that are part of the distributed transaction are committed. When the PREPARE TO COMMIT statement successfully completes, the local transaction is in a *willing commit* state.

CONNECT

The CONNECT statement, when specified with the distributed transaction ID, provides the means for a coordinator application to re-connect to a local DBMS, if the original connection was severed for any reason, for the purposes of committing or aborting a local transaction associated with the specified distributed transaction. When a local transaction is in the willing commit state, the coordinator application controls further processing of that transaction.

Both the PREPARE TO COMMIT and the CONNECT statements make use of the distributed transaction ID, an 8-byte integer that must be supplied by the coordinator application. The distributed transaction ID must be a unique number. The local DBMS returns an error to the coordinator application if a specified distributed transaction ID is not unique within the local DBMS.

VMS: Two phase commit is not supported for VMS cluster installations.

Coordinator Applications for a Two Phase Commit

To use a two phase commit, coordinator applications are used. The coordinator application is responsible for:

- Generating a unique distributed transaction ID for each distributed transaction, and passing this ID to each concerned local DBMS.

- Performing the necessary logging and recovery tasks to handle any failure occurring during the transaction processing. This includes logging the following information:

  - Distributed transaction IDs

  - The states of all the slave transactions

If the connection between a coordinator application and a local DBMS breaks while a distributed transaction is still open, the action taken by the local recovery process depends on the state of the local transaction associated with the distributed transaction:

- If the local transaction is not in a willing commit state, the local DBMS aborts the transaction.

- If the local transaction is in a willing commit state, the local transaction is not aborted until the connection is re-established by the coordinator application and the transaction is committed or rolled back.

If the connection between a coordinator application and a local DBMS breaks, use the CONNECT statement to re-establish the connection with the local DBMS and transaction. If the local DBMS has rolled back the local transaction associated with the distributed transaction, the DBMS returns an error statement indicating this when issuing the CONNECT statement. Otherwise, after reconnecting, a COMMIT or a ROLLBACK statement can be issued to close the transaction.

If a local DBMS encounters a log file full condition and the oldest transaction is a local transaction that is associated with a distributed transaction and is in the willing commit state, the local logging system does not abort the local transaction. Normally the logging system aborts the oldest transactions first. For details about transaction logging, see the *Database Administrator Guide*.

Manual Termination of a Distributed Transaction

To terminate a local transaction associated with a distributed transaction, use the logstat utility to obtain the local transaction ID of the transaction. With this ID, use the utility lartool to manually terminate the transaction.

Lartool is a simple command-line utility that allows the commit or rollback of the transaction identified by the local transaction ID.

For more information about the logstat and lartool utilities, see the *Database Administrator Guide.*

Example: Using Two-Phase Commit

The following is an example of a two-phase commit used in a banking application. It illustrates the use of the PREPARE TO COMMIT and CONNECT statements.

```
exec sql begin declare section;
    from_account    integer;
    to_account      integer;
    amount          integer;
    high            integer;
    low             integer;
    acc_number      integer;
    balance         integer;
exec sql end declare section;

define              SF_BRANCH 1
define              BK_BRANCH 2
define              BEFORE_WILLING_COMMIT 1
define              WILLING_COMMIT 2


exec sql whenever sqlerror stop;

/* Connect to the branch database in S.F */

exec sql connect annie session :SF_BRANCH;

Program assigns value to from_account, to_account, and amount

/* Begin a local transaction on S.F branch to
** update the balance in the from_account. */

exec sql update account
    set balance = balance - :amount
    where acc_number = :from_account;

/* Connect to the branch database in Berkeley. */

exec sql connect aaa session :BK_BRANCH;
```

```
/* Begin a local transaction on Berkeley branch
** to update the balance in the to_account. */

exec sql update account
    set balance = balance + :amount
    where acc_number = :to_account;

/* Ready to commit the fund transfer transaction. */
/* Switch to S.F branch to issue the prepare to
** commit statement. */

exec sql set_sql (session = :SF_BRANCH);

/* Store the transaction's state information in a
** file */

store_state_of_xact(SF_BRANCH,
    BEFORE_WILLING_COMMIT, high, low, "annie");

exec sql prepare to commit with highdxid = :high,
    lowdxid = :low;

/* Store the transaction's state information in a
** file */
store_state_of_xact(SF_BRANCH, WILLING_COMMIT,
    high, low, "annie");

/* Switch to Berkeley branch to issue the prepare
** to commit statement. */

exec sql set_sql (session = :BK_BRANCH);

/* Store the transaction's state information in a
** file */

store_state_of_xact(BK_BRANCH,
    BEFORE_WILLING_COMMIT, high, low, "annie");

exec sql prepare to commit with highdxid = :high,
    lowdxid = :low;

/* Store the transaction's state information in a
** file */

store_state_of_xact(BK_BRANCH, WILLING_COMMIT,
    high, low, "annie");

/* Both branches are ready to COMMIT; COMMIT the
** fund transfer transaction. */
```

```
/* Switch to S.F branch to commit the local
** transaction. */

exec sql set_sql (session = :SF_BRANCH);

exec sql commit;

/* Switch to Berkeley branch to commit the local
** transaction. */

exec sql set_sql (session = :BK_BRANCH);

exec sql commit;

/* Distributed transaction complete */
/* Switch to S.F branch to verify the data. */

exec sql set_sql (session = :SF_BRANCH);

exec sql select acc_number, balance
    into :acc_number, :balance
    from account;
exec sql begin;

    print (acc_number, balance);

exec sql end;

/* Switch to Berkeley branch to verify the data. */

exec sql set_sql (session = :BK_BRANCH);

exec sql select acc_number, balance
    into :acc_number, :balance
    from account;
exec sql begin;

    print (acc_number, balance);

exec sql end;

/* Exit the S.F database. */

exec sql set_sql (session = :SF_BRANCH);
exec sql disconnect;

/* Exit the Berkeley database. */
```

```
exec sql set_sql (session = :BK_BRANCH);
exec sql disconnect;
```

This portion of the example shows how the information logged in the procedure store_state_of_xact is used for recovery after a system failure at either branch.

The first part of the recovery process is to read the state of each transaction from information logged by store_state_of_xact. If either state is in BEFORE_WILLING_COMMIT, the program connects to the specific transaction in both databases and executes a rollback. Although the local DBMS can roll back the transaction, the recovery process reconnects to the specific transaction. This occurs because a PREPARE TO COMMIT has been sent, received, and acted upon, but a crash occurred before the acknowledgment was received by the coordinator application.

If both states are in WILLING_COMMIT, the program connects to the specific transactions and commits them:

```
exec sql begin declare section;
    high                integer;
    low                 integer;
    session1            integer;
    session2            integer;
    dbname1             character_string(25);
    dbname2             character_string(25);
exec sql end declare section;

/* Read information saved by store_state_of_xact */

read_from_file(address(session1),
    address(session2),
    address(dbname1), address(dbname2),
    address(high), address(low));

/* Assume that a global flag has been set to
** commit or rollback based on the information
** in the file */

if (flag = 'COMMIT') then
    exec sql connect :dbname1 session :session1
        with highdxid = :high, lowdxid = :low;
    exec sql commit;
    exec sql disconnect;

exec sql connect :dbname2 session :session2
        with highdxid = :high, lowdxid = :low;
    exec sql commit;
    exec sql disconnect;
```

```
else
     exec sql connect :dbname1 session :session1
          with highdxid = :high, lowdxid = :low;
     exec sql rollback;
     exec sql disconnect;

exec sql connect :dbname2 session :session2
          with highdxid = :high, lowdxid = :low;
     exec sql rollback;
     exec sql disconnect;

endif;
```

Status Information

The following functions enable an embedded SQL application program to obtain status information:

session_priv

Returns session privilege information.

dbmsinfo

Returns information about the current session.

inquire_sql

Returns information about the last database statement that was executed.

SQLCA (SQL Communications Area)

Returns status and error information about the last SQL statement that was executed.

SQLCODE and SQLSTATE

These are the stand-alone variables in which the DBMS returns status information about the last SQL statement that was executed.

session_priv Function

The session_priv function determines whether the current session has a subject privilege, or can request it. The input is any single subject privilege name, such as operator, specified as a text string in upper or lower case.

The following values are returned:

Y

Indicates session has privilege.

N

Indicates session does not have the privilege.

R

Indicates the session can request the privilege and make it active.

The following example checks whether the current session has auditor privilege:

```
SELECT session_priv('AUDITOR');
```

dbmsinfo Function

A *dbmsinfo function* is a SQL function that returns a string containing information about the current session. Use this function in a terminal monitor or in an embedded SQL application.

The dbmsinfo function is used in a SELECT statement as follows:

```
select dbmsinfo ('request_name')
```

For example, to find out which runtime of Ingres you are using, enter:

```
select dbmsinfo('_version');
```

Use the dbmsinfo function in WHERE clauses in SELECT statements. For example:

```
exec sql select dept from employee
    where ename=dbmsinfo('username');
```

Valid Request Names for dbmsinfo Function

The following table lists valid *request_names* that can be used with the dbmsinfo function:

Request Name	Description
autocommit_state	Returns 1 if autocommit is on and 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.
_bio_cnt	Returns the number of I/Os to and from the front-end client (application) that created your session.
charset	Returns the value of II_CHARSETxx, the character set setting for the installation.
collation	Returns the collating sequence defined for the database associated with the current session. This returns blanks if the database is using the collating sequence of the machine's native character set, such as ASCII or EBCDIC.
connect_time_limit	Returns the session connect time limit or -1 if there is no connect time limit.
create_procedure	Returns Y if the session has create_procedure privileges in the database or N if the session does not.
create_table	Returns Y if the session has create_table privileges in the database or N if the session does not.
_cpu_ms	Returns the CPU time for your session in milliseconds.
current_priv_mask	Returns the decimal number representing a mask of internal privilege bits currently enabled for the user.
cursor_default_mode	Returns the default mode for the cursor.
cursor_update_mode	Returns the mode of the current user.
database	Returns the database name.
datatype_major_level	Returns -2147483648 unless the user datatypes are in use.
datatype_minor_level	Returns 0 unless the user datatypes are in use.
dba	Returns the user name of the database owner.

Request Name	Description
db_admin	Returns Y if the session has db_admin privileges, and N if the session does not have db_admin privileges.
db_cluster_node	Returns the machine you are connected to. Valid even if not clustered.
db_count	Returns the number of distinct databases opened.
db_real_user_case	Returns lower, upper, or mixed.
dbms_bio	Returns the cumulative non-disk I/O's performed by the server hosting session.
dbms_cpu	Returns the cumulative CPU time for the DBMS Server, in milliseconds, for all connected sessions.
dbms_dio	Returns the cumulative disk I/O's performed by the server hosting session.
db_delimited_case	Returns LOWER if delimited identifiers are translated to lower case, UPPER if delimited identifiers are translated to upper case, or MIXED if the case of delimited identifiers is not translated.
db_name_case	Returns LOWER if regular identifiers are translated to lower case or UPPER if regular identifiers are translated to upper case.
db_privileges	Returns a decimal integer which represents a bit mask of "Subject" privileges.
db_tran_id	Returns the 64 bit internal transaction ID as two decimal numbers.
_dio_cnt	Returns the number of disk I/O requests for your session.
_et_sec	Returns the elapsed time since the start of your session, in seconds.
flatten_aggregate	Returns Y if the DBMS Server is configured to flatten queries involving aggregate subselects; otherwise, returns N. (Query flattening options are specified when the DBMS Server is started.)
flatten_singleton	Returns Y if the DBMS Server is configured to flatten queries involving singleton subselects; otherwise, returns N. (Query flattening options are specified when the DBMS Server is started.)
group	Returns the group identifier of the session or blanks if no group identifier is in effect.

Request Name	Description
idle_time_limit	Returns the session idle time limit or -1 if there is no idle time limit.
ima_server	Equivalent to IMA registration exp.gwf.gwm.glb.this_server, which returns the listen address of the attached server.
ima_session	Returns the internal session ID in decimal format.
ima_vnode	Equivalent to IMA registration exp.gwf.gwm.glb.def_vnode configuration value if set for the connected server. If not set, defaults to the local host name.
initial_user	Returns the user identifier in effect at the start of the session.
language	Returns the language used in the current session to display messages and prompts.
lockmode	Returns Y if the user possesses lockmode database privileges or N if the user lacks these privileges.
lp64	Returns Y if 64 bit pointers are in use, or N if 32 bit addresses are used.
maxconnect	Returns the current connect time limit, as set by the set maxconnect statement, or the initial value if no connect time limit has been set.
maxcost	Returns the value specified in the last set maxcost statement. If no previous set maxcost statement was issued or if set nomaxcost was specified last, this returns the same value as the request name query_io_limit.
maxcpu	Returns the value specified in the last set maxcpu statement. If no previous set maxcpu statement was issued or if set nomaxcpu was specified last, this returns the same value as the request name query_io_limit.
maxidle	Returns the current idle time limit, as set with the set maxidle statement, or the initial value if no idle time limit has been set.
maxio	Returns the value specified in the last set maxio statement. If no previous set maxio statement was issued or if set nomaxio was specified last, this returns the same value as the request name query_io_limit.
maxquery	Same as maxio.

Request Name	Description
maxrow	Returns the value specified in the last set maxrow statement. If no previous set maxrow statement was issued or if set nomaxrow was specified last, this returns the same value as the request name query_row_limit.
maxpage	Returns the value specified in the last set maxpage statement. If no previous set maxpage statement was issued or if set nomaxpage was specified last, this returns the same value as the request name query_io_limit.
max_page_size	Returns the size of the largest enable page cache in bytes.
max_priv_mask	Returns the decimal number representing a mask of internal privilege bits for which privileges the user might possess if all his/her privileges were enabled.
max_tup_len	Returns the max width for a non-segmented tuple. This depends on max_page_size.
on_error_state	Returns the current setting for transaction error handling: rollback transaction or rollback statement. To set transaction error handling, use the set session with on_error statement.
page_size_2k	Returns Y if this size cache is enabled.
page_size_4k	Returns Y if this size cache is enabled.
page_size_8k	Returns Y if this size cache is enabled.
page_size_16k	Returns Y if this size cache is enabled.
page_size_32k	Returns Y if this size cache is enabled.
page_size_64k	Returns Y if this size cache is enabled.
pagetype_v1	Returns Y if this page type is supported.
pagetype_v2	Returns Y if this page type is supported.
pagetype_v3	Returns Y if this page type is supported.
_pfault_cnt	Returns the number of page faults for the server.
query_cost_limit	Returns the session value for query_io_limit or -1 if no limit is defined for the session.
query_cpu_limit	Returns the session value for query_io_limit or -1 if no limit is defined for the session.
query_flatten	Returns Y if the query flattening is in effect or N if the query flattening is not in effect.

Request Name	Description
query_io_limit	Returns the session value for query_io_limit or -1 if no limit is defined for the session.
query_language	Returns sql or quel.
query_page_limit	Returns the session value for query_io_limit or -1 if no limit is defined for the session.
query_row_limit	Returns the session value for query_row_limit or -1 if no limit is defined for the session.
role	Returns the role identifier of the session or blanks if no role identifier is in effect.
security_audit_log	Returns the name of the current security auditing log file if it is enabled and the user has maintain_audit privileges, otherwise, remains blank.
security_audit_state	Returns the current Ingres security audit state. The following values are returned: (blank) – Ingres security auditing is not available STOP – Security auditing is stopped SUSPEND – Security auditing is suspended ACTIVE – Security auditing is active
security_priv	Returns Y if the effective user has the security privilege or N if the effective user does not have the security privilege.
select_syscat	Returns Y if the session has select_syscat privilege or N if the session does not have select_syscat privilege.
session_id	Returns the internal session identifier in hexadecimal.
session_priority	Returns the current session priority.
session_priority_limit	Returns the highest session priority that can be set, or an empty string if no session priority limit applies.
session_user	Returns the current effective user ID of the session.
system_user	Returns the system user ID.
table_statistics	Returns Y if the session has table_statistics privilege, or N if the session does not have table_statistics privilege.
terminal	Returns the terminal address.

Request Name	Description
transaction_state	Returns 1 if currently in a transaction and returns 0 if not currently in a transaction.
tup_len_2k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
tup_len_4k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
tup_len_8k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
tup_len_16k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
tup_len_32k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
tup_len_64k	Returns the largest tuple for this page size or returns 0 if the cache for this page size is turned off.
ucollation	Returns Unicode collation. The default is "udefault."
unicode_level	Returns the Unicode level for this database. Returns 0 if there is no Unicode level, otherwise returns 1 if the database is created with -n flag.
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.
update_rowcnt	Returns qualified if inquire_sql(rowcount) returns the number of rows that qualified for change by the last query, or changed if inquire_sql(rowcount) returns the number of rows that were actually changed by the last query.
update_syscat	Returns Y if the effective user is allowed to update system catalogs or N if the effective user is not allowed to update system catalogs.
username	Returns the user name of the user currently running Ingres.
_version	Returns the Ingres runtime number.

inquire_sql Function

The inquire_sql function returns information about the status of the last SQL database statement issued by an application.

Information Provided by the inquire_sql Function

The INQUIRE_SQL function can provide the following information on the occurrence of an error:

- Return the error number and the complete error text.

- Retrieve the message number and text from a message statement that was executed inside a database procedure.

- Determine if your session is returning local or generic errors, if a transaction is open, or what session is currently active (useful in multiple-session applications).

The INQUIRE_SQL function does not return status information about forms statements. To obtain information about the results of forms statements, use the INQUIRE_FRS statement.

For a complete list and description of available status information, see Inquire_sql (see page 562).

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 SQL database statements. Forms statements do not affect these variables. Because each embedded SQL statement has the potential to change values in the SQLCA, the application must perform any checking and consequent processing required to deal with a status condition immediately after the statement in question. If it does not, the next executed SQL statement changes 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*.

Variables that Compose SQLCA

The following list describes the variables that compose the SQLCA (not all of the variables are currently used):

SQLCA Variable	Description
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 SQL return code. Its value falls into one of three categories: = 0—The statement executed successfully (though there have been warning messages: check sqlwarn0). < 0—An error occurred. The value of sqlcode is the negative value of the error number returned to errorno. A negative value sets the sqlerror condition of the WHENEVER statement. > 0—The statement executed successfully but an exception condition occurred. The following values are returned: 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. 700- Indicates that a message statement in a database procedure has just executed, setting the sqlmessage condition of the WHENEVER statement. 710- Indicates that a database event was raised.

SQLCA Variable	Description
sqlerrm	A varying length character string variable with 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 contains 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. **sqlerrmc**—A 70-byte fixed length character string buffer.
sqlerrp	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 the values returned in sqlerrd(1), see Types of Errors (see page 247). Sqlerrd(3) indicates the number of rows processed by a DELETE, FETCH, INSERT, SELECT, UPDATE, COPY, MODIFY, 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 the subscript, 2, to select the third array element.

SQLCA Variable	Description
sqlwarn0- sqlwarn7	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 hostvariable. 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.

SQLCODE and SQLSTATE

SQLCODE and SQLSTATE are variables in which the DBMS returns ANSI/ISO Entry-92-compliant status codes indicating the results of the last SQL statement that was executed.

SQLCODE Variable

SQLCODE is an integer variable in which the DBMS returns the status of the last SQL statement executed. For details about the requirements for declaring the SQLCODE variable in embedded programs, see the *Embedded SQL Companion Guide*.

Note: The ANSI Entry SQL-92 specification describes SQLCODE as a deprecated feature, and recommends using the SQLSTATE variable.

Values Returned by SQLCODE

The values returned in the standalone SQLCODE variable are the same as those returned in the sqlcode member of the SQLCA structure. The value of SQLCODE is meaningful only in the context of a session.

The values returned in SQLCODE are listed in the following table:

Value	Description
0	Successful completion.
+100	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.
+700	A message statement in a database procedure has just executed, setting the sqlmessage condition of the WHENEVER statement.
+710	A database event was raised.
Negative Value	An error occurred. The value of SQLCODE is the negative value of the error number returned to errorno. For information on errorno, see Error Checking Using Inquire Statements (see page 254). A negative value sets the sqlerror condition of the WHENEVER statement.

SQLSTATE Variable

The SQLSTATE variable is a 5-character string in which the DBMS Server 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*.

Note: If queries are executed while connected (through an Enterprise Access product) to a DBMS server 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, use one of the other error-checking methods.

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 example illustrates 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!'
```

For a list mapping Ingres generic errors to SQLSTATE values, see the appendix "SQLSTATE Values and Generic Error Codes."

Error Handling

The following sections describe how the DBMS returns error information.

Types of Error Codes

Three types of error codes are returned to applications:

Local errors

Local errors are error codes specific to the DBMS.

Generic errors

Generic errors are a set of error codes that are mapped to both the DBMS and to error codes returned through Enterprise Access products from other relational and non-relational databases. Generic errors allow portable applications to be written.

ANSI/ISO error codes

SQLSTATE and SQLCODE are ANSI/ISO-compliant error code variables. (SQLCODE is supported by Ingres but designated by ANSI/ISO Entry SQL-92 as a deprecated feature. SQLSTATE is the ANSI/ISO Entry SQL-92-compliant method for returning errors.)

By default, the DBMS returns generic and local errors as follows:

Generic errors

Returned in sqlcode (an SQLCA variable) as a negative value. (Also in the SQLCODE standalone variable.)

Returned when your application issues the INQUIRE_SQL(ERRORNO) statement.

Local errors

Returned in sqlerrd(1), the first element of the SQLCA 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 Ingres error message consists of an error code and the accompanying error message text.

All Ingres 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).

If the error is a local error, the two letters following E_ indicate which Ingres facility issued the error. If the error is a generic error number, the two letters are GE. The hexadecimal error code is unique for each error.

Local error codes are stored in $II_SYSTEM/ingres/files/english/messages/message.text

Generic error codes are stored in $II_system/ingres/files/generr.h

Display of Error Messages

When working in one of the forms-based user interfaces (such as Query-By-Forms (QBF) or the forms-based Terminal Monitor), error messages appear on a single line across the bottom of your terminal screen. The text appears first, followed by the error code. If the text is longer than one line, press the Help key to see the rest of the message. To clear the error message from the screen, press the Return key.

When not working in a forms-based user interface, the DBMS displays the error code followed by the entire message text.

If an SQLCA is included in an embedded SQL application, automatic display of error messages is disabled. Program code that displays errors must be provided.

Error Handling in Embedded Applications

SQL provides a variety of tools for trapping and handling errors in embedded SQL applications, including:

- SQLCA
- SQLSTATE
- 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 SQL database statements. (The status of forms statements is not returned 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 a value of 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 either +100, indicating that no rows were processed by a DELETE, FETCH, INSERT, UPDATE, MODIFY, COPY, or CREATE TABLE...AS statement, or +700, indicating that a MESSAGE statement inside a database procedure has just executed.

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, the message is truncated when it is assigned to sqlerrm. To retrieve the full message, use the INQUIRE_SQL statement. 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. For a complete listing of these variables, see the table titled SQLCA Variables.

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, the sqlerror condition of the WHENEVER statement is true and any action specified for that condition is performed. For details, see Trapping Errors Using Whenever Statement (see page 250).

The SQLCA variables can also be accessed directly. For information about using the SQLCA in an application, see the *Embedded SQL Companion Guide*.

SQLSTATE

SQLSTATE is a variable in which the DBMS returns error codes as prescribed by the ANSI/ISO Entry SQL-92 standard. For a list of the values returned in SQLSTATE and the corresponding generic error, see the appendix "SQLSTATE Values and Generic Error Codes."

Error Trapping Using Whenever Statement

The WHENEVER statement specifies a particular action to be performed whenever a particular condition is true. Because conditions are set to true by values in the SQLCA sqlcode, the WHENEVER statement responds only to errors generated by embedded SQL database statements. Forms statements do not set sqlcode.

The following conditions indicate errors or warnings:

Warnings/Error	Explanation
sqlwarning	Indicates that the executed SQL 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.

For a complete discussion of all the conditions, see Whenever in the chapter "Using SQL Statements."

The actions that can be specified for these conditions are listed in the following table:

Action	Explanation
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, a standard sqlprint procedure is called. This procedure prints the error or warning message and continues with the next statement. A database procedure cannot be specified.

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;
```

The DBMS traps errors for all database statements in your program that (physically) follow the WHENEVER statement, to the "myhandler" procedure. A WHENEVER statement does not affect the statements that physically precede it.

The following diagram illustrates the scope of the WHENEVER statement:

Physical Source Code File

```
maincode:

whenever        condition       ACTIONA

  ──────────────────
  ──────────────────
call subroutinea                          ←── Scope of
  ──────────────────                           ACTIONA
  ──────────────────

whenever        condition       ACTIONB

  ──────────────────
  ──────────────────
end program                               ←── Scope of
                                               ACTIONB
subroutinea

whenever        condition       ACTIONC

  ──────────────────
  ──────────────────
return

subroutineb                               ←── Scope of
  ──────────────────                           ACTIONC
  ──────────────────
return
```
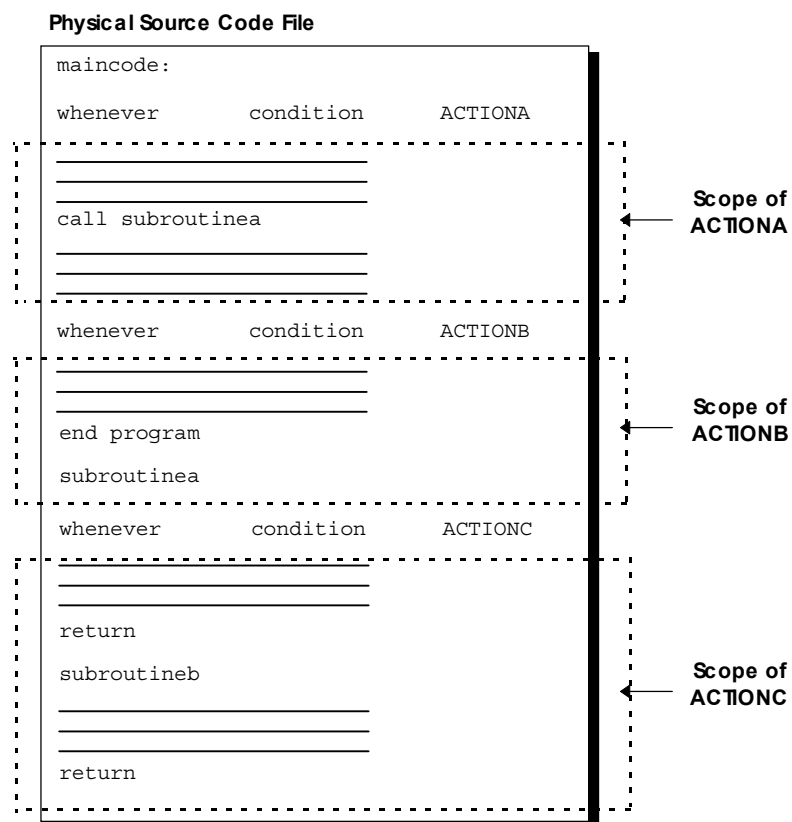
If your program includes an SQLCA, error and database procedure messages are not displayed unless your application issues a WHENEVER...SQLPRINT statement, or II_EMBED_SET is set to sqlprint. For details about II_EMBED_SET, see the *System Administrator Guide*.

Define Error Handling Functions

An error handling function can be defined to be called when SQL errors occur. To do this, perform the following:

1. Write the error handling routine and link it into your embedded SQL application.

2. 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 that was created. Do not declare *error_routine* in an SQL 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 SQL errors are trapped to your routine until error trapping is disabled (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 or switch to another session.

To obtain error information, your error-handling routine must issue the INQUIRE_SQL statement.

Other Types of Handlers

In addition to error-handling routines, routines can be defined that enable embedded SQL applications to trap the following:

Event notifications:

To enable or disable an event-handling routine, issue the following SET_SQL statement:

```
exec sql set_sql(dbeventhandler = event_routine | 0)
```

When an event notification is received by your application, the specified routine is automatically called. To obtain the event notification information, the event handler routine must use the INQUIRE_SQL statement.

Messages from database procedures:

To enable or disable a message handling routine, issue the following SET_SQL statement:

```
exec sql set_sql(messagehandler = message_routine | 0)
```

The message handler routine traps all messages from database procedures, including messages from procedures that are executed when rules are fired.

Specify the routine as a function pointer. For more information about specifying function pointers, see the *Embedded SQL Companion Guide*.

Error Checking Using Inquire Statements

The inquire statements that can be used to perform error checking are:

INQUIRE_SQL and INQUIRE_FRS

Both statements return error numbers and messages using the constants errorno and errortext. INQUIRE_SQL returns the error number and text for the last executed SQL database statement. INQUIRE_FRS returns the same 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. By default, INQUIRE_SQL(ERRORNO) returns a generic error number, but the SET_SQL statement can be used to specify that local errors are returned. For a discussion of local and generic errors, see Types of Errors (see page 247).

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.

Program Termination When Errors Occur

The SET_SQL(PROGRAMQUIT) statement specifies how an embedded SQL application handles the following types of errors:

- Attempt to execute a query when not connected to a database

- DBMS server failure

- Communications service failure

By default, when these types of errors occur, the DBMS 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, the DBMS issues an error and rolls back open transactions and disconnects all open sessions. To disable aborting and restore the 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.

PROGRAMQUIT can also be specified by using II_EMBED_SET. (For details about II_EMBED_SET, see the *System Administrator Guide.)*

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 (execution continues on any of the errors) or 1 if PROGRAMQUIT is set (the application exits after these errors).

Deadlocks

A 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.

Example: Handling Deadlocks When Transactions Do Not Contain Cursors

The following example 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 */
exec sql commit;
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 = deadlock)
        or (sqlca.sqlcode = forceabort) 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:
```

Example: Handling Deadlocks with One Cursor

The following example assumes your transactions contain a single cursor:

```
exec sql whenever not found continue;
exec sql whenever sqlwarning continue;
exec sql whenever sqlerror goto err;

exec sql declare c1 cursor for ...

exec sql commit;
start:
    exec sql open c1;
    while more rows loop
        exec sql fetch c1 into ...
        if (sqlca.sqlcode = zero_rows) then
            exec sql close c1;
            exec sql commit;
            goto end;
        end if;

exec sql insert into ...
        exec sql update ...
    exec sql select ...

end loop;
err:
    exec sql whenever sqlerror call sqlprint;
    if (sqlca.sqlcode = deadlock)
    or (sqlca.sqlcode = forceabort) 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:
```

Example: Handling Deadlocks with Two Cursors

The following example assumes your transactions contains two cursors (two cursors with a master/detail relationship):

```
exec sql whenever not found continue;
exec sql whenever sqlwarning continue;
exec sql whenever sqlerror goto err;

exec sql declare master cursor for ...
     exec sql declare detail cursor for ...

exec sql commit;

start:
     exec sql open master;
     while more master rows loop
          exec sql fetch master into ...
          if (sqlca.sqlcode = zero_rows) 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 = zero_rows) 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 */
err:
     exec sql whenever sqlerror call sqlprint;
     if (sqlca.sqlcode = deadlock)
     or (sqlca.sqlcode = forceabort) 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:
```

Chapter 7: Understanding Database Procedures, Sessions, and Events

This chapter discusses the following features of SQL:

- Database procedures

- Rules

- Multiple session connections

- Database events

How Database Procedures Are Created, Invoked, and Executed

A *database procedure* is a named routine composed of SQL statements stored in a database.

Database procedures are created using the CREATE PROCEDURE statement and dropped using the DROP PROCEDURE statement.

Database procedures can be called or invoked in the following ways:

- From an embedded SQL program

- From interactive SQL

- From a 4GL program

- By rules

A database procedure query execution plan is created at the time the procedure is created. If objects named in the procedure are modified in a way that invalidates the query execution plan, the DBMS Server recreates the query execution plan the next time the procedure is invoked.

Benefits of Database Procedures

Database procedures provide the following benefits:

- Enhanced performance

- Reduced amount of communication between an application and the DBMS Server. The DBMS Server retains the query execution plan for a database procedure, reducing execution time.

- Control over access to data. The DBA can use the GRANT statement to enable a user to execute a procedure even if the user does not have permission to directly access the tables referenced in the procedure.

- Reusability and reduced coding time. The same procedure can be used in many applications.

- The ability to enforce integrity constraints (when used in conjunction with rules).

Contents of Database Procedures

A database procedure can include the following entities:

- Local variable declarations

- Data manipulation statements such as SELECT or INSERT

- Control flow statements such as IF, FOR, and WHILE

- Status statements, such as MESSAGE, RETURN, RETURN ROW, and RAISE ERROR

Database procedures can be executed by running the CREATE PROCEDURE statement from a terminal monitor or from within an embedded SQL program.

The DBMS Server resolves all references to database objects in a database procedure at the time the procedure is created. For this reason, all referenced objects must exist at the time the procedure is created. If, at the time it is created, a procedure refers to a DBA-owned table, the procedure always uses that table, even if a user that owns a table with the same name executes the procedure.

Permissions on Database Procedures

A procedure is owned by the user who creates it or by the group or role specified in the CREATE PROCEDURE statement. A procedure can be executed by the owner and by any user, group, or role to whom the owner has granted execute permissions. Users, groups, and roles to which the owner has granted execute permission with grant option can grant execute permission to other users.

Although a user can create a private procedure that accesses public tables, the user must have all required permissions on those tables to be able to execute the procedure.

Methods of Executing Procedures

Database procedures can be executed in the following ways:

- By using the EXECUTE PROCEDURE statement in an embedded SQL application. This statement executes a specified procedure and passes parameter values to the procedure. To specify different parameter lists at runtime, use the dynamic version of the EXECUTE PROCEDURE statement. To execute a procedure owned by a user other than the effective user of the session, specify the procedure name using the *schema*.*procedure_name* syntax.

- When a rule is fired. Indirectly executed procedures must handle errors and messages differently than directly executed procedures. A procedure that is executed as the result of a rule can execute statements that trigger other rules, and so on.

- From interactive SQL, from within another database procedure, or by using the dynamic SQL EXECUTE IMMEDIATE statement. A procedure cannot be executed using the dynamic SQL PREPARE and EXECUTE statements. For details about executing a database procedure, see Execute Procedure in the chapter "SQL Statements." For information about using database procedures with rules, see Rules (see page 268).

All referenced objects must exist at the time the procedure is executed. Between the time of creation and the time of execution, objects such as tables and columns can be modified, reordered, or dropped and recreated without affecting the procedure definition. However, if an object is redefined in a way that invalidates the procedure definition, drop and recreate the procedure.

How Parameters Are Passed in Database Procedures

By default, the EXECUTE PROCEDURE statement passes parameters to a database procedure by value. To pass a value by reference, use the BYREF option. If a parameter is passed by reference, the called database procedure can change the contents of the variable, and the change is visible to the calling program.

In addition to the BYREF option for passing parameters from an invoking application, parameters can be passed in one of three declared modes: IN, OUT, and INOUT. IN is the default parameter mode and is equivalent to the by value mode described above. It cannot return values to the calling procedure, rule, or application. The OUT and INOUT modes allow the passing of possibly modified parameter values back to the calling database procedure or triggering operation, for example, allowing the application to modify a column value in a row before it is inserted or updated.

Row Producing Procedures

A row producing procedure is a database procedure that is capable of returning 0 or more rows to its caller.

Format of Row Producing Procedures

The format of row producing procedures is defined by the RESULT ROW clause in the procedure definition. The value returned in each "column" of a result row is specified in a RETURN ROW statement and can be a local variable or parameter of the procedure, or any expression involving constants, local variables and parameters. The local variables must contain data retrieved in the procedure by a SELECT statement. Multiple result rows must be returned to the caller using the For-loop that retrieves data from a SELECT statement.

Row producing procedures must only be called directly from an embedded SQL host program (not using dynamic SQL, a terminal monitor, or by nesting a call in another database procedure). However, the host program must include a Begin/End block to process the rows as they are returned from the procedure. This block functions much the same as the "select block" used with embedded SELECT statements.

Effects of Errors in Database Procedures

When an error occurs in a database procedure, the behavior of the DBMS Server depends on whether the procedure was invoked by a rule or executed directly (using EXECUTE PROCEDURE).

If the procedure was invoked by a rule, an error has the following effects:

- The procedure is terminated.

- Those statements in the procedure which have been executed are rolled back.

- The statement that fired the rule is rolled back.

If the procedure was executed directly, an error has the following effects:

- All statements in the procedure up to the point of the error are rolled back.

- The procedure continues execution with the statement following the statement that caused the error.

- Parameters passed by reference are not updated.

In both instances, the error is returned to the application in SQLSTATE, SQLCODE, and ERRORNO. In the case of the directly executed procedure, an error number is also returned to iierrornumber, a built-in variable available only in database procedures for error handling.

Note: A fatal error, such as dividing by zero, cannot be trapped in a database procedure and will cause the procedure to fail. For example, executing the following database procedure, in which the divisor variable is 0, will result in program failure:

```
...
update a_test set an_integer=int4(an_integer/:divisor);
select iierrornumber, iirowcount into :enumber, :rowcount;
if enumber=0 then
if rowcount=0 then
...
```

iierrornumber and iirowcount Variables

The iierrornumber and iirowcount variables, in conjunction with the RAISE ERROR statement, handle errors in database procedures.

The iirowcount variable contains the number of rows affected by the last executed SQL statement. The iierrornumber variable contains the error number (if any) associated with the execution of a database procedure statement.

Because both iierrornumber and iirowcount reflect the results of the preceding query, beware of inadvertently resetting the value of one when checking the other.

The following example from a database procedure illustrates this error:

```
...

update emp set ...

/* The following statement resets iierrornumber, which will reflect the results
of the second statement and not the first, as desired. */

/* wrong way to check iirowcount */

rcount = iirowcount;

/* The error number reflects the results of the preceding assignment, not the
update statement */

enumber = iierrornumber;
```

The following example illustrates the correct way to check iierrornumber and iirowcount: select both values into variables, and then check the contents of the variables (because iierrornumber and iirowcount is reset to reflect the results of the SELECT statement).

```
...

update emp set ...

/* right way to check iirowcount (using select) */

select iirowcount, iierrornumber into rcount, enumber;
```

The following table lists the values of iirowcount and iierrornumber after the successful or unsuccessful execution of an SQL statement:

Statement	Success	Success	Error	Error
	iirowcount	iierrornumber	iirowcount	iierrornumber
Insert	number of rows	0	0	Ingres error number

Statement	Success	Success	Error	Error
Update	number of rows	0	0	Ingres error number
Delete	number of rows	0	0	Ingres error number
Select	0 or 1	0	0	Ingres error number
Assignment	1	0	0	Ingres error number
Commit	-1	0	-1	Ingres error number
Rollback	-1	0	-1	Ingres error number
Message	-1	0	-1	Ingres error number
Return	-1	0	-1	Ingres error number
If	no change	no change	no change	Ingres error number
Elseif	no change	no change	no change	Ingres error number
While	no change	no change	no change	Ingres error number
Else	no change	no change	no change	no change
Endif	no change	no change	no change	no change
Endloop	no change	no change	no change	no change
Endwhile	no change	no change	no change	no change

The execution of each database procedure statement sets the value of iierrornumber either to zero (no errors) or an error number. To check the execution status of any particular statement, iierrornumber must be examined immediately after the execution of the statement.

Errors occurring in IF, WHILE, MESSAGE, and RETURN statements do not set iierrornumber. However, any errors that occur during the evaluation of the condition of an IF or WHILE statement terminate the procedure and return control to the calling application.

Raise Error Statement

The RAISE ERROR statement generates an error. The DBMS Server responds to this error exactly as it does to any other error. If the RAISE ERROR statement is issued by a database procedure that is directly executed, the error is handled using the default error handling behavior or the user-supplied error handling mechanism. If the statement is executed inside a procedure invoked by a rule, the DBMS Server terminates the database procedure and rolls back any changes made by the procedure and any made by the statement that fired the rule.

The error number that is specified as an argument to raise error is returned to sqlerrd(1), and can be accessed using INQUIRE_SQL(DBMSERROR).

The RAISE ERROR statement can be used in conjunction with the conditional statements to tell the DBMS Server that the results from the statement that fired the rule violate some specified condition or constraint. For example, if a user attempts to update a table, a rule can invoke a database procedure that checks the updated values for compliance with a specified constraint. If the updated values fail the check, the RAISE ERROR statement can be used to roll back those updates.

Messages from Database Procedures

Database procedures use the SQL MESSAGE statement to return messages to users and applications. (The SQL MESSAGE statement is not the same as the forms MESSAGE statement.) Messages from database procedures can be trapped using the WHENEVER SQLMESSAGE statement or the SET_SQL(MESSAGEHANDLER) statement.

Messages from database procedures can return to your application before the database procedure has finished executing. For this reason, any message-handling routine must not execute any database statements in the current session. To issue database statements from a message-handling routine, switch sessions or open another session; if your message-handling routine switches sessions, it must switch back to the original session before returning from the message-handling routine.

Message Handling Using the Whenever Statement

If your application does not include an SQLCA, messages from database procedures are displayed on the terminal. If your application includes an SQLCA, use the WHENEVER statement to trap and handle messages from database procedures. If your application includes an SQLCA, messages are displayed only if your application issues the WHENEVER SQLMESSAGE CALL SQLPRINT statement.

The WHENEVER statement handles the following scenarios:

- All messages returned from directly executed database procedures

- The last message returned from a procedure called when a rule is fired

Messages issued by database procedures return message text and a message number to the calling application, and set sqlcode to +700.

Note: If a database procedure issues a MESSAGE statement and subsequently raises an error, the WHENEVER SQLMESSAGE does not trap the message. To trap all messages, use a message handler routine.

Message Handling Using User-Defined Handler Routines

To define a message handler routine, use the SET_SQL MESSAGEHANDLER statement. Routines defined this way can trap all messages returned by procedures that are executed by rules; the WHENEVER statement traps only the last message.

To enable or disable a message-handling routine, your application must issue the following SET_SQL statement:

```
exec sql set_sql(messagehandler = message_routine | 0)
```

To enable message handling, specify *message_routine* as a pointer to your message-handling routine or function. (For more information about pointers to functions, see the *Embedded SQL Companion Guide*.) To disable message handling, specify 0.

In addition to issuing the SET_SQL statement shown above, create the message-handling routine and link it with your embedded SQL application.

Rules

A rule invokes a specified database procedure when it detects specified changes to the database. When the specified change is detected by the DBMS Server, the rule is fired, and the database procedure associated with the rule is executed. Rules can be fired by:

- Any INSERT, UPDATE, or DELETE on a specified table (including a cursor update or delete)

- An update that changes one or more columns in a table

- A change that results in a specified condition (expressed as a qualification)

Note: Rules can also be fired by the QUEL statements APPEND, DELETE, and REPLACE.

Rules are created with the CREATE RULE statement and dropped with the DROP RULE statement. Dropping the procedure invoked by a rule does not drop the rule. For more information about creating and dropping rules, see Create Rule and Drop Rule in the chapter "SQL Statements."

Use rules to enforce referential and general integrity constraints, and for general purposes such as tracking all changes to particular tables or extending the permission system. For a detailed discussion of the use of rules to enforce referential integrity and data security, see the *Database Administrator Guide*.

When the firing condition of a rule is met, the specified database procedure is executed. The statement that fires the rule can originate in an application, a database procedure, or an Ingres tool such as QBF.

The statement that fires a rule and the database procedure invoked by the rule are treated as part of the same statement. The database procedure is executed before the statement that fired the rule completes. For this reason, a COMMIT or ROLLBACK statement cannot be issued in a database procedure invoked by a rule. If a statement fires more than one rule, the order in which the database procedures are executed is undefined. To trace the execution of rules, use the SET PRINTRULES statement.

For an UPDATE or DELETE statement, the DBMS Server executes a rule once for each row of the table that meets the firing condition of the rule. The rule is actually executed when the row is updated or deleted and not after the statement has completed. Thus, an UPDATE statement that ranges over a set of rows and that has a rule applied to it fires the rule each time a row is modified, at the time the row is modified. This style of execution is called instance-oriented.

Rules can be fired as the result of a statement issued from an invoked database procedure. Rules can be forward-chained, or nested, in this manner to a predefined number of levels. If this depth is exceeded, the DBMS Server issues an error and the statement is rolled back. By default, 20 levels of nesting can be defined. To change this value, set the RULE_DEPTH server parameter. Like a non-nested rule, when a nested rule fires, its database procedure is executed before the statement that fired it is completed.

Before creating or invoking a rule, the associated database procedure must exist. If it does not exist when the rule is created, the DBMS Server issues an error. If it does not exist when the rule is invoked, the DBMS Server issues an error and aborts the statement that attempted to fire the rule.

If an error occurs in the execution of a rule, the DBMS Server responds as if the statement firing the rule has experienced a fatal error and rolls back any changes made to the database by the statement and any made by the fired rule. An error also occurs when the RAISE ERROR statement is issued.

To create a rule against a table, you must own the table. In addition, you must either own the invoked database procedure or have execution privileges for that procedure.

After a rule is created against a table, any user who has permission to access the table using the operation specified by the rule has implicit permission to fire the rule and execute its associated database procedure.

Note: The DBA for a database can disable the firing of rules within that database during a session using the SET [NO]RULES statement. This option is provided as an aid to rule development and database maintenance tasks.

Important! If rules are disabled using the SET NORULES statement, the DBMS Server does not enforce table constraints or check options for views.

Rules are not fired by the COPY and MODIFY statements.

Multiple Session Connections

Embedded SQL can maintain multiple sessions (connections to a database). An application can open an initial session and, with subsequent CONNECT statements, open additional sessions connected with the same database or with different databases.

Multiple Sessions

To open a session, issue the CONNECT statement. To identify individual sessions in a multiple-session application, assign a connection name or numeric session identifier when issuing the CONNECT statement. Create multiple sessions that connect to the same database. For each connection, specify different runtime options, including the effective user.

The current session is established when an application connects to a database (by issuing the CONNECT statement) or switches sessions (using the SET CONNECTION or SET_SQL(SESSION) statements). If an error occurs when a program attempts to connect to a database, there is no current session in effect. Before the program can issue any queries, it must establish the current session by (successfully) connecting to a database or switching to a previously established session.

Session Identification

The CONNECT statement assigns each session a numeric session identifier and a connection name. The numeric identifier must be a positive integer. The connection name must be no longer than 128 characters.

Session Switching

To switch sessions using a numeric session identifier, use the SET_SQL(SESSION) statement. To switch sessions using the connection name, use the SET CONNECTION statement.

To determine the numeric session identifier for the current session, use the INQUIRE_SQL(SESSION) statement. To determine the connection name for the current statement, use the INQUIRE_SQL(SESSION) statement.

Applications can switch sessions in the following circumstances:

- Within a transaction

- While cursors are open

- Within SQL block statements, such as a select loop

  The program code for the nested session must be inside a host language subroutine. If it is not, the SQL preprocessor issues an error.

- Within subroutines called by a WHENEVER statement

- Within the following types of routines:

  - Data handlers (for long varchar or long byte data)

  - Error handlers

  - Message handlers

  - Database event handlers

Note: Sessions cannot be switched inside a database procedure.

After an application switches sessions, the error information obtained from the SQLCA or the INQUIRE_SQL statement is not updated until an SQL statement has completed in the new session.

Disconnection of Sessions

To disconnect from the current session, the application issues the DISCONNECT statement. To disconnect a session other than the current session, specify the numeric session identifier or connection name. To disconnect all connected sessions, issue the DISCONNECT ALL statement. For details, see Disconnect in the chapter "SQL Statements."

After an application disconnects from the current session in a multi-session application, the application must establish the current session by issuing the SET CONNECTION, SET_SQL(SESSION), or CONNECT statement. If no current session is in effect when an application issues a query, an error is returned.

Status Information in Multiple Sessions

The SQL Communications Area (SQLCA) is a data area in which the DBMS Server passes query status information to your application program. Although an application can sustain multiple sessions, there is only one SQLCA per application. However, the values returned by the INQUIRE_SQL(ERRORCODE) and INQUIRE_SQL(ERRORTEXT) statements are specific to a session.

If sessions are switched in a select loop (for example, by calling a routine that switches sessions) and database statements are executed in the alternate session, the values in the SQLCA are reset. When returning to the original session, the values in the SQLCA reflect the results of the statements issued in the alternate session and not the results of the select loop.

When sessions are switched, the values in the SQLCA fields are not updated until after the first SQL statement in the new session has completed. In contrast, the error information returned by IINQUIRE_SQL(ERRORTEXT and ERRORNO) always applies to the current session. The results of the session switch are returned in SQLSTATE.

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 either until the statement following a loop completes.) For this reason, the application must reset the sqlcode and sqlwarn fields before continuing the loop.

Multiple Sessions and the DBMS Server

The DBMS Server treats each session in a multiple-session application as an individual application. When creating multiple-session applications, keep the following points in mind:

- Be sure that the server parameter CONNECTED_SESSIONS is large enough to accommodate the number of sessions required by the application.

- In a multiple-session application, an application can encounter deadlock against itself. For example, one session must 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 selected from in another transaction in a second session, the second session waits indefinitely.

Example: Two Open Sessions

The following example shows the use of two open sessions in an application that gathers project information for updating the projects database using the personnel database to verify employee identification numbers. This example illustrates session switching and the use of connection names.

```
exec sql begin declare section;
    empid integer;
    found integer;
...
exec sql end declare section;

/* Set up two database connections */

exec sql connect projects as projects;
exec sql connect personnel as personnel;

/* Set 'projects' database to be current session */

exec sql set connection projects;
display project form
position cursor to emp id field

/* Validate user-entered employee id against
** master list of employees in personnel
** database. */
    found = 0;
    load empid host variable from field on form
/* Switch to 'personnel' database session */
    exec sql set connection personnel;
    exec sql repeated select 1 into :found
        from employee
        where empid = :empid;
/* Switch back to 'project' database session */
    exec sql set connection projects;
    if (found !=1) then
        print 'Invalid employee identification'
    else
        position cursor to next field
    endif;
end if

/* program code to validate other fields in 'projectform' */
if user selects 'Save' menu item
    get project information and update 'projectinfo' table
...
exec sql disconnect personnel;
exec sql disconnect projects;
```

Examples: Session Switching

The following examples illustrate 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. This example illustrates the use of numeric session identifiers.

The following is an example of 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;

exec sql set_sql(session = 1);

exec sql disconnect;
```

The following is an example of 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 Events

Database events enable an application or the DBMS Server to notify other applications that a specific event has occurred. An *event* is any occurrence that your application program is designed to handle.

The following diagram illustrates a typical use of database events: various applications or database procedures raise database events, and the DBMS Server 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.
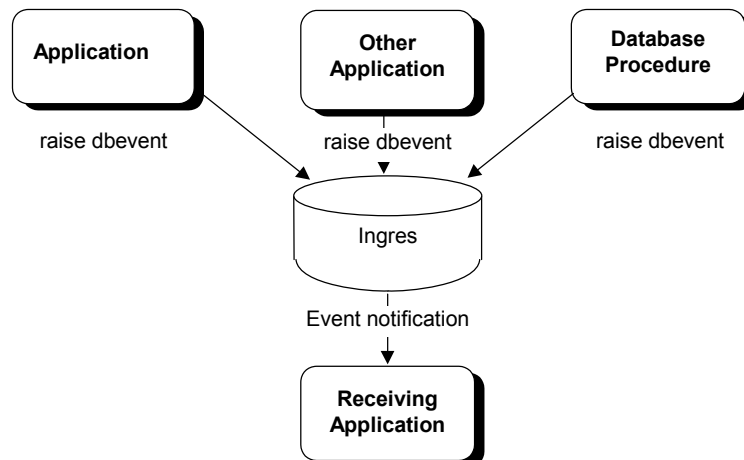
Database events can be raised by any of the following entities:

- An application that issues the RAISE DBEVENT statement

- An application that executes a database procedure that issues the RAISE DBEVENT statement

- As the result of firing a rule that executes a database procedure that issues the RAISE DBEVENT statement

    VMS: Database events cannot be broadcast across the nodes of a VMS cluster.

Example: Database Events in Conjunction with Rules

The following example uses database events in conjunction with rules to maintain inventory stock levels, as follows:

- When the inventory table is updated, a rule is fired.

- The rule executes a database procedure that checks stock levels.

- If the on-hand quantity of a part falls below the required minimum, the procedure raises a stock_low database event.

- Another application polls for stock_low database events. When the monitor application receives a stock_low database event, it generates a purchase order.

The detailed steps for this application are as follows:

1. Create a database event to be raised when the on-hand quantity of a part is low:

    ```
    create dbevent stock_low;
    ```

2. Create a rule that is fired when the qty_on_hand column of the inventory table is updated; the rule executes a database procedure when the quantity falls below 100 (assuming your business requires a minimum of 100 of each part you stock):
    ```
    create rule check_qty after update(qty_on_hand) of
    inventory where qty_on_hand < 100
    execute procedure issue_reorder(partno = old.partno);
    ```

3. Create the database procedure that raises the stock_low database event:

    ```
    create procedure reorder(partno varchar(25)) as
        begin
            raise dbevent stock_low
                (dbeventtext = partno)
        end
    ```

4. At runtime, the stock monitoring application must register to receive the stock_low database event:

    ```
    register dbevent stock_low
    ```

When the applications are running, the pieces work together as follows:

■   Whenever the qty_on_hand column of the inventory table is adjusted, the check_qty rule is fired; when the quantity falls below 100, the check_qty rule executes the reorder database procedure. The reorder procedure raises the stock_low database event.

■   The stock monitoring application issues the GET DBEVENT statement to check the database event queue and extract any database events (for which it is registered) that have been raised.

■   For each database event detected, the stock monitoring application issues an IINQUIRE_SQL statement to retrieve information about the database event. If it is the stock_low database event, the stock monitoring application generates a purchase order for the part.

The following diagram illustrates the process:
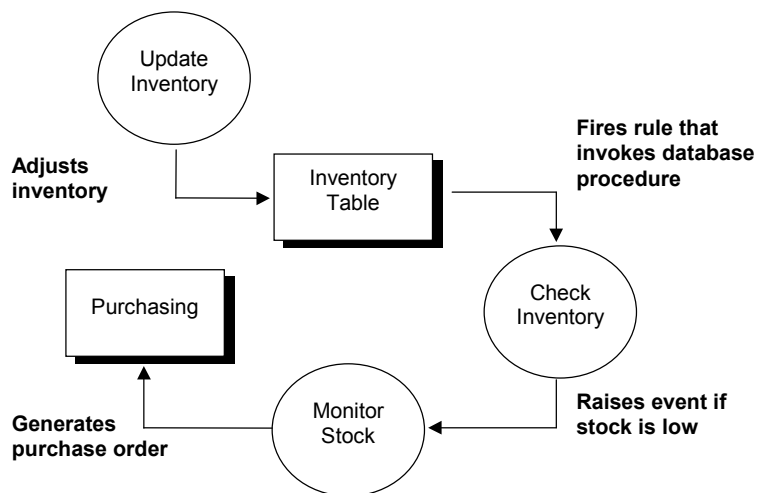
Database Event Statements

Database events use the following SQL statements:

- CREATE DBEVENT
- RAISE DBEVENT
- REGISTER DBEVENT
- GET DBEVENT
- REMOVE DBEVENT
- DROP DBEVENT
- INQUIRE_SQL
- SET_SQL
- GRANT…ON DBEVENT
- HELP PERMIT ON DBEVENT

Create a Database Event

To create a database event, use the CREATE DBEVENT statement:

```
create dbevent event_name
```

event_name

Is a unique database event name and a valid object name.

Database events for which appropriate permissions have been granted (raise or register) 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.

If a database event is created from within a transaction and the transaction is rolled back, creation of the database event is also rolled back.

Raise a Database Event

To raise a database event, use the RAISE DBEVENT statement:

```
raise dbevent event_name [event_text] [with [no] share]
```

The RAISE DBEVENT statement can be issued from interactive or embedded SQL applications, or from within a database procedure, including procedures that execute as the result of a rule firing. When the RAISE DBEVENT statement is issued, the DBMS Server 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.

A session can raise any database event that is owned by the effective user of the session, and any database event owned by another user who has granted the raise privilege to the effective user, group, role, or public.

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.

The [NO]SHARE parameter specifies whether the DBMS Server issues database event messages to all applications registered for the database event, or only to the application that raised the database event (or, if the database event was raised as the result of a rule firing, issued the query that raised the database event). If share or omit this parameter are specified, the DBMS Server notifies all registered applications when the database event is raised. If NOSHARE is specified, the DBMS Server notifies only the application that issued the query that raised the database event (assuming the program was also registered to receive the database event).

If a transaction issues the RAISE DBEVENT statement, and the transaction is subsequently rolled back, database event queues are not affected by the rollback: the raised database event remains queued to all sessions that registered for the database event.

Register Applications to Receive a Database Event

To register an application to receive database events, use the REGISTER DBEVENT statement:

```
register dbevent event_name
```

event_name

Specifies an existing database event.

Sessions must register for each database event to be received. A session can register for all database events that the session's effective user owns, and all database events for which the effective user, group, role, or public has been granted register privilege. For each database event, the registration is in effect until the session issues the REMOVE DBEVENT statement or disconnects from the database.

The DBMS Server issues an error if:

- A session attempts to register for a non-existent database event

- A session attempts to register for a database event for which the session does not have register privilege

- A session attempts to register twice for the same database event. If the REGISTER DBEVENT statement is issued from within a transaction that is subsequently rolled back, the registration is not rolled back.

The REGISTER DBEVENT statement can be issued from interactive or embedded SQL, or from within a database procedure.

Receive a Database Event

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:

Session Event Queue
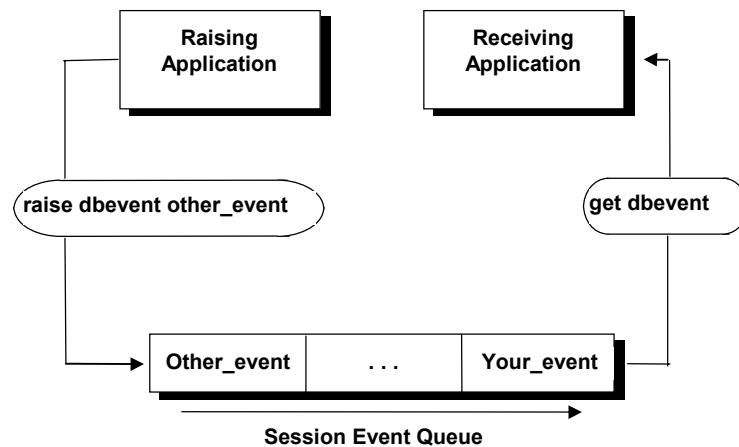
GET DBEVENT returns database events for the current session only; if an application runs multiple sessions, each session must register to receive the desired database events, and the application must switch sessions to receive database events queued for each session.

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 period, the GET DBEVENT statement times out, and no database event is returned. If GET DBEVENT WITH NOWAIT is specified, the DBMS Server 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 Server 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; returned in lowercase letters.

- **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.

Process Database Events

Three methods can be used to process database events:

- The GET DBEVENT statement is used to explicitly consume each database event from the database event queue of the session. Typically, a loop is constructed to poll for database events and call 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 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 INIQUIRE_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 must be purged.

Database events (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 must ensure that all database events in the database events queue are processed by using the GET DBEVENT loop, which is described below.

Get Dbevent Statement

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 = ''
```

Whenever Dbevent Statement

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 (also the standalone SQLCODE variable is set to 710; SQLSTATE is not affected). However, if a query results in an error that resets sqlcode, the WHENEVER statement does not trap the database event. The database event is still 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 must 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
```

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.

Remove a Database Event Registration

To remove a database event registration, use the REMOVE DBEVENT statement:

```
remove dbevent event_name
```

event_name

Specifies a database event for which the application has previously registered.

After a database event registration is removed, the DBMS Server does 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 Server issues an error.

Drop 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 Server 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.

Privileges and Database Events

The raise privilege is required to raise database events, and the register privilege is required to register for database events. To grant these privileges, use the GRANT statement:

```
grant raise on dbevent event_name to
```

```
grant register on dbevent event_name to
```

To revoke these privileges, use the REVOKE statement. To display the number for the raise or register privileges, use the HELP PERMIT statement. To display the permits defined for a specific database event, use the following statement:

```
help permit on dbevent event_name{, event_name}
```

Trace Database Events

The following features enable your application to display and trace database events:

- To enable or disable the display of database event trace information for an application when it raises a database event, SET [NO]PRINTDBEVENTS statement.

  To enable the display of database events as they are raised by the application, specify set PRINTDBEVENTS. To disable the display of database events, specify SET NOPRINTDBEVENTS.

- To enable or disable the logging of raised database events to the installation log file, use the SET [NO]LOGDBEVENTS statement:

  To enable the logging of database events as they are raised by the application, specify SET LOGDBEVENTS. To disable the logging of database events, specify SET NOLOGDBEVENTS.

- To enable or disable the display of database events as they are received by an application, use the EXEC SQL SET_SQL(DBEVENTDISPLAY = 1| 0 | *variable*)

  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 the *System Administrator Guide*.

- A routine can be created that traps 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 EXEC SQL SET_SQL(DBEVENTHANDLER = *event_routine* | 0) statement.

  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 the *Embedded SQL 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: SQL Statements

This chapter describes the purpose, syntax, and use of each SQL statement. SQL statements that pertain only to distributed databases are not included.

SQL statements described in this chapter are for the release of SQL indicated by the following value in the iidbcapabilities system catalog table:

CAP_CAPABILITY	CAP_VALUE
INGRES/SQL_LEVEL	00930

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

Statements for Ingres Star

The following statements, which pertain solely to distributed databases, are described in the *Ingres Star User Guide*:

- DIRECT CONNECT
- DIRECT DISCONNECT
- DIRECT EXECUTE
- REGISTER...AS LINK
- REMOVE

Certain statements in this guide have additional considerations when used in a distributed environment, which is noted in the statement description.

Alter Group

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER GROUP statement adds or drops user identifiers from the user list associated with a group identifier.

Syntax

The Alter Group statement has the following format:

```
[EXEC SQL] ALTER GROUP group_id {, group_id}
ADD USERS (user_id {, user_id}) | DROP USERS (user_id {, user_id}) | DROP ALL
```

ALTER GROUP *group_id* {, *group_id*}

Modifies the list of users associated with a group identifier (*group_id*). Individual users can be added or dropped, or the entire list can be dropped. An add and a drop operation cannot be performed in the same ALTER GROUP statement.

The *group_id* must be an existing group identifier. If a *group_id* that does not exist is specified, the DBMS Server issues a warning but continues processing any valid *group_ids* in the list. If a specific *user_id* occurs more than once in the user list, additional occurrences of the specified *user_id* are ignored. No errors are issued.

ADD USERS (*user_id* {, *user_id*})

Adds the specified users to the user list associated with the *group_ids*. The *user_ids* must exist when they are added to a group. If a specified user is not defined in the installation, the DBMS Server issues an error but processes the remaining user identifiers. If any of the specified users are already part of the group user list, the DBMS Server returns a warning but adds any other valid specified users to the list.

DROP USERS (*user_id* {, *user_id*}) | DROP ALL

Removes the specified users from the user list of the group identifier. If any of the specified users are not in the group's user list, the DBMS Server returns a warning but does not abort the statement, and any other valid specified users are dropped. A user cannot be dropped from a group if that group is the default group of the user. (Use the alter user statement to change a user's default group.)

The drop all clause removes all users from the group's user list. A group cannot be dropped if it has any members in its user list. If any member of the specified group has that group as its default group, drop all results in an error. Use the alter user statement to change the user's default group before attempting to drop all.

If a user is dropped from a group in a session that is associated with that group, the user retains the privileges of the group until the session terminates.

Embedded Usage

You cannot use host language variables in an embedded ALTER GROUP statement.

Permissions

You must have maintain_users privileges and be working in a session connected with the iidbdb database.

Locking

The ALTER GROUP statement locks pages in the iiusergroup catalog in the iidbdb. This can cause sessions attempting to connect to the server to be suspended until the ALTER GROUP statement is completed.

Related Statements

Create Group (see page 352)

Drop Group (see page 486)

Examples: Alter Group

The following examples add and drop user identifiers from the user list associated with a group identifier:

1. Add users to the group, sales_clerks.

    ```
    exec sql alter group sales_clerks
        add users (dannyh, helent);
    ```

2. Drop three users from the group, tel_sales.

    ```
    exec sql alter group tel_sales
        drop users (harryk, joanb, elainet);
    ```

3. In an application, drop all users from the group, researchers.

    ```
    exec sql alter group researchers drop all;
    ```

Alter Location

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER LOCATION statement changes the type of files that can be created at an existing location.

Current usage of the location is unaffected, but future attempts to extend a database to the target extension are constrained by the new usage setting.

Syntax

The ALTER LOCATION statement has the following format:

```
[EXEC SQL] ALTER LOCATION location_name
              WITH USAGE = (usage_type {, usage_type}) | NOUSAGE
```

location name

Specifies the name of an existing disk and directory combination.

usage_type

Specifies the types of file that can be stored at this location. Valid values are:

- DATABASE
- WORK
- JOURNAL
- CHECKPOINT
- DUMP
- ALL

NOUSAGE

Prevents any files from being stored at the location.

Embedded Usage

In an embedded ALTER LOCATION statement, the usage portion of the WITH clause can be specified using a host string variable. The preprocessor does not validate the WITH clause.

Permissions

You must have maintain_locations privileges and be connected to the iidbdb to issue this statement.

Locking

The ALTER LOCATION statement locks pages in the iilocation_info system catalog.

Related Statements

Create Location (see page 363)

Drop Location (see page 489)

Examples: Alter Location

The following examples change the type of files that can be created at an existing location:

1. Specify that only checkpoint files can be created at the checkpoint_disk location.

    ```
    alter location checkpoint_disk
        with usage = (checkpoint);
    ```

2. Prevent any files from being created at the new_db location.

    ```
    alter location new_db with nousage;
    ```

Alter Profile

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER PROFILE statement alters a user profile.

Syntax

The ALTER PROFILE statement has the following format:

```
[EXEC SQL] ALTER [DEFAULT] PROFILE [profile_name]
[ADD PRIVILEGES( priv {,priv}) | DROP PRIVILEGES( priv {,priv})]
[WITH with_item {, with_item}]

with_item = NOPRIVILEGES | PRIVILEGES = ( priv {, priv} )
                        | NOGROUP | GROUP = default_group
                        | SECURITY_AUDIT = ( audit_opt {,audit_opt})
                        | NOEXPIRE_DATE | EXPIRE_DATE = 'expire_date'
                        | DEFAULT_PRIVILEGES = ( priv {, priv} ) | all
                        | NODEFAULT_PRIVILEGES
```

ALTER DEFAULT PROFILE

Modifies the settings of a default profile. You cannot specify both DEFAULT and a *profile_name* in the same statement.

profile_name

Specifies the name of an existing profile. Each user can be given a profile, which provides the default attributes for that user. The *profile_name* can be specified using a delimited identifier.

ADD PRIVILEGES | DROP PRIVILEGES

Adds or drops privileges to or from the user profile. You cannot use either ADD PRIVILEGES or DROP PRIVILEGES if *with_option* is specified in the *with_clause*.

If a *with_clause* item is not specified, its value is left unchanged.

priv

Specifies one of the following subject privileges, which apply to the user regardless of the database to which the user is connected. If the privileges clause is omitted, the default is NOPRIVILEGES.

CREATEDB

Allows users to create databases.

TRACE

Allows the user to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows the user to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also allows the user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating or altering users, profiles, group and roles, and to grant or revoke database and installation resource controls.

default_group

Specifies the default group for users with this profile. Must be an existing group.

To specify that the user is not assigned to a group, use the NOGROUP option.

Default: NOGROUP if the group clause is omitted.

audit_opt

Defines security audit options:

ALL_EVENTS

All activity by the user is audited.

DEFAULT_EVENTS

Only default security auditing is performed, as specified with the ENABLE and DISABLE SECURITY_AUDIT statements. This is the default if the SECURITY_AUDIT clause is omitted.

QUERY_TEXT

Auditing of the query text associated with specific user queries is performed. Security auditing of query text must be enabled as a whole, using the ENALBE and DISABLE SECURITY_AUDIT statements with the QUERY_TEXT option (for example: ENABLE SECURITY_AUDIT QUERY_TEXT).

expire_date

Specifies an optional expiration date associated with each user using this profile. Any valid date can be used. When the expiration date is reached, the user is no longer able to log on. If NOEXPIRE_DATE is specified, this profile has no expiration limit.

DEFAULT_PRIVILEGES =
(*priv {, priv}*) | ALL| NODEFAULT_PRIVILEGES

Defines the privileges initially active when connecting to Ingres. These must be a subset of those privileges granted to the user.

ALL

All the privileges held by the profile are initially active.

NODEFAULT_PRIVILEGES

No privileges are initially active.

More Information:

Create Profile (see page 378)

Embedded Usage

In an embedded ALTER PROFILE statement, the WITH clause can be specified using a host string variable (with :*hostvar*).

Permissions

You must have maintain_users privileges and be connected to the iidbdb database. Additional privileges are required to perform certain operations, as summarized in the table below:

Action	Privilege Required
Change security audit attributes	maintain_audit

Locking

The ALTER PROFILE statement locks iiprofile exclusively.

Related Statements

Alter User (see page 312)

Create Profile (see page 376)

Create User (see page 440)

Drop Profile (see page 491)

Examples: Alter Profile

The following examples alter a user profile:

1. Update a default profile by using the alter default profile variant of the ALTER PROFILE statement.

```
alter default profile
    with expire_date = '30 days';
```

2. Change the default profile to include createdb privileges.

```
alter default profile
    add privileges ( createdb );
```

Only one of default profile and profile *profile_name* can be specified.

3. Alter the *trusted* profile to add the createdb privilege and make the default group *trusted_group*:

```
alter profile trusted
add privileges ( createdb )
    with group = trusted_group
```

All users currently using this profile have the appropriate changes made to their security privilege and group.

4. Alter the security auditing for profile, clerk.

```
alter profile clerk
with security_audit = (query_text, default_events);
```

Alter Role

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER ROLE statement changes the attributes associated with a role identifier.

Use add privileges to give the user additional privileges. Use drop privileges to remove privileges from the user. You cannot use either add privileges, or drop privileges if *with_option* is specified in the *with_clause*.

Syntax

The ALTER ROLE statement has the following format:

```
[EXEC SQL] ALTER ROLE role_id {, role_id}
[ADD PRIVILEGES ( priv {,priv} ) |  DROP PRIVILEGES ( priv {,priv} )]
[WITH with_option {,with_option}]

with_option = NOPASSWORD | PASSWORD = 'role_password' | EXTERNAL_PASSWORD
                       | NO PRIVILEGES | PRIVILEGES = ( priv {,priv} )
                       | NOSECURITY_AUDIT | SECURITY_AUDIT
```

role_id

Must exist in the installation. If one or more of the specified role identifiers do not exist, the DBMS Server issues a warning, but all valid role identifiers are processed.

To create roles, use the Create Role statement. For more information about role identifiers, see the *Database Administrator Guide*.

priv

Specifies one of the following subject privileges, which apply to the user regardless of the database to which the user is connected. If the privileges clause is omitted, the default is NOPRIVILEGES.

CREATEDB

Allows users to create databases.

TRACE

Allows the user to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows the user to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also allows the user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating or altering users, profiles, group and roles, and to grant or revoke database and installation resource controls.

NOSECURITY_AUDIT | SECURITY_AUDIT

Specifies audit options, as follows:

NOSECURITY_AUDIT

Uses the security_audit level for the user using the role.

SECURITY_AUDIT

Audits all activity for anyone who uses the role, regardless of any SECURITY_AUDIT level set for an individual user.

Default: NOSECURITY_AUDIT if the security_audit clause is omitted.

Caution! If no password is specified, any session has access to the specified role identifier and its associated permissions.

Embedded Usage

In an embedded ALTER ROLE statement, the preprocessor does not validate the syntax of the WITH clause.

Permissions

You must have maintain_users privileges and be connected to the iidbdb database. Additional privileges are required to perform certain operations, as summarized in the table below:

Action	Privilege Required
Change security audit attributes	maintain_audit

Locking

The ALTER ROLE statement locks pages in the iirole catalog of the iidbdb. This can cause sessions attempting to connect to the server to suspend until the statement is completed.

Related Statements

Create Role (see page 380)

Drop Role (see page 492)

Examples: Alter Role

The following examples change the attributes associated with a role identifier:

1. Change the password for the role identifier, new_accounts, to eggbasket.

   ```
   alter role new_accounts with
       password = 'eggbasket';
   ```

2. Remove the password associated with the identifier, chk_inventory.

   ```
   alter role chk_inventory with nopassword;
   ```

3. In an application, change the password for the role identifier, mon_end_report to goodnews.

   ```
   exec sql alter role mon_end_report with
       password = goodnews;
   ```

4. Alter a role to remove a privilege and audits all activity performed when the role is active.

   ```
   alter role sysdba
       drop privileges (trace)
       with security_audit;
   ```

Alter Security_Audit

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER SECURITY_AUDIT statement allows the current security audit log to be switched and for security auditing to be suspended or resumed in the current installation. This statement takes effect immediately and cannot be issued within a multi-statement transaction. It is available in dynamic SQL.

Syntax

The ALTER SECURITY_AUDIT statement has the following format:

```
[EXEC SQL] ALTER SECURITY_AUDIT [SUSPEND|RESUME|RESTART|STOP] [WITH AUDIT_LOG =
'audit_filename']
```

ALTER SECURITY_AUDIT SUSPEND|RESUME

Allows auditing to be suspended and later resumed. This allows maintenance on security audit logs to take place as required. When auditing is suspended any sessions that attempt to generate security audit records are stalled until auditing is resumed. Auditing is suspended immediately after the audit record logging the ALTER SECURITY_AUDIT statement is written.

Auditing can only be suspended when it is active, and resumed when it is suspended.

On installation restart, auditing is resumed automatically.

To allow the audit system to be resumed, users with maintain_audit privilege can continue to access Ingres even when auditing is suspended. In this case any audit events generated are written to the audit log.

ALTER SECURITY_AUDIT RESTART

Restarts auditing.

ALTER SECURITY_AUDIT STOP

Stops auditing on request. This statement cannot be used to start security logging for servers that were not started with logging enabled. Auditing can only be stopped when it is active, and restarted when it is stopped.

Security auditing can be stopped, either by issuing an ALTER SECURITY_AUDIT STOP statement, or as the result of an audit system condition such as *logfull* or *on-error*.

ALTER SECURITY_AUDIT WITH AUDIT_LOG = '*audit_filename*'

Sets the current installation security log. The security audit log can be changed whenever auditing is active (that is, when it is not stopped or suspended), or when restarting or resuming auditing. The audit log file specified must actually exist in the Ingres audit configuration.

Embedded Usage

Audit_filename can be specified using a string hostname variable in an embedded ALTER SECURITY_AUDIT statement.

Permissions

ALTER SECURITY_AUDIT requires the maintain_audit privilege and can only be issued while connected to the iidbdb database.

Related Statements

Disable Security_Audit (see page 477)

Enable Security_Audit (see page 499)

Examples: Alter Security_Audit

The following examples allow the current security audit log to be switched and for security auditing to be suspended or resumed in the current installation:

1. Restart security auditing after it has been suspended.

   ```
   alter security_audit resume;
   ```

2. Restart auditing, switching to a new audit log.

   Windows:

   ```
   alter security_audit restart
   with audit_log = 'd:\oping\ingres\files\audit.log'
   ```

   UNIX:

   ```
   alter security_audit restart
   with audit_log = /install/ingres/files/audit.3
   ```

   VMS:

   ```
   alter security_audit restart
   with audit_log = disk$7:[ingres.files]audit.3
   ```

3. Cause Ingres to log events to the auditlog.7 file.

   ```
   alter security_audit
       with audit_log = '/auditdisk/auditlog.7';
   ```

Alter Sequence

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER SEQUENCE statement changes sequence settings that were specified when the sequence was created.

Syntax

The ALTER SEQUENCE syntax has the following format:

```
[EXEC SQL] ALTER SEQUENCE [schema.]sequence_name [sequence_options]
```

sequence_options

See Create Sequence (see page 398) for details.

Permissions

You must have "create_sequence" privileges. You must also have the "next" privilege to retrieve values from a defined sequence. For information on the "next" privilege, see Grant (privilege) (see page 534).

Locking and Sequences

In applications, sequences use logical locks that allow multiple transactions to retrieve and update the sequence value while preventing changes to the underlying sequence definition. The logical lock is held until the end of the transaction.

Related Statements

Create Sequence (see page 398)

Drop Sequence (see page 495)

Examples: Alter Sequence

The following examples change sequence settings that were specified when the sequence was created:

1. Change the start value so that sequence "XYZ" starts at sequence item 10.

    ```
    alter sequence XYZ restart with 10
    ```

2. Change the increment value of sequence "XYZ" to 20.

    ```
    alter sequence XYZ increment by 20
    ```

Alter Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER TABLE statement adds or removes a table-level constraint or a column from a base table.

The statement also modifies the character columns in a table to Unicode column types.

Constraints can also be specified when the base table is created using the CREATE TABLE statement.

Note: To use this statement, the table must have a page size of 4K or larger.

Syntax

The ALTER TABLE statement has the following format:

```
[EXEC SQL] ALTER TABLE [schema.]table_name
    ADD [column] column_name FORMAT [default_clause]
        [null_clause] [column_constraint] [COLLATE collation_name]
| DROP [COLUMN] column_name RESTRICT | CASCADE
| ADD [CONSTRAINT constraint_name] constraint_spec
| DROP CONSTRAINT constraint_name RESTRICT | CASCADE
| ALTER [COLUMN] column_name FORMAT [default_clause]
        [null_clause] [column_constraint] [COLLATE collation_name]
```

ALTER TABLE *table_name* ADD [COLUMN] *column_name* FORMAT [*default_clause*] [*null_clause*] [*column_constraint*] [COLLATE *collation_name*]

Adds a column. The column_name cannot exist in the table at the time the ALTER TABLE statement is issued. The format default_clause, null_clause, column_constraint, and collation_name of the column have the same structure as for the CREATE TABLE statement, except that WITH NULL WITH DEFAULT and NOT NULL NOT DEFAULT are not allowed.

The column is logically placed in the table definition after the last existing column. Only one column at a time can be added with the ALTER TABLE statement. When a column is added, the number of columns in the table cannot exceed the maximum number of columns in a table, (which is 1024), and the row width cannot exceed the maximum row width for the page size or the max_tuple_length setting.

Note: When a column is added to a table, the logical definition of the table is changed without physically reorganizing the data. Therefore, after adding columns, use the modify command to rebuild the table.

ALTER TABLE *table_name* **DROP [COLUMN]** *column_name* **RESTRICT | CASCADE**

Drops a column. The column *column_name* must exist in the table's *table_name.* Only one column can be dropped in a single ALTER TABLE statement. One of the following options must be specified: restrict or cascade (see page 309).

Note: A column cannot be dropped that is being used as a partitioning column or storage structure key column.

Note: When a column is dropped, the logical definition of the table is changed without physically reorganizing the data. The column number and the space associated with the dropped column are not reused. After dropping columns, use the modify command to clean up space and to rebuild the table.

ALTER TABLE *table_name* **ADD CONSTRAINT** *constraint_name* *constraint_clause*

Adds a table-level constraint (see page 307).

ALTER TABLE *table_name* **DROP CONSTRAINT** *constraint_name* **RESTRICT | CASCADE**

Drops a constraint (see page 307).

One of the following options must be specified: restrict or cascade (see page 309).

ALTER TABLE *table_name* **ALTER [COLUMN]** *column_name* **FORMAT** [*default_clause*]
[*null_clause*] [*column_constraint*] **[COLLATE** *collation_name*]

The ALTER TABLE *table_name* ALTER COLUMN *column_name* statement changes the characteristics of a column.

This statement can be used to:

- Change the size of a character column to preserve the existing default_clause, null_clause, column_constraint, and collation_name.

- Change the column from a non-Unicode data type to a Unicode data type.

  Note: The database must be Unicode enabled either having been created as a Unicode-enabled database with the –i (Normalization Form C (NFC)) or –n (Normalization Form D (NFD)) flag, or by using the alterdb command.

- Change from one character data type to another.

- Change a column from not null to null.

- Change the default value of a column.

  Note: The default value of a column cannot be changed to null.

- Change the collation sequence of the column. The *collation_name* can be one of the following:

  - **UNICODE—**Specifies collation for columns containing Unicode data (nchar and nvarchar data types). This is the default collation for Unicode columns.

  - **UNICODE_CASE_INSENSITIVE—**Specifies case insensitive collation for columns containing Unicode data (nchar and nvarchar data types).

  - **SQL_CHARACTER—**Specifies the collation for columns containing char, C, varchar, and text data. This is the default collation for non-Unicode columns.

Constraint Specifications

When a constraint (see page 414) is added to a table, the table is checked to ensure that its contents do not violate the constraint. If the contents of the table do violate the constraint, the DBMS Server returns an error and does not add the constraint.

The following table summarizes the elements of the constraint_clause:

Type	Keyword	Example
referential	REFERENCES	alter table dept add constraint chkmgr foreign key(mgr) references emp(ename) on delete set null;
unique	UNIQUE	alter table emp add unique (eno, ename);
check	CHECK	alter table emp add check (salary>0);
primary key	PRIMARY KEY	alter table emp add constraint ekey primary key(eno);

Named Constraints

If the constraint name is omitted, the DBMS Server assigns a name.

To assign a name to a constraint on the ALTER TABLE statement, use the following syntax:

```
ALTER TABLE table_name ADD CONSTRAINT constraint_name constraint_clause
```

constraint_name

Assigns a name to the constraint. It must be a valid object name. The keyword CONSTRAINT must be used only when specifying a name.

For example, the following statement adds a named constraint to the emp table:

```
alter table emp add constraint chksal check (salary>0);
```

The following statement adds an internally named constraint to the emp table:

```
alter table emp add check (age>0);
```

To drop a constraint, using the following syntax:

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name RESTRICT|CASCADE
```

For example, the following ALTER TABLE statement drops the constraint named chksal:

```
alter table emp drop constraint chksal restrict;
```

To find a system-defined constraint name, select the name from the iiconstraints system catalog:

```
select * from iiconstraints where table_name = table_name;
```

If a system-defined constraint name is being dropped, specify the constraint name using a delimited identifier (that is, in double quotes), because system-defined constraint names include special characters.

If a unique constraint upon which referential constraints depend is dropped, the dependent constraints are automatically dropped (unless restrict is specified). For example, given the following tables and constraints:

```
create table dept (     dname    char(10) not null unique,
            ...);
create table emp  (     ename    char(10),
                            dname         char(10)
        references dept(dname));
```

If the unique constraint on the dname column of the dept table is dropped, the referential constraint on the dname column of emp is dropped automatically.

Restrict and Cascade

When a constraint or a column is dropped, specify either the RESTRICT or CASCADE option:

RESTRICT

Does not drop the constraint or column if one or more objects exist that depend on it. For example:

- A view with reference to the column in the base table

- A check constraint on the column being dropped

- A secondary index defined with this column

CASCADE

Deletes all objects that depend on the dropped constraint or column.
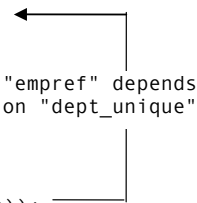
For example, if a unique constraint upon which a referential constraint is dependent is dropped, the dependent constraints are dropped. If a column is dropped, all integrity constraints, grants, views, and indexes that depend on the column are dropped.

The user is not provided with information describing the dependent objects that are dropped.

Note: Procedures and rules dependent on dropped columns are not dropped; instead, an error is returned when the rule or procedure is executed.

For example, the following statements create two tables with referential constraint. The referential constraint of the second table depends on the unique constraint of the first table:

```
create table dept (
    name char(10) not null,
    location char(20),
constraint dept_unique unique(name)
    with structure=hash);


create table emp (
    name char(10) not null,
    salary decimal(10,2),
    dept char(10)
constraint empref references dept(name));
```

"empref" depends on "dept_unique"

If the dept_unique constraint is dropped, the RESTRICT and CASCADE clauses determine the outcome of the ALTER TABLE statement as follows:

```
alter table dept drop constraint dept_unique restrict;
```

returns an error, because there is a referential constraint that depends on dept_unique. However,

```
alter table dept drop constraint dept_unique cascade;
```

deletes both the dept_unique constraint and the dependent empref constraint.

Embedded Usage

In an embedded ALTER TABLE statement, specify the WITH clause using a host string variable (WITH :*hostvar*).

Permissions

Only the owner of a table can add or drop constraints or columns for that table. To define a referential constraint that refers to a table owned by another user, you must have the references privilege for the columns to which the constraint refers.

Locking

The ALTER TABLE statement acquires an exclusive lock on the table at the start of execution. The lock is held until the end of the transaction.

Related Statements

Create Index (see page 353)

Create Table (see page 402)

Modify (see page 579)

Examples: Alter Table

The following examples add and remove a table-level constraint and a column from the existing base table.

The examples are based on the following table:

```
create table emp         (
name    char(10) not null not default,
salary  decimal(10,2)
dept    char(10),
age     integer not null not default);
```

1. Add a check constraint to ensure that employee ages are correct.

   ```
   alter table emp add constraint
   check_age check (age > 0);
   ```

2. Drop the age-checking constraint and any dependent constraints.

   ```
   alter table emp drop constraint check_age cascade;
   ```

3. Add a column to an existing table.

   ```
   alter table emp add column location char(10);
   ```

4. Drop a column from an existing table.

   ```
   alter table emp drop column location restrict;
   ```

5. Change the size of a character column.

   ```
   alter table emp alter column name char(32);
   ```

6. Change the column from a non-Unicode data type to a Unicode data type.

   ```
   alter table emp alter column name nchar(32);
   ```

7. Change from one character data type to another. For example, from char to varchar.

   ```
   alter table emp alter column name varchar(32)  not null with default;
   ```

8. Change a column from not null to null

   ```
   alter table emp alter column name char(32) with null;
   ```

9. Change the collation sequence of a column

   ```
   alter table emp alter column name nchar(32) not null not default collate
   unicode_case_insensitive;
   ```

Alter User

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ALTER USER statement changes the characteristics of an existing user.

Use ADD PRIVILEGES to give the user additional privileges. Use DROP PRIVILEGES to remove privileges from the user.

Note: You cannot use either ADD PRIVILEGES or DROP PRIVILEGES if *with_option* is specified in the *with_clause*.

Syntax

The ALTER USER statement has the following format:

```
[EXEC SQL] ALTER USER user_name
[ADD PRIVILEGES (priv {, priv}) |DROP PRIVILEGES (priv {, priv})]
[WITH with_item {, with_item}]

with_item = NOPRIVILEGES| PRIVILEGES = ( priv {, priv} )
                          | NOGROUP | GROUP = default_group
                          | SECURITY_AUDIT= ( audit_opt {,audit_opt})
                          | NOEXPIREDATE | EXPIRE_DATE = 'expire_date'
                          | DEFAULT_PRIVILEGES = (priv {,priv})| ALL
                                      | NODEFAULT_PRIVILEGES
                          | NOPROFILE | PROFILE= profile_name
                          | NOPASSWORD | PASSWORD = 'user_password'
                                      | PASSWORD = X'encrypted_role_password'
                          | EXTERNAL_PASSWORD
                          | OLDPASSWORD = 'oldpassword'
```

user_name

Specifies the user name. The user must be an existing Ingres user.

priv

Specifies one of the following subject privileges, which apply to the user regardless of the database to which the user is connected. If the privileges clause is omitted, the default is NOPRIVILEGES.

CREATEDB

Allows users to create databases.

TRACE

Allows the user to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows the user to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also allows the user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating or altering users, profiles, group and roles, and to grant or revoke database and installation resource controls.

default group

Specifies the default group to which the user belongs. Must be an existing group. For details about groups, see Create Group (see page 352). To specify that the user is not assigned to a group, use the NOGROUP option.

Default: NOGROUP if the group clause is omitted.

audit_opt

Defines security audit options:

ALL_EVENTS

All activity by the user is audited.

DEFAULT_EVENTS

Only default security auditing is performed, as specified with the ENABLE and DISABLE SECURITY_AUDIT statements. This is the default if the SECURITY_AUDIT clause is omitted.

QUERY_TEXT

Auditing of the query text associated with specific user queries is performed. Security auditing of query text must be enabled as a whole, using the ENALBE and DISABLE SECURITY_AUDIT statements with the QUERY_TEXT option (for example: ENABLE SECURITY_AUDIT QUERY_TEXT).

expire_date

Specifies an optional expiration date associated with each user. Any valid date can be used. Once the expiration date is reached, the user is no longer able to log on.

Default: NOEXPIRE_DATE if the EXPIRE_DATE clause is omitted.

DEFAULT_PRIVILEGES

Defines the privileges initially active when connecting to Ingres. These must be a subset of those privileges granted to the user.

NODEFAULT_PRIVILEGES

Specifies that the session is started with no privileges active. Allows default privileges to be removed.

profile_name

Allows a profile to be specified for a particular user.

Default: NOPROFILE if the profile clause is omitted.

user_password

Allows users to change their own password. If the OLDPASSWORD clause is missing or invalid, the password is unchanged. In addition, users with the maintain_users privilege can change or remove any password.

EXTERNAL_PASSWORD

Allows a user's password to be authenticated externally to Ingres. The password is passed to an external authentication server for authentication.

oldpassword

Specifies the user's old password.

Embedded Usage

In an embedded ALTER USER statement, specify the WITH clause using a host string variable (with :*hostvar*). The privilege type can be specified using a host string variable.

Permissions

You must be connected to the iidbdb database. The maintain_users privilege is required, except for users who simply want to change their own password. You must have maintain_audit privileges to change security audit attributes.

Locking

The ALTER USER statement locks pages in the iiuser system catalog.

Related Statements

Create User (see page 440)

Create Profile (see page 376)

Alter Profile (see page 294)

Drop Profile (see page 491)

Examples: Alter User

The following examples change the characteristics of an existing user:

1. Change an existing user, specifying privileges and group.

```
alter user bspring with
    group = engineering,
    noprivileges;
```

2. Change an existing user, specifying privileges and group.

```
alter user barney with
    group = marketing,
    privileges = (createdb,trace,security);
```

3. Specify no expiration date for a predefined user.

```
alter user bspring
    with noexpiration_date
```

4. Allow a user to change their existing password.

```
alter user with
    oldpassword='myoldpassword',
    password='mypassword';
```

5. Allow a user with maintain_users privilege to change or remove any password.

```
alter user username
    with password='theirpassword'
    | nopassword
```

6. Grant createdb privilege to user bspring.

```
alter user bspring add privileges ( createdb )
```

7. Specify a profile for a particular user
```
alter user bspring with profile = dbop
```

   where "dbop" is an existing profile.

8. Specify that a user has an externally verified password.
```
alter user bspring
    with external_password;
```

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.

Permissions

This statement is available to all users.

Related Statements

Declare Table (see page 469)

End Declare Section (see page 502)

Include (see page 560)

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.

Permissions

This statement is available to all users.

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

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. The cursor name cannot exceed 32 characters.

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.

Permissions

This statement is available to all users.

Locking

In the CLOSE statement, closing a cursor does not release the locks held by the cursor. (The locks are released when the transaction is completed.)

Related Statements

Declare Cursor (see page 451)

Fetch (see page 523)

Open (see page 601)

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 sql open c1;
loop until no more rows;
exec sql fetch c1
    into :name, :jobid;
print name, jobid;
end loop;

exec sql close c1;
```

Comment On

Valid in: SQL, ESQL

The COMMENT ON statement creates a comment on a table, view, or column.

Syntax

The COMMENT ON statement has the following format:

```
[EXEC SQL] COMMENT ON
            TABLE [schema.]table_name | COLUMN [schema.]table_name.column_name
            IS remark_text
```

table_name

Specifies the table for which the constraint is defined.

remark_text

Defines the text of the comment.

Limits: The maximum length for a comment is 1600 characters.

Description

The COMMENT ON statement stores comments about a table, view, or column.

To display the comments, use the HELP COMMENT statement.

To delete the comments, issue the COMMENT ON statement and specify an empty string (' '). Comments on tables and views are deleted when the table or view is dropped.

Embedded Usage

You cannot use host language variables in an embedded COMMENT ON statement.

Permissions

You can only create comments on tables or views that you own.

Locking

The COMMENT ON statement locks the iidbms_comment system catalog and takes an exclusive lock on the table on which the comment is being created.

Related Statements

Help (see page 551)

Examples: Comment On

The following examples store comments about a table:

1. Create a comment on the authors table.

```
comment on table authors is
    'It was the best of times, it was the worst
    of times. It was...'
```

2. Delete comments on the authors table.

```
comment on table authors is '';
```

3. Comment on column, name, in the authors table.

```
comment on column authors.name is 'Call me Ishmael';
```

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.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

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.

Permissions

This statement is available to all users.

Locking

All locks acquired during the transaction are released in the CLOSE statement.

Performance

Issuing multiple updates inside a single transaction is generally faster than committing each update individually.

Related Statements

Rollback (see page 636)

Savepoint (see page 640)

Set (see page 671)

Example: Commit

The following embedded example issues two updates, each in its own transaction:

```
exec sql connect 'personnel';

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
              [AS connection_name]
              [SESSION session_number]
              [IDENTIFIED BY username]
              [DBMS_PASSWORD = dbms_password]
              [OPTIONS = flag {, flag}]
              [WITH HIGHDXID = value, LOWDXID = value]
```

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.

connection_name

Specifies an alphanumeric identifier to be associated with the session. The connection name must be a string of up to 128 characters that identifies the session. If the as *connection_name* clause and the session clause are omitted, the default connection name is the specified database name.

Connection_name must be specified using a quoted string literal or a host language variable.

session_number

Specifies a numeric identifier to be associated with the session. The session number must be a positive integer literal or variable, and must be unique among existing session numbers in the application.

username

Specifies the user identifier under which this session runs. *Username* can be specified using a quoted or unquoted string literal or string variable.

dbms_password

Specifies the valid password either as string constant or a string program variable. This parameter allows the application to specify the password at connection time if required.

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 maximum number of flags is 12.

If the -R flag is specified and the role ID has a password, use the following format:
`'-Rroleid/password '`

The flags can be specified using quoted or unquoted character string literals or string variables.

value

Highdxid specifies the high-order 4 bytes of a distributed transaction ID. Lowdxid specifies the low-order 4 bytes of a distributed transaction ID. These options are used for two phase commit of distributed transactions. For details, see the chapter "Working with Transactions and Handling Errors."

Description

The CONNECT statement connects the application to a database and, optionally, to a specified distributed transaction. 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 SQL statement. To terminate a connection, use the DISCONNECT statement.

Connecting with Distributed Transactions

To connect to a specified database and the local transaction associated with a distributed transaction, include the WITH clause. In a two-phase commit application, this option allows a coordinator application to re-establish a connection that was unintentionally severed due to software or hardware problems.

The distributed transaction is identified by its distributed transaction ID, an 8-byte integer that is specified by the application. In the WITH clause, the value specified for highdxid must be the high-order 4 bytes of this ID and the value specified for lowdxid must be the low-order 4 bytes of the distributed transaction ID. The distributed transaction ID must have been previously specified in a prepare to commit statement.

When the program issues a CONNECT statement that includes the WITH clause, a commit or a rollback statement must immediately follow the CONNECT statement. Commit commits the open local transaction, and rollback aborts it. For more information about distributed transactions, see the chapter "Transactions and Error Handling."

Creating Multiple Sessions

If your application requires more than one connection to a database, a session identifier or number can be assigned to each session, and the SET CONNECTION or SET_SQL (SESSION) statements can be used to switch sessions.

Using Session Identifiers

To assign a numeric session identifier to a connection, specify the session clause. For example:

```
exec sql connect accounting session 99;
```

assigns the numeric session identifier 99 to the connection to the accounting database. To determine the session identifier for the current session, use the INQUIRE_SQL(SESSION) statement.

To switch sessions using the numeric session identifier, use the SET_SQL(SESSION) statement. For example:

```
exec sql set_sql(session = 99);
```

Using Connection Names

To assign a name to a connection, specify the as clause. For example:

```
exec sql connect act107b as accounting;
```

assigns the name, accounting, to the connection to the act107b database. To switch sessions using the connection name, use the set connection statement. For example:

```
exec sql set connection accounting;
```

If the AS clause is omitted, the DBMS Server assigns a default connection name-the database specified in the CONNECT statement. This connection name can be used in subsequent set connection statements to switch sessions. If the AS clause is omitted and a numeric session identifier is specified (using the SESSION clause), the default connection name is "ii$n$," where n is the specified numeric session identifier.

To determine the connection name for the current session, use the INQUIRE_SQL(connection_name) statement.

Permissions

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

Locking

The CONNECT statement takes a database lock on the specified database. Unless an exclusive lock using the -l flag is explicitly requested, the database lock is a shared lock.

Related Statements

Set (see page 671)

Disconnect (see page 480)

Examples: Connect

The following examples connect a coordinator application to a database and, optionally, to a specified distributed transaction:

1. Connect to the master database with the current user ID, specifying both a numeric identifier and a connection name, locking the database for exclusive use.

```
exec sql connect 'masterdb'
    as master_database
    identified by :user_id
    options = '-l';
```

2. Connect to a database passed as a parameter in a character string variable.

```
exec sql connect :dbname;
```

3. Assuming that the connection between the coordinator application and the local DBMS has been broken, use the connect statement to reconnect the application to the specified local transactions associated with a distributed transaction.

```
exec sql begin declare section;
    int        high = 1;
    int        low = 200;
    char        branch1[24] = "annie";
    char        branch2[24] = "annie";
    exec sql end declare section;
    define SF_BRANCH 1
    define BK_BRANCH 2
    define BEFORE_WILLING_COMMIT 1
    define WILLING_COMMIT 2
    int tx_state1 = 1;
    int tx_state2 = 1;

/* Read transaction state information from file */

    read_from_file(&tx_state1, &high, &low, branch1);
    read_from_file(&tx_state2, &high, &low, branch2);

if (tx_state1 equals WILLING_COMMIT and
    tx_state2 equals WILLING_COMMIT) then
    print "Both local transactions are ready to commit."
    print "Re-connect to SF to commit local trx."

    exec sql connect :branch1 session :SF_BRANCH
    with highdxid = :high, lowdxid = :low;

    exec sql commit;

    print "Re-connect to Berkeley to commit local trx."

    exec sql connect :branch2 session :BK_BRANCH
    with highdxid = :high, lowdxid = :low;
    exec sql commit;

else
    print "Not all local trxs are ready to commit."
    print "Rollback all the local transactions."
    print "Re-connect to S.F to rollback the local trx."

    exec sql connect :branch1 session :SF_BRANCH
    with highdxid = :high, lowdxid = :low;

    exec sql rollback;

    print "Re-connect to Berkeley to rollback local trx."
```

```
        exec sql connect :branch2 session :BK_BRANCH
        with highdxid = :high, lowdxid = :low;

        exec sql rollback;

    endif
    print "Distributed transaction complete."
    ...
```

Copy

Valid in: SQL, ESQL, OpenAPI

The COPY statement copies the contents of a table to a data file (COPY INTO) or copies the contents of a file to a table (COPY FROM). For more information on the COPY statement, see Populating Tables in the *Database Administrator Guide.*

Note: In OpenAPI, COPY is supported through API calls.

Syntax

The COPY statement has the following format:

```
[EXEC SQL] COPY [TABLE] [schema.]table_name
            ([column_name = format [WITH NULL [(value)]]
            {, column_name = format [WITH NULL [(value)]]}])
            INTO | FROM 'filename[, type]'
            [with_clause]
```

table_name

Specifies an existing table from which data is read or to which data is written.

column_name

Specifies the column from which data is read or to which data is written.

format

Specifies the format in which a value is stored in the file.

filename

Specifies the file from which data is read or to which data is written.

type

> Specifies the file type: text, binary, or variable. (Optional) On VMS platforms only.

with_clause

> Consists of the word WITH, followed by a comma-separated list of one or more of the following items:
>
> - ON_ERROR = TERMINATE | CONTINUE
>
> - ERROR_COUNT = *n*
>
> - ROLLBACK = ENABLED | DISABLED
>
> - LOG = '*filename*'
>
> The following options are valid for bulk copy operations only. For details about these settings, see Modify (see page 579). The value specified for any of these options becomes the new setting for the table and overrides any previously made settings (either using the MODIFY statement or during a previous copy operation).
>
> - ALLOCATION = *n*
>
> - EXTEND = *n*
>
> - FILLFACTOR=*n* (ISAM, HASH, and BTREE only)
>
> - MINPAGES=*n* (hash only)
>
> - MAXPAGES=*n* (hash only)
>
> - LEAFFILL=*n* (btree only)
>
> - NONLEAFFILL=*n* (btree only)
>
> - ROW_ESTIMATE = *n*

Binary Copying

To copy all rows of a table to a file using the order and format of the columns in the table, omit the column list from the COPY statement. This operation is referred to as a *binary* copy.

For example, to copy the entire employee table into the file, emp_name, issue the following statement:

```
copy table employee () into 'emp_name';
```

Parentheses must be included in the statement, even though no columns are listed. The resulting file contains data stored in proprietary binary formats. To load data from a file that was created by a binary copy (COPY INTO), use a binary copy (COPY FROM).

VMS: Bulk copy always creates a binary file.

Bulk Copying

To improve performance when loading data from a file into a table, use a *bulk copy*. The requirements for performing a bulk copy are:

- The table is not journaled

- The table has no secondary indexes

- For storage structures other than heap, the table is empty and occupies fewer than 18 pages

- The table is not partitioned

If the DBMS Server determines that all these requirements are met, the data is loading using bulk copy. If the requirements are not met, data is loaded using a less rapid technique. For detailed information about bulk copying, see the *Database Administrator Guide.*

row_estimate Option

To specify the estimated number of rows to be copied from a file to a table during a bulk copy operation, use the row_estimate option. The DBMS Server uses the specified value to allocate memory for sorting rows before inserting them into the table. An accurate estimate can enhance the performance of the copy operation.

The estimated number of rows must be no less than 0 and no greater than 2,147,483,647. If this parameter is omitted, the default value is 0, in which case the DBMS Server makes its own estimates for disk and memory requirements.

Data File Format versus Table Format

Table columns need not be the same data type or length as their corresponding entries in the data file. For example, numeric data from a table can be stored in char(0) or varchar(0) fields in a data file. The COPY statement converts data types as necessary. When converting data types (except character to character), COPY checks for overflow. When converting from character to character, COPY pads character strings with blanks or nulls, or truncates strings from the right, as necessary.

When copying from a table to a file, specify the column names in the order the values are to be written to the file. The order of the columns in the data file can be different from the order of columns in the table. When copying from a file to a table, specify the table columns in sequence, according to the order of the fields in the data file.

Note: If II_DECIMAL is set to comma, be sure that when SQL syntax requires a comma (such as 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 ti:
```

Column Formats

The following sections describe how to specify the data file format for table columns. The format specifies how each is stored and delimited in the data file.

Note: When copying to or from a table that includes long varchar or long byte columns, specify the columns in the order they appear in the table.

Storage Format

This section describes specifying the format of fields in the data file. When specifying storage formats for COPY INTO, be aware of the following points:

- Data from numeric columns, when written to text fields in the data file, is right-justified and filled with blanks on the left.

- When copying data from a floating-point table column to a text field in a data file, the format the data according to the options specified by the -i and -f flags. For information about these flags, see the sql command description in the *System Administrator Guide*.

- To avoid rounding of large floating point values, use the sql command -f flag to specify a floating point format that correctly accommodates the largest value to be copied. For information about this flag, see the sql command description in the *System Administrator Guide*.

The following table explains the data file formats for the various SQL data types. Delimiters are described in the section following this table.

Format	How Stored (COPY INTO)	How Read (COPY FROM)
Byte(0)	Stored as fixed-length binary data (padded with zeros to the declared length if necessary).	Read as variable-length binary data terminated by the first comma, tab, or newline encountered.
Byte(0)*delim*	Stored as fixed-length binary data (padded with zeros to the declared length if necessary). The one-character delimiter is inserted immediately after the value. Because this format uses zeros to pad data, a zero is not a valid delimiter for this format.	Read as variable-length binary data terminated by the specified character.
Byte(*n*) where *n* is 1 to the maximum row size configured, not exceeding 32,000.	Stored as fixed-length binary data.	Read as fixed-length binary data.
Byte varying(0)	Stored as variable-length binary data preceded by a 5-character, right-justified length specifier.	Read as variable-length binary data, preceded by a 5-character, right-justified length specifier.

Format	How Stored (COPY INTO)	How Read (COPY FROM)
byte varying(*n*) where *n* is 1 to the maximum row size configured, not exceeding 32,000.	Stored as fixed-length binary data preceded by a 5-character, right-justified length specifier. If necessary, the value is padded with zeros to the specified length.	Read as fixed-length binary data, preceded by a 5-character, right-justified length specifier.
Char(0)	Stored as fixed-length strings (padded with blanks if necessary). For character data, the length of the string written to the file is the same as the column length.	Read as variable-length character string terminated by the first comma, tab, or newline encountered.
Char(0)*delim*	Stored padded to the declared width of the column. The one-character delimiter is inserted immediately after the value. Because this format uses spaces to pad data, a space (sp) is not a valid delimiter for this format.	Read as variable-length character string terminated by the specified character.
char(*n*) where *n* is 1 to the maximum row size configured, not exceeding 32,000 (16,000 in a UTF8 instance).	Stored as fixed-length strings.	Read as fixed-length string.
D0	(not applicable)	Dummy field. Read as a variable-length character string terminated by the first comma, tab, or newline encountered. The data in the field is skipped.

Format	How Stored (COPY INTO)	How Read (COPY FROM)
D0*delim*	Indicates a delimited dummy column. Instead of placing a value in the file, COPY inserts the specified delim. (Unlike the dn format, this format does not insert the column name.)	Dummy field. Read as a variable-length character string delimited by the specified character. The data in the field is skipped.
Date	Stored in date format.	Read as a date field.
decimal	Stored in decimal data format.	Read as decimal data.
D*n*	Dummy column. Instead of placing a value in the file, COPY inserts the name of the column n times. For example, if you specify x=d1, the column name, x, is inserted once; if you specify x=d2, COPY inserts the column name, x, twice, and so on. You can specify a delimiter as a column name, for example, nl=d1.	Dummy field, read as a variable-length character string of the specified length. The data in the field is skipped.
Float	Stored as double-precision floating point.	Read as double-precision floating point.
Float4	Stored as single-precision floating point.	Read as single-precision floating point.
integer	Stored as integer of 4-byte length.	Read as integer of 4-byte length.
integer1	Stored as integer of 1-byte length.	Read as integer of 1-byte length.

Format	How Stored (COPY INTO)	How Read (COPY FROM)
Long byte(0)	Binary data stored in segments, and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of characters. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The following example shows two data segments, followed by the termination zero length segment. The first segment is 5 characters long, the second segment is 10 characters long, and the termination segment is 0 character long. The maximum length of each segment is 32737.	Read under the same format as COPY INTO.
	5 abcde10 abcdefghij 0 (with a space after the terminating 0 character)	
	(In this example, the effective data that was in the column is abcdeabcdefghij)	
	If the long byte column is nullable, specify the with null clause. An empty column is stored as an integer 0, followed by a space.	

Format	How Stored (COPY INTO)	How Read (COPY FROM)
Long nvarchar(0)	Stored in segments, and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of Unicode characters in UTF-8 format. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The maximum segment size for the long nvarchar segment is 32727 bytes. The UTF-8 encoded long nvarchar data segments are similar to long varchar data segments. See the description for long varchar(0) for an example of the encoded data segment. If the long nvarchar column is nullable, specify the with null clause. An empty column is stored as an integer 0, followed by a space.	Read under the same format as COPY INTO.

Format	How Stored (COPY INTO)	How Read (COPY FROM)
Long varchar(0)	Stored in segments, and terminated by a zero length segment. Each segment is composed of an integer specifying the length of the segment, followed by a space and the specified number of characters. The end of the column data is specified through a termination, zero length segment (that is, an integer 0 followed by a space). The following example shows two data segments, followed by the termination zero length segment. The first segment is 5 characters long, the second segment is 10 characters long, and the termination segment is 0 character long. The maximum length of each segment is 32737.

5 abcde10 abcdefghij 0 (with a space after the terminating 0 character)

(In this example, the effective data that was in the column is abcdeabcdefghij)

If the long varchar column is nullable, specify the with null clause. An empty column is stored as an integer 0, followed by a space. | Read under the same format as COPY INTO. |
| money | Stored in money format. | Read as a money field. |

Format	How Stored (COPY INTO)	How Read (COPY FROM)
nchar(0)	Stored as fixed-length Unicode strings in UTF-8 format (padded with blanks if necessary).	Read as fixed-length Unicode string in UTF-8 format but converted to UTF-16 for storage. The string is terminated by the first comma, tab, or newline encountered.
nvarchar(0)	Stored as a variable-length Unicode string in UTF-8 format preceded by a 5-character, right-justified length specifier.	Read as variable-length Unicode string in UTF-8 format, preceded by a 5-character, right-justified length specifier.
smallint	Stored as integer of 2-byte length.	Read as integer of 2-byte length.
varchar(0)	Stored as a variable-length string preceded by a 5-character, right-justified length specifier.	Read as variable-length string, preceded by a 5-character, right-justified length specifier.
varchar(*n*) where *n* is 1 to the maximum row size configured, not exceeding 32,000 (16,000 in a UTF8 instance).	Stored as fixed-length strings preceded by a 5-character, right-justified length specifier. If necessary, the value is padded with null characters to the specified length.	Read as fixed-length string, preceded by a 5-character, right-justified length specifier.

Note: The dummy format (d*n*) behaves differently for COPY FROM and COPY INTO. When a table is copied into a file, *n* specifies the number of times the column name is repeated. When copying from a file to a table, *n* specifies the number of bytes to skip.

For user-defined data types (UDTs), use char or varchar.

Delimiters

Delimiters are those characters in the data file that separate fields and mark the end of records. Valid delimiters are listed in the following table:

Delimiter	Description
Nl	newline character

Delimiter	Description
Tab	tab character
Sp	Space
nul or null	null character
comma	Comma
colon	Colon
Dash	Dash
lparen	left parenthesis
rparen	right parenthesis
X	any non-numeric character

When a single character is specified as the delimiter, enclose that character in quotes. If the data type specification is d0, the quotes must enclose the entire format. For example, 'd0%' specifies a dummy column delimited by a percent sign (%).

If the data type specification is char(0) or varchar(0), only the delimiter character must be quoted. For example, char(0)'%' specifies a char field delimited by a percent sign.

Do not use the space delimiter (sp) with char(0) fields: the char(0) format uses spaces as padding for character and numeric columns.

When copying from a table into a file, insert delimiters independently of columns. For example, to insert a newline character at the end of a line, specify 'nl=d1' at the end of the column list. This directs the DBMS Server to add one (d1) newline (nl) character. (Do not confuse lowercase 'l' with the number '1'.)

With Null Clause

When copying data from a table to a file, the WITH NULL clause directs COPY to put the specified value in the file when a null value is detected in the corresponding column. Specify the WITH NULL clause for any column that is nullable. If the WITH NULL clause is omitted, the DBMS Server returns an error when it encounters null data, and aborts the COPY statement.

When copying data from a file to a table, the WITH NULL clause specifies the value in the file to be interpreted as a null. When COPY encounters this value in the file, it writes a null to the corresponding table column. The table column must be nullable.

To prevent conflicts between valid data and null entries, choose a value that does not occur as part of the data in your table. The value chosen to represent nulls must be compatible with the format of the field in the file: character formats require quoted values and numeric formats require unquoted numeric values. For example:

This example of a value is incorrect:

```
c0comma with null(0)
```

because the value specified for nulls (numeric zero) conflicts with the character data type of the field. However, this example is correct:

```
c0comma with null('0')
```

because the null value is character data, specified in quotes, and does not conflict with the data type of the field. Do not use the keyword null, quoted or unquoted, for a numeric format.

When copying from a table to a file, be sure that the specified field format is at least as large as the value specified for the with null clause. If the column format is too small, the DBMS Server truncates the null value written to the data file to fit the specified format.

For example, in the following statement the string, 'NULL,' is truncated to 'N' because the format is incorrectly specified as one character:

```
copy table t1 (col1 = varchar(1) with null ('NULL')) into 't1.dat';
```

The correct version specifies a 4-character format for the column.

```
copy table t1 (col1 = varchar(4) with null ('NULL')) into 't1.dat';
```

If with null is specified but *value* is omitted, COPY appends a trailing byte indicating whether the field is null. For null fields, COPY inserts an undefined data value in place of the null and sets the trailing byte to indicate a null field. *Value* must be specified for nullable char(0) and varchar(0) columns.

Filename Specification

Filename must be enclosed in single quotation marks; the file specification can include a directory/path name. For COPY INTO, if the file does not exist, COPY creates the file.

UNIX: For COPY INTO, if the file already exists, COPY overwrites it.

VMS: For COPY INTO, if the file already exists, COPY creates another version of the file.

VMS File Types

File type can be specified using the optional type parameter. *Type* must be one of the values listed in the following table.

Type	Record Format	Record Attributes
Text	Variable length	Records delimited by carriage return
binary	Fixed length	None
variable	Variable length	None

If type is omitted, COPY determines the file type as follows:

- If all fields in the file are character types (char, varchar), and all records end in <newline>, COPY creates a text file.

- If the file contains variable length records, its file type is variable. Variable length records occur if one or more fields are stored as varchar(0).

- If none of the preceding conditions apply, COPY creates a binary file.

If type is specified, the contents of the file must be in accordance with these rules. If it is not, COPY creates the data file according to the preceding rules.

With Clause Options

Valid WITH clause options for the COPY statement are as follows:

ON_ERROR = TERMINATE | CONTINUE

Directs COPY to continue after encountering conversion errors.

To direct copy to continue until a specified number of conversion errors have occurred, specify the ERROR_COUNT option instead.

By default, COPY terminates when an error occurs while converting a table row into file format.

When ON_ERROR is set to CONTINUE, COPY displays a warning whenever a conversion error occurs, skips the row that caused the error, and continues processing the remaining rows. At the end of the processing, COPY displays a message that indicates how many warnings were issued and how many rows were successfully copied.

Setting ON_ERROR to CONTINUE does not affect how COPY responds to errors other than conversion errors. Any other error, such as an error writing the file, terminates the COPY operation.

ERROR_COUNT = *n*

Specifies how many errors can occur before processing terminates.

Default: 1.

If ON_ERROR is set to continue, setting ERROR_COUNT has no effect.

LOG = '*filename*'

Stores to a file any rows that COPY cannot process. This option can be used only if ON_ERROR CONTINUE is specified. When specified with log, COPY places any rows that it cannot process into the specified log file. The rows in the log file are in the same format as the rows in the database.

Logging works as follows:

Windows: COPY opens the log file prior to the start of data transfer. If it cannot open the log file, copy halts. If an error occurs when writing to the log file, copy issues a warning, but continues. If the specified log file already exists, it is overwritten with the new values (or truncated if the copy operation encounters no bad rows).

UNIX: COPY opens the log file prior to the start of data transfer. If it cannot open the log file, copy halts. If an error occurs when writing to the log file, copy issues a warning, but continues. If the specified log file already exists, it is overwritten with the new values (or truncated if the copy operation encounters no bad rows).

VMS: COPY attempts to open the log file prior to the start of data transfer. If it cannot open the log file, copy halts. If an error occurs when writing to the log file, copy issues a warning, but continues. If the log file already exists, copy creates a new version.

If copying from a data file that contains duplicate rows (or rows that duplicate rows already in the table) to a table that has a unique key, copy displays a warning message and does not add the duplicate rows. If the WITH LOG option is specified, COPY does not write the duplicate rows to the log file.

ROLLBACK = ENABLED | DISABLED

Enables or disables rollback, as follows:

ENABLED

Directs the DBMS Server to back out all rows appended by the copy if the copy is terminated due to an error.

DISABLED

Retains the appended rows.

The ROLLBACK=DISABLED option does not mean that a transaction cannot be rolled back. Database server errors that indicate data corruption still causes rollback, and rows are not committed until the transaction is complete.

Default: ENABLED

When copying to a file, the WITH ROLLBACK clause has no effect.

Permissions

One of the following must apply:

- You own the table.

- The table has select (for COPY INTO) or insert (for COPY FROM) permission granted to public.

- You have been granted copy_into (for COPY INTO) or copy_from (for COPY FROM) privileges on the table.

Locking

- When copying from a table into a file, the DBMS Server takes a shared lock on the table.

- When performing a bulk copy into a table, the DBMS Server takes an exclusive lock on the table. Because bulk copy cannot start until it gets an exclusive lock, this operation can be delayed due to lock contention.

- When performing a non-bulk copy into a table, the DBMS server takes an "intended exclusive" lock on the table, and uses insert to update the table. As a result, the operation can be aborted due to deadlock.

Restrictions and Considerations

COPY cannot be used to add data to a view, index, or system catalog.

When copying data into a table, COPY ignores any integrity constraints defined (using the CREATE INTEGRITY statement) against the table.

When copying data into a table, COPY ignores ANSI/ISO Entry SQL-92 check and referential constraints (defined using the CREATE TABLE and ALTER TABLE statements), but does not ignore unique (and primary key) constraints.

The COPY statement does not fire any rules defined against the table.

Values cannot be assigned to SYSTEM_MAINTAINED logical key columns. The DBMS Server assigns values when copying from a data file to a table.

Related Statements

Create Table (see page 402)

Modify (see page 579)

Example: Copy

The following examples illustrate the correct use of the COPY statement:

1. In the following Data File Format example, the contents of the file,
    emp.txt, are copied into the employee table. To omit the city column, a
    dummy column is employed. The format of the employee table is as
    follows:

    ```
    ename      char(15)
        age        integer4
        dept       char(10)
        comment    varchar(20)
    ```

    The emp.txt file contains the following data:

    ```
    Jones,J.  32  Anytown,USA  toy,00017A comment

    Smith,P.  41  New York,NY  admin,00015 Another comment
    ```

    The following diagram illustrates the COPY statement that copies the file,
    emp.txt, into the employee table, and maps the fields in the file to the
    portions of the statement that specify how the field is to be copied. Note
    the following points:

    A dummy column is used to skip the city and state field in the data file,
    because there is no matching column in the employee table.

    The department field is delimited by a comma.

    The comment field is a variable-length varchar field, preceded by a five-
    character length specifier.

    ```
    Jones,J.^^^^^ 32  ^^^Anytown,USA^^^  toy,   00017This is a comment
    Smith,P.^^^^^ 41  ^^^New York,NY^^^  admin,   00015Another comment


                      copy table employee

                      (

                      ename=char(12),

                      age=char(3),

                      xxx=d17,

                      dept=char(0) comma,

                      comment=varchar (0) nl

                      )

                      from 'emp.txt'
    ```
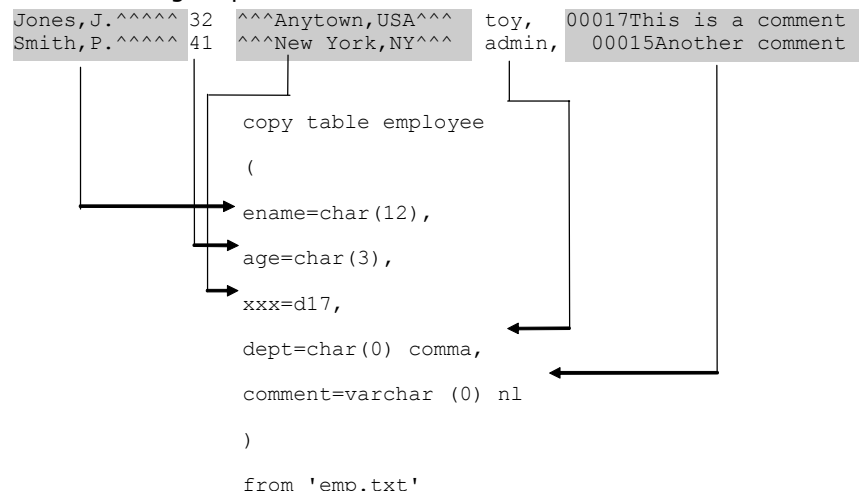
2. Load the employee table from a data file. The data file contains binary
    data (rather than character data that can be changed using a text editor).

    ```
    copy table employee (eno=integer2, ename=char(10),
        age=integer2, job=integer2, sal=float4,
        dept=integer2, xxx=d1)
        from 'myfile.in';
    ```

3. Copy data from the employee table into a file. The example copies
    employee names, employee numbers, and salaries into a file, inserting
    commas and newline characters so that the file can be printed or edited.
    All items are stored as character data. The sal column is converted from its
    table format (money) to ASCII characters in the data file.

    ```
    copy table employee (ename=char(0)tab,
        eno=char(0)tab, sal= char(0)nl)
        into 'mfile.out';
    ```

    ```
    Joe Smith          ,           101,          $25000.00

    Shirley Scott      ,           102,          $30000.00
    ```

4. Bulk copy the employee table into a file. The resulting data file contains
    binary data.

    ```
    copy table employee () into 'ourfile.dat';
    ```

5. Bulk copy from the file created in the preceding example.

    ```
    copy table other_employee_table () from 'ourfile.dat';
    ```

6. Copy the acct_recv table into a file. The following statement skips the
    address column, uses the percent sign (%) as a field delimiter, uses 'xx' to
    indicate null debit and credit fields, and inserts a newline at the end of
    each record.

    ```
    copy table acct_recv
        (acct_name=char(0)'%',
        address='d0%',
        credit=char(0)'%' with null('xx'),
        debit=char(0)'%' with null('xx'),
        acct_mngr=char(15),
        nl=d1)
        into 'qtr_result';
    ```

    ```
    Smith Corp%%      $12345.00%      $-67890.00%Jones

    ABC Oil   %%      $54321.00%      $-98765.00%Green

    Spring Omc%%xx            %xx              %Namroc
    ```

7. Copy a table called, gifts, to a file for archiving. This table contains a record of all non-monetary gifts received by a charity foundation. The columns in the table contain the name of the item, when it was received, and who sent it. Because givers are often anonymous, the column representing the sender is nullable.

```
copy table gifts
    (item_name=char(0)tab,
    date_recd=char(0)tab,
    sender=char(20)nl with null('anonymous'))
    into 'giftdata';
toaster    04-mar-1993              Nicholas
sled       10-oct-1993              anonymous
rocket     01-dec-1993              Francisco
```

8. Create a table and load it using bulk copy, specifying structural options.

```
create table mytable (name char 25, ...);

modify mytable to hash;

copy mytable() from 'myfile' with minpages = 16384,
maxpages = 16384, allocation = 16384;
```

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.

Permissions

This statement is available to all users.

Locking

The CREATE DBEVENT statement locks pages in the iievent catalog.

Related Statements

Create Group

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE GROUP statement establishes a group identifier and associates it with the specified list of users. Group identifiers enable the database administrator (or user that has the security privilege) to grant identical privileges to a group of users. For a complete discussion of group identifiers and their use, see the *Database Administrator Guide.*

After creating a group identifier and specifying its members, the system administrator can grant privileges to the group identifier. When a member of the group begins a session, the group identifier can be specified in the SQL or CONNECT statement (or on the operating system command line, using the -G flag) to obtain the privileges associated with the group.

Syntax

The CREATE GROUP statement has the following format:

```
[EXEC SQL] CREATE GROUP group_id {, group_id}
            [WITH USERS = (user_id {, user_id})]
```

group id

Is the group identifier. It must be a valid object name that is unique among all user, group, and role identifiers in the installation. If an invalid identifier is specified in the list of group identifiers, the DBMS Server returns an error but processes all valid group identifiers. Group identifier names are stored in the iiusergroup catalog in the iidbdb database.

user id

Must be a valid user name. If an invalid user identifier is specified, the DBMS Server issues an error but processes all valid user identifiers. A group can contain any number of users. A group identifier can be created without specifying a user list. To add users to an existing group identifier, use the ALTER GROUP statement.

Permissions

You must have maintain_users privileges and be working in a session connected with the iidbdb.

Embedded Usage

In an embedded CREATE GROUP statement, neither *group_id* nor *user_id* can be specified using host language variables.

Locking

The CREATE GROUP statement locks pages in the iiusergroup catalog in the iidbdb. This can cause sessions attempting to connect to the server to be suspended until the CREATE GROUP statement is completed.

Related Statements

Alter Group (see page 290)

Drop Group (see page 486)

Examples: Create Group

The following are CREATE GROUP statement examples:

1. Create a group identifier for the telephone sales force of a company and put the user IDs of the salespeople in the user list of the group.

```
create group tel_sales with users = (harryk,
    joanb, jerryw, arlenep);
```

2. In an application, create a group identifier for the inventory clerks of a store and place their user IDs in the user list of the group.

```
exec sql create group inv_clerk with users =
    (jeanies, louisem, joep);
```

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 [schema.]table_name
          (column_name {, column_name})[UNIQUE]
          [WITH with_clause]
```

To build a set of indexes on the same table in parallel:

```
[EXEC SQL] CREATE [UNIQUE] INDEX [schema.]index_name
          ON [schema.] table_name
          (column_name [ASC|DESC]{, column_name..})[UNIQUE]
          [WITH with_clause]){, ([schema.]index_name }
```

or
```
[EXEC SQL] CREATE [UNIQUE] INDEX ([schema.] index_name
          ON table_name
          (column_name [ASC|DESC]{, column_name..})
          [UNIQUE]) {,([schema.]index_name…}
          [WITH with_clause]
```

Note: When using parallel index syntax, concurrent access is not allowed on readonly tables.

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. If the key option is used, the columns specified as keys must head this list and must appear in the same order in which they are specified in the key option. If the structure is rtree, only one column can be named.

STRUCTURE = BTREE | ISAM | HASH | RTREE

Specifies the storage structure of the index. Defaults to isam if the parameter is not included. If the structure is rtree, unique cannot be specified.

Default: isam

WITH *with_clause*

Specifies a comma-separated list of any of the following items:

KEY = (*columnlist***)**

Specifies the columns on which the index is keyed. If this parameter is not included, the index is keyed on the columns in the index definition. If the structure is rtree, only one column can be named.

FILLFACTOR = *n*

Specifies the percentage of each primary data page that can be filled with rows.

Limits: 1 to 100 and must be expressed as an integer literal or integer variable.

Default: Default values differ for each storage structure.

MINPAGES = *n*

Defines the minimum number of primary pages a hash or compressed hash index table must have. The value can be expressed as an integer literal or integer variable.

Default: 16 for a hash table; 1 for a compressed hash table.

MAXPAGES = *n*

Defines the maximum number of primary pages that a hash or compressed hash index can have. The value can be expressed as an integer literal or integer variable.

Default: No default

LEAFFILL = *n*

Defines the percentage full each leaf index page is when the index is created. This option can be used when creating an index with a btree or compressed btree structure.

Limits: 1 to 100 and must be an integer literal or integer variable.

NONLEAFFILL = *n*

Specifies the percentage full each nonleaf index page is when the index is created. This option can be used when creating an index with a btree or compressed btree structure.

Limits: 1 to 100, and must be an integer literal or integer variable.

LOCATION = (*location_name* **{,** *location_name***})**

Specifies the areas on which the index is created. Location_name must be a string literal or string variable.

Default: The default area for the database

ALLOCATION = *n*

Specifies the number of pages initially allocated for the index.

Limits: An integer between 4 and 8,388,607

Default: 4

EXTEND = *n*

Specifies the number of pages by which the index is extended when more space is required.

Limits: An integer between 1and 8,388,607

Default: 16

COMPRESSION [= ([[NO]KEY] [,[NO|HI]DATA])] | NOCOMPRESSION

Specifies whether the index key and data are to be compressed. If the structure is RTREE, compression cannot be specified.

Default: NOCOMPRESSION

[NO]PERSISTENCE

Specifies whether the modify statement recreates the index when its base table is modified.

Default: nopersistence (indexes are not recreated).

UNIQUE_SCOPE = STATEMENT | ROW

For unique indexes only. Specifies whether rows are checked for uniqueness one-by-one as they are inserted or after the update is complete. If the structure is rtree, unique_scope cannot be specified.

Default: unique_scope = row

RANGE = ((min_x, min_y), (max_x, max_y))

For RTREE indexes only. Specifies the minimum and maximum values of the index column.

Limits: The values must have the same data type as the index column, either integer4 or float8. The RANGE parameter must be specified if the structure is RTREE.

PAGE_SIZE = *n*

Specifies page size.

PRIORITY = *cache_priority*

Allows tables to be assigned fixed priorities

Limits: An integer between 0 and 8

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:
```

Description

The CREATE INDEX statement creates an index on an existing base table. The index contains the columns specified. Any number of indexes can be created for a table, but each index can contain no more than 32 columns. The contents of indexes are sorted in ascending order by default.

Indexes can improve query processing. To obtain the greatest benefit, create indexes that contain all of the columns that are generally queried. The index must be keyed on a subset of those columns.

By default, the index is keyed on the columns in the column list, in the order they are specified. If the key option is specified, the index is keyed on the columns specified in the key list, in the order specified. For example, if you issue the statement:

```
create index nameidx on employee
        (last, first, phone);
```

you create an index called, nameidx, on the employee table that is keyed on the columns last, first, and phone in that order.

However, if you issue the statement:

```
create index nameidx on employee
(last, first, phone)
        with key = (last, first);
```

the index is keyed only on the two columns, last and first.

The columns specified in the key column list must be a subset of the columns in the main column list. A long varchar column cannot be specified as part of a key.

Index Storage Structure

By default, indexes are created with an ISAM storage structure. There are two methods to override this default:

- To specify the default index storage structure for indexes created during the session, use the -n flag when issuing the command that opens the session (SQL, ISQL, or CONNECT). For more information about this flag, see the *System Administrator Guide.*

- To override the session default when creating an index, specify the desired storage structure using the STRUCTURE option when issuing the CREATE INDEX statement.

To specify whether the index is to be compressed, use the WITH [NO]COMPRESSION clause. By default, indexes are not compressed. If with compression is specified, the STRUCTURE clause must be specified. An RTREE index cannot be compressed. To change the storage structure of an index, use the MODIFY statement. For details about table storage structures, see Modify (see page 579).

Unique Indexes

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 key fields of the index, the CREATE INDEX statement fails. Similarly, if an insert or update is attempted that violates the uniqueness constraint of an index created on the table, the insert or update fails. This is true for an UPDATE statement that updates multiple rows: the UPDATE statement fails when it attempts to write a row that violates the uniqueness constraint.

Effect of the Unique_Scope Option on Updates

The UNIQUE_SCOPE option can affect the outcome of an update. For example, suppose you create an index on the employee numbers in an employee table, and the table contains employee numbers in sequence from 1 to 1000. If you issue an UPDATE statement that increments all employee numbers by 1, uniqueness is checked according to the UNIQUE_SCOPE option as follows:

- **UNIQUE_SCOPE = ROW** - Employee number 1 is incremented to 2. The row is checked for uniqueness-of course, employee number 2 already exists. Result: the update fails.

- **UNIQUE_SCOPE = STATEMENT** - Employees 1 through 1000 are incremented before uniqueness is checked. All employee numbers remain unique. Result: the update succeeds.

Index Location

Location_name refers to the areas where the new index is created. The *location_names* must be defined on the system, and the database must have been extended to the corresponding areas. If no *location_name* is specified, the index is created in the default database area. If multiple *location_names* are specified, the index is physically partitioned across the locations. For more information about creating locations and extending databases, see the *Database Administrator Guide.*

Parallel Index Building

Use parallel index to more efficiently create indexes in parallel. Each of these indexes can also be marked as persistent, which means that if the underlying base structure of the table is reorganized (or modified), the indexes are recreated automatically.

Note: UNIQUE cannot be specified before both the INDEX keyword and with an individual index specification. If UNIQUE is used before INDEX, all the indexes being created are unique indexes. See the Examples (see page 360) later in this section.

Embedded Usage

In an embedded CREATE INDEX statement, the following elements can be replaced with host language variables:

Elements	Description
location_name	Specifies the location of the index; must be a string variable.
N	Specifies fill and page values; must be an integer variable.

The preprocessor does not validate the WITH clause syntax. The WITH clause can be specified using a host string variable (with :*hostvar*).

Permissions

You must be the owner of a table. Users cannot update indexes directly. When a table is changed, the DBMS Server updates indexes as required. To create indexes on system tables, the effective user of the session must be $ingres.

Locking

Creating an index on a table requires an exclusive lock on the table. This lock prevents other sessions, even those using the READLOCK=NOLOCK option, from accessing the table until CREATE INDEX completes and the transaction containing it is completed.

Related Statements

Create Table (see page 402)

Drop (see page 482)

Modify (see page 579)

Examples: Create Index

The following are CREATE INDEX statement examples:

1. Create an index for the columns, ename and age, on the employee table. The index is recreated when the table is modified.

```
create index ename on employee (ename, age)
    with persistence;
```

2. Create an index called ename and locate it on the area referred to by the location_name, remote.

```
create index ename on employee (ename, age)
    with location = (remote);
```

3. Create a B-tree index on the ename and eage columns, keyed on ename with leaf index pages filled 50 percent.

```
create index ename2 on employee (ename, eage)
    with key = (ename),
    structure = btree,
    leaffill = 50;
```

4. Create a unique index, specifying that uniqueness is checked after any UPDATE statements are completed.

```
create unique index ename3 on employee (ename, empno)
    with key = (ename, empno),
    unique_scope = statement;
```

5. Create a single, unique index on column c1 in table tl.

```
create index i1 on t1 (c1) unique
```

6. Create a unique index using the WITH clause to override the default index structure.

```
create unique index (i1 on t1(c1) with structure=hash)
```

7. Create multiple indexes at one time using the UNIQUE qualifier before the INDEX keyword. All indexes created by this statement are unique.

```
create unique index (i1 on t1(c1) with structure=hash,persistence),
    (i2 on t1(c2) with structure=btree)
```

8. Create multiple indexes at one time using the UNIQUE keyword within each index specification.

```
create index (i1 on t1 (c1) unique with structure=hash,persistence), (i2 on
    t1(c2) unique with structure=btree)
```

9. Create both unique and non-unique indexes.

```
create index (i1 on t1(c1) unique with structure=hash,persistence),
            (i2 on t1(c2) with structure=btree)
```

Note: Examples 7 and 8 perform the same operation, while 9 demonstrates individual control of the UNIQUE attribute.

Create Integrity

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE INTEGRITY statement creates an integrity constraint for the specified base table.

Syntax

The CREATE INTEGRITY statement has the following format:

```
[EXEC SQL] CREATE INTEGRITY ON table_name [corr_name]
            IS search_condition
```

table_name

Specifies the table for which the constraint is defined.

corr name

Specifies a correlation name (see page 43) for the table for use in the *search condition*.

search condition

Defines the actual constraint. For example, if you want to create a constraint on the employee table so that no employee can have a salary of greater than $75,000, issue the following statement:

```
create integrity on employee is salary <= 75000;
```

The search condition must reference only the table on which the integrity constraint is defined, and cannot contain a subselect or any aggregate (set) functions.

At the time the CREATE INTEGRITY statement is executed, the search condition must be true for every row in the table, or the DBMS Server issues an error and aborts the statement. If the search condition is defined on a column that contains nulls, the statement fails unless the is null predicate is specified in the statement.

After the constraint is defined, all updates to the table must satisfy the specified search condition. Integrity constraints that are violated are not specifically reported: updates and inserts that violate any integrity constraints are simply not performed.

Locking

The CREATE INTEGRITY statement takes an exclusive lock on the specified table.

Performance

The time required to execute the CREATE INTEGRITY statement varies with the size of the table, because the DBMS Server must check the specified base table to ensure that each row initially conforms to the new integrity constraint.

Embedded Usage

In an embedded CREATE INTEGRITY statement, variables can be used to see constant values in the search condition.

Permissions

You must own the table.

Related Statements

Drop Integrity (see page 487)

Examples: Create Integrity

The following are CREATE INTEGRITY statement examples:

1. Make sure that the salaries of all employees are no less than 6000.

    ```
    create integrity on employee is salary >= 6000;
    ```

2. In an embedded application, define an integrity using a host language variable.

    ```
    exec sql create integrity on employee
        is sal < :sal_limit;
    ```

Create Location

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE LOCATION statement assigns a name to a physical disk and directory location.

For detailed information about locations, see the *Database Administrator Guide*. To specify the work (sorting) locations for a session, use the SET WORK LOCATIONS statement.

Syntax

The CREATE LOCATION statement has the following format:

```
[EXEC SQL] CREATE LOCATION location_name
WITH AREA = area_name,
USAGE = (usage_type {, usage_type}) | NOUSAGE
RAWPCT = n
```

location_name

Specifies the name to be assigned to the disk and directory combination. Must be a valid object name.

area_name

Specifies the disk and directory location to which the location is mapped. Must be a valid operating-system specification. This parameter can be specified using a quoted string or an unquoted string that does not include special (non-alphanumeric) characters.

usage_type

Specifies the types of file that can be stored at this location. Valid values are:

DATABASE

WORK

JOURNAL

CHECKPOINT

DUMP

ALL

NOUSAGE

To prevent any files from being stored at the location, specify WITH NOUSAGE.

RAWPCT=n

Defines the relative amount of the area to be allocated to the location. RAWPCT=0 is equivalent to omitting the parameter, resulting in a cooked definition. When RAWPCT is greater than zero, the only valid usage is DATABASE.

Limits: 1 to 100

Embedded Usage

In an embedded CREATE LOCATION statement, the WITH clause can be specified using a host string variable (with :*hostvar*). *Usage_type* and *area_name* can be specified using host string variables.

Permissions

You must have maintain_locations privileges and be working in a session that is connected to the iidbdb.

Locking

The CREATE LOCATION statement locks pages in the iilocation_info catalog.

Related Statements

Alter Location (see page 292)

Drop Location (see page 489)

Grant (privilege) (see page 534)

Modify (see page 579)

Locations can be assigned when creating tables or indexes by using the following statements:

Create Index (see page 353)

Create Table (see page 402)

Declare Global Temporary Table (see page 462)

Modify...to Relocate (see page 590)

Set Work Locations (see page 688)

Examples: Create Location

The following are CREATE LOCATION statement examples:

VMS:

Create a new location for databases; allow all types of files to be stored.

```
create location accounting_db with area = 'disk1:',

usage = (all);
```

Create a new location, but prevent any files from being created there.

```
create location new_db with area = 'disk2:',

nousage;
```

UNIX:

Create a location using a UNIX path.

```
create location extraloc

with area = '/usr/ingres_extra',

usage = journal, checkpoint;
```

Create Procedure

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE PROCEDURE statement creates a database procedure.

Syntax

The CREATE PROCEDURE statement has the following format:

```
[EXEC SQL] [CREATE] PROCEDURE [schema.]proc_name
            [[(set_param_name [=] SET OF]
            ([param_mode] param_name [=] param_type
                    [WITH | NOT DEFAULT] [WITH | NOT NULL]
            {, [param_mode] param_name [=] param_type
                    [WITH | NOT DEFAULT] [WITH | NOT NULL]})[)]]]
            [RESULT ROW (result_type [WITH | NOT DEFAULT] [WITH | NOT NULL]
            {,  result_type [WITH | NOT DEFAULT] [WITH | NOT NULL]}) =|AS
            [declare_section]
BEGIN
            statement {; statement}[;]
END
```

proc name

Defines the name of the procedure. This must be a valid object name.

set_param_name

Defines the name of the set of parameter. This must be a valid object name. The set of parameters are referenced like base tables in the body of the procedure.

param_name

Defines the name of a procedure parameter. This must be a valid object name. Parameters can be passed by value or by reference.

param_mode

Assigns one of the following modes to the procedure parameter:

IN

Declares the parameter as an input only parameter.

OUT

Declares the parameter as an output only parameter.

INOUT

Declares the parameter as one that passes a value into the procedure and returns it, possibly modified, to the calling program.

param_type

Specifies the data type of the associated parameter. The data type can be any legal SQL data type, and the with|not null clause can be part of the specification.

declare_section

A list of local variables for use in the procedure. For details, see Declare (see page 446).

statement

> Local variable assignments and any of the statements listed in the text of the CREATE PROCEDURE description.

result_type

> The data type of the associated entry in a return row statement. The data type can be any legal SQL data type, and the WITH | NOT NULL clause can be part of the specification. For details, see Return Row (see page 629).

> **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:
> ```

Description

The CREATE PROCEDURE statement creates a database procedure that is managed as a named database object by the DBMS Server. A database procedure can be executed directly using the EXECUTE PROCEDURE statement or can be invoked by a rule.

A procedure that is directly executed can contain any of the following statements:

- COMMIT
- DELETE
- ENDLOOP
- EXECUTE PROCEDURE
- FOR
- IF
- INSERT
- MESSAGE
- RAISE DBEVENT
- RAISE ERROR
- REGISTER DBEVENT
- REMOVE DBEVENT

- RETURN

- RETURN ROW

- ROLLBACK

- SELECT

- UPDATE

- WHILE

- Assignment statements

Procedures that are invoked by rules must not issue the COMMIT and ROLLBACK statements, and cannot use the RETURN statement to return values to an application. Procedures invoked by DELETE or UPDATE rules must not reference the old blob column values. Procedures invoked by rules can use the RAISE ERROR statement to signal error conditions.

A procedure cannot contain any data definition statements, such as CREATE TABLE, nor can a procedure create or drop another procedure. Database procedures can execute other database procedures.

The repeated clause cannot be used in a statement in the procedure body. However, database procedures confer the same performance benefits as the repeated clause.

In a procedure, SELECT statements must assign their results to local variables. Also, SELECT statements can return only a single row of data unless they are contained in a FOR statement. If more rows are returned, no error is issued, but only the first row retrieved is in the result variables.

Both procedure parameters and local variables can be used in place of any constant value in statements in the procedure body. The DBMS Server treats procedure parameters as local variables inside the procedure body, although they have an initial value assigned when the procedure is invoked. Preceding colons (:) are only necessary if the referenced name can be interpreted to see more than one object.

Assignment statements assign values (see page 95) to local variables and procedure parameters in the body of the procedure. Local variables are variables that are declared using the DECLARE statement in the database procedure. The scope of these variables is the database procedure in which they are declared. Variable assignment statements use the '=' or ':=' operator to assign values to local variables. The value assigned can be a constant or the result of the evaluation of an expression. The data types of the value and the local variable must be compatible.

Procedure parameters explicitly declared with the INOUT or OUT modes pass their values back to the calling procedure.

All statements, except a statement preceding an END, ENDFOR, or ENDIF, must be terminated with a semicolon.

If working interactively, the BEGIN and END keywords can be replaced with braces { }, but the terminating semicolon must follow the closing brace if another statement is entered after the CREATE PROCEDURE statement and before committing the transactions.

Parameter Modes

By default, parameters to a database procedure are INPUT only. Though the parameter value may be updated in the body of the procedure, the changed value is not passed back to the calling application, rule, or procedure. The BYREF designation used in a calling application can be used to force the return of the modified parameter value. For more information on BYREF, see Execute Procedure (see page 515).

For database procedures called from other database procedures or by the firing of a rule, the INOUT and OUT modes can be coded in a parameter declaration to return the modified value of the parameter back to the caller of the procedure. This allows results to be passed from one procedure to another and column values to be changed by BEFORE rules defined on a particular table.

Nullability and Default Values for Parameters

Database procedures can be called from embedded SQL applications or from interactive SQL. The caller supplies values for procedure parameters. The WITH DEFAULT, NOT DEFAULT, WITH NULL, and NOT NULL clauses can be used to specify whether parameters have default values and whether they are nullable. These clauses have the following meanings for database procedure parameters:

WITH DEFAULT

The caller does not have to specify a value for the parameter. If the parameter is nullable, its default value is null. If the parameter is not nullable, its default value is 0 (for numeric data types) or blanks (for character data types).

NOT DEFAULT

The caller must specify a value for the parameter. If no value is specified, the DBMS Server issues an error.

WITH NULL

The parameter can be null.

NOT NULL

The parameter cannot be null.

The combined effects of these clauses are as follows:

Parameter	Description
WITH NULL	The parameter can be null. If no value is provided, the DBMS Server passes a null.
NOT NULL WITH DEFAULT	The parameter does not accept nulls. If no value is provided, the DBMS Server passes 0 for numeric and money columns, or an empty string for character and date columns.
NOT NULL NOT DEFAULT or NOT NULL	The parameter is mandatory and does not accept nulls.
WITH NULL WITH DEFAULT	Not allowed.
WITH NULL NOT DEFAULT	Not allowed.
WITH DEFAULT	Not allowed without not null clause.
NOT DEFAULT	Not allowed without not null clause.

Set Of Parameters

A SET OF parameter is required either when a global temporary table is being passed to the procedure or when the procedure is invoked by the triggering of a statement level rule. Also, a SET OF parameter declaration consists of a SET OF parameter name and an accompanying elements list. For more information, see Create Rule (see page 384).

In the case of a procedure invoked by an EXECUTE PROCEDURE statement with a global temporary table parameter, the SET OF elements correspond to the temporary table columns. For more information, see Temporary Table Parameter (see page 519) under Execute Procedure (see page 515).

In the case of a procedure invoked by a statement level rule, the SET OF element list consists of one entry for each actual parameter in the create rule execute procedure clause. The syntax of these entries is identical to that of normal (that is, non-SET OF) formal parameters. The type definitions must be compatible with (though not necessarily identical to) the corresponding actual parameters. The names must be the same, however, as this is how the equivalence between the actual parameters and the SET OF elements is determined.

Once a SET OF parameter is defined in a CREATE PROCEDURE statement, it can be treated exactly like any base table or view from within the procedure. The SET OF elements are the columns of the table and the parameter name is the surrogate name of the table. The parameter name can be used as a table name in any SELECT, DELETE, UPDATE, or INSERT statement within the procedure.

For example, it can be used in an INSERT...SELECT... statement to return the multi-row result of a complex SELECT statement with a single procedure call, or it can be used in the FROM clause of an UPDATE to effect the update of many rows with a single procedure call.

For example:

```
create procedure gttproc (gtt1 set of (coll int, col2 float not null, col3
char(8))) as begin
....
insert into table select * from gtt1;
....
end;
```

gtt1 is defined as a SET OF parameter to procedure gttproc and is used in the FROM clause of a SELECT statement in the body of the procedure.

Embedded Usage

The embedded CREATE PROCEDURE statement is identical to its interactive version, with the following exceptions and additions:

- Braces {} cannot be used in place of the BEGIN and END clauses.

- All statements within the procedure must be separated by semicolons. The statement terminator after the final end clause follows the syntax of the host language.

- The CREATE PROCEDURE statement cannot contain any references to host language variables.

- The rules for the continuation of statements over multiple lines follow the embedded SQL host language rules. String literals, continued over multiple lines, also follow the host language rules. For details about the continuation of database procedure statements and string literals, see the *Embedded SQL Companion Guide*.

- Comments in a procedure body follow the comment rules of the host language.

- The INCLUDE statement cannot be issued inside a database procedure. However, an include file can contain an entire CREATE PROCEDURE statement.

The preprocessor does not validate the SQL syntax of the CREATE PROCEDURE statement. The syntax is validated at runtime.

Permissions

You must have permission to access the tables and views specified in queries issued by the procedure. If the procedure uses database events owned by other users, you must have the required permissions (raise and register) on the database events. If the procedure executes database procedures owned by other users, you must have execute permission for those procedures.

If permissions are changed after the procedure is created and the creator of the procedure no longer has permissions to access the tables and views, a runtime error results when the procedure is executed.

The GRANT statement can be used to assign the [NO]CREATE_PROCEDURE privilege to specific users, groups, and roles.

Related Statements

Create Rule (see page 384)

Declare (see page 446)

Drop Procedure (see page 490)

Execute Procedure (see page 515)

Grant (privilege) (see page 534)

Examples: Create Procedure

The following are CREATE PROCEDURE statement examples:

1. This database procedure, mark_emp, accepts an employee ID number and a label string as input. The employee matching that ID is labeled and an indication is returned.

```
create procedure mark_emp
    (id integer not null, label varchar(100)) as
begin
    update employee
    set comment = :label
    where id = :id;
    if iirowcount = 1 then
    message 'Employee was marked';
    commit;
    return 1;
    else
    message'Employee was not marked - record error';
    rollback;
    return 0;
    endif;
end;
```

2. In this example, the database procedure, add_n_rows, accepts as input a label, a base number, and a number of rows. It inserts the specified number of rows into the table blocks, starting from the base number. If an error occurs, the procedure terminates and the current row number is returned.

```
create procedure add_n_rows
    (base integer not null, n integer,
    label varchar(100)) as
declare
    limit integer;
    err integer;
begin
    limit = base + n;
    err = 0;
    while (base < limit) and (err = 0) do
    insert into blocks values (:label, :base);
    if iierrornumber > 0 then
        err = 1;
    else
        base = base + 1;
    endif;
    endwhile;
    return :base;
end;
```

3. The following example illustrates the use of global temporary tables as procedure parameters. The database procedure, gttproc, accepts as input a surrogate table name; gtt1 is defined as a SET OF parameter to the gttproc procedure and is used in the FROM clause of a SELECT statement in the body of the procedure.

```
create procedure gttproc
    (gtt1 set of (col1 int, col2 float not null, col3 char(8))) as
begin
...
    insert into table1
    select * from gtt1;
...
end;
```

4. The following example illustrates the use of parameter modes to effect the return of the customer name and zipcode column values for a given customer number to the calling procedure.

```
create procedure getnamezip (IN custno int not null, OUT custname, OUT
custzip) as
begin
...
    select c_name, c_zip into :custname, :custzip from customer where c_id =
:custno;
...
end;
```

Create Profile

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE PROFILE statement creates a new user profile.

Syntax

The CREATE PROFILE statement has the following format:

```
[EXEC SQL] CREATE PROFILE profile_name [WITH with_item {,with_item}]

with_item = NOPRIVILEGES | PRIVILEGES = ( priv {, priv} )
                        | NOGROUP | GROUP = default_group
                        | SECURITY_AUDIT= ( audit_opt {,audit_opt})
                        | NOEXPIREDATE | EXPIRE_DATE = 'expire_date'
                        | DEFAULT_PRIVILEGES = (priv {,priv})| ALL
                                    | NODEFAULT_PRIVILEGES
```

profile_name

Defines the name of the profile that is being created. Must be a valid object name that is unique in the installation.

priv

Specifies a subject privilege. Subject privileges apply to the user regardless of the database to which the user is connected. If the privileges clause is omitted, the default is NOPRIVILEGES.

The *priv* must be one of the following:

CREATEDB

Allows users to create databases.

TRACE

Allows users to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows users to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also, allows user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating or altering users, profiles, group and roles, and to grant or revoke database and installation resource controls.

default_group

Specifies the default group for users with this profile. Must be an existing group. For details about groups, see Create Group (see page 352). To specify that the user is not assigned to a group, use the NOGROUP option. If the GROUP clause is omitted, the default is NOGROUP.

audit_opt

Defines security audit options:

ALL_EVENTS

All activity by the user is audited.

DEFAULT_EVENTS

Only default security auditing is performed, as specified with the ENABLE and DISABLE SECURITY_AUDIT statements.

QUERY_TEXT

Auditing of the query text associated with specific user queries is performed. Security auditing of query text must be enabled as a whole, using the ENABLE and DISABLE SECURITY_AUDIT QUERY_TEXT statements.

expire_date

> Specifies an optional expiration date associated with each user using this profile. Any valid date can be used. When the expiration date is reached, the user is no longer able to log on. If NOEXPIRE_DATE is specified, this profile has no expiration limit.

DEFAULT_PRIVILEGES = (*priv* **{,** *priv***}) | ALL | NODEFAULT_PRIVILEGES**

> Defines the privileges initially active when connecting to Ingres. These must be a subset of those privileges granted to the user.

ALL

> All the privileges held by the profile are initially active.

NODEFAULT_PRIVILEGES

> No privileges are initially active.

Description

User profiles are a set of subject privileges and other attributes that can be applied to a user or set of users. Each user can be given a profile, which provides the default attributes for that user.

A profile includes:

- Subject privileges
- Default subject privileges
- Default user groups
- Security auditing attributes
- Expire date

A default profile, changeable by the system administrator, is created during installation that determines the default user attributes when no profile is explicitly specified. The initial default profile is:

- NOPRIVILEGES
- NODEFAULT_PRIVILEGES
- NOEXPIRE_DATE
- NOGROUP
- NOSECURITY_AUDIT

Embedded Usage

The WITH clause in the embedded CREATE PROFILE statement can be specified using a host string variable (with :*hostvar*).

Permissions

You must have MAINTAIN_USERS privileges and be connected to the iidbdb database. Additional privileges are required to perform certain operations, as summarized in this table:

Action	Privilege Required
Set security audit attributes	Maintain_audit

Locking

The CREATE PROFILE statement locks the iiprofile system catalog exclusively.

Related Statements

Alter Profile (see page 294)

Alter User (see page 312)

Create User (see page 440)

Drop Profile (see page 491)

Drop User (see page 498)

Examples: Create Profile

The following are CREATE PROFILE statement examples:

1. Specify a profile for a particular user.

```
create profile dbop;
    create user bspring with profile = dbop;
```

2. Create a dbop profile with the appropriate privileges to maintain a database.

```
create profile dbop with
privileges = ( operator, maintain_locations, trace ),
group = dbopgroup;
```

Create Role

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE ROLE statement defines one or more role identifiers and their associated password. Role identifiers are used to associate privileges with applications. After the role identifiers are created and privileges have been granted to them, use them with the CONNECT statement to associate those privileges with the session. For a discussion of role identifiers, see the *Database Administrator Guide*. For information about the privileges granted to role identifiers, see Grant (role) (see page 550) .

Only users who have been granted access to a role can use a role. The creator of a role is automatically granted access to that role.

Syntax

The CREATE ROLE statement has the following format:

```
[EXEC SQL] CREATE ROLE role_id {, role_id}
[WITH with_option {,with_option}]

with_option = NOPASSWORD | PASSWORD = 'role_password'
                | PASSWORD = X'encrypted_role_password'
                        | EXTERNAL_PASSWORD
                        | NO PRIVILEGES | PRIVILEGES = ( priv {,priv} )
                        | NOSECURITY_AUDIT | SECURITY_AUDIT
```

role_id

Specifies the user name to be created. Must be a valid object name that is unique among all role, group, and user identifier names in the installation.

If an invalid role identifier is specified, the DBMS Server returns an error but processes all valid role identifiers.

Role identifiers are stored in the iirole catalog of the iidbdb. For details about system catalogs, see the *Database Administrator Guide.*

role_password

Allows a user to change his or her own password. In addition, users with the MAINTAIN_USERS privilege can change or remove any password. If *role_password* contains uppercase or special characters, enclose it in single quotes. Any blanks in the password are removed when the password is stored.

Limits: *Role_password* can be no longer than 24 characters

Default: NOPASSWORD if the password clause is omitted.

To remove the password associated with *role_id*, specify NOPASSWORD.

To allow a user's password to be passed to an external authentication server for authentication, specify EXTERNAL_PASSWORD.

priv

Specifies one of the following subject privileges, which applies to the user regardless of the database to which the user is connected:

CREATEDB

Allows the user to create databases.

TRACE

Allows the user to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows the user to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also allows the user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating, altering or dropping users, profiles and group and roles, and to grant or revoke database and installation resource controls.

Default: NOPRIVILEGES if the privileges clause is omitted.

NOSECURITY_AUDIT | SECURITY_AUDIT

Specifies audit options, as follows:

NOSECURITY_AUDIT

Uses the security_audit level for the user using the role.

SECURITY_AUDIT

Audits all activity for anyone who uses the role, regardless of any SECURITY_AUDIT level set for an individual user.

Default: NOSECURITY_AUDIT if the security_audit clause is omitted.

Embedded Usage

The WITH clause in an embedded CREATE ROLE statement can be specified using a host string variable (with :*hostvar*).

Permissions

You must have the MAINTAIN_USERS privilege and be connected to the iidbdb database. Additional privileges are required to perform certain operations, as summarized in the table below:

Action	Privilege Required
Set security audit attributes	maintain_audit

Locking

The CREATE ROLE statement locks pages in the iirole catalog of the iidbdb. This can cause sessions attempting to connect to the server to suspend until the statement is completed.

Related Statements

Alter Role (see page 298)

Drop Role (see page 492)

Grant (role) (see page 550)

Examples: Create Role

The following are CREATE ROLE statement examples:

1. Create a role identifier and password for the inventory application of a bookstore.

   ```
   create role bks_onhand with password = 'hgwells';
   ```

2. Create a role identifier with no password for the daily sales application of the bookstore.

   ```
   create role dly_sales with nopassword;
   ```

3. Create a role identifier and its password for the new employee application of the bookstore.

   ```
   create role new_emp with password = 'good luck';
   ```

4. In an application, create a role identifier and its password for an accounts payable application.

   ```
   exec sql create role acct_pay with
       password = piper;
   ```

5. Create a role with a password and additional privileges.

   ```
   create role sysop
       with password = 'sysoppwd',
       privileges = (operator, createdb, maintain_locations);
   ```

6. Create a role with external password verification.

   ```
   create role sysop
       with external_password;
   ```

Create Rule

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

CREATE RULE statement defines an Ingres rule. A rule executes a specified database procedure whenever a specified condition is true. For a detailed discussion of the use of rules to enforce referential integrity and security, see the *Database Administrator Guide.*

Syntax

The CREATE RULE statement has the following format:

```
[EXEC SQL] CREATE RULE [schema.]rule_name table_condition
            [FOR EACH {ROW | STATEMENT}]
            EXECUTE PROCEDURE [schema.]proc_name[(parameter = value
                        {, parameter = value})]
```

The *table_condition* has the following format:

```
BEFORE|AFTER statement_type {, statement_type} ON|OFF|FROM|INTO
[schema.]table_name
```

        `[REFERENCING [OLD AS old_corr_name] [NEW AS new_corr_name]]`

        `[WHERE qualification]`

rule_name

Specifies the name of the rule. Rules cannot be defined against views, only against base tables.

Limits: The rule name must be a valid object name of no more than 32 characters that is unique within the set of rules owned by the user issuing the CREATE RULE statement.

table_condition

Defines the action that fires the rule. For details, see Table_Condition (see page 388).

FOR EACH ROW|STATEMENT

Defines a row- or statement-level rule. For details, see Row and Statement Level Rules (see page 386).

proc_name

Specifies the procedure to be executed when a statement fires the rule. The procedure must exist at the time the rule is created.

parameter

Specifies one or more values to be passed to the procedure. Not all of the parameters appearing in the definition of the invoked procedure have to be included. However, those that are included must match in name and data type. Parameters can be specified using host language variables. Parameters cannot be passed by reference to a procedure that is executed by a rule.

value

Contains a constant or an old or new value in the row that caused the rule to fire. Constant values cannot contain function expressions. If *value* represents a row value, it must be qualified by a correlation name. For details, see the description of the referencing clause.

Row and Statement Level Rules

The FOR EACH clause optionally allows defining a row or statement rule; FOR EACH ROW is the default and the only option for BEFORE rules.

When the row level rule is executed, a parameter list is built and the procedure is invoked for each row touched by the statement. If a single DELETE or UPDATE statement affects 100 rows, the procedure invocation occurs 100 times.

When the statement level rule is executed, the parameters passed in the procedure invocation for each qualifying row of the triggering statement are accumulated in an internal temporary table. The temporary table, containing information from all rows touched by the triggering statement, is passed with a single call to the rule procedure. This can potentially save many calls to the rule procedure.

All qualifying rows contained in an internal temporary table are processed by the triggering statement so that the rule procedure is invoked just once.

Examples of both row and statement level rules follow.

In this example, a row level rule (the default) executes the *ruleproc1* procedure for every INSERT INTO table_x WHERE col1 > 5000:

```
create rule r1 after insert into table_x where   new.col1> 5000
        execute procedure ruleproc1 (p1 = new.col1, p2 = new.col5);
```

The following example is an exact equivalent of the preceding one; either version can be used:

```
create rule r1 after insert into table_x where   new.col1> 5000
        for each row execute procedure ruleproc1 (p1 = new.col1, p2 = new.col5);
```

In this example, a statement level rule executes the procedure *ruleproc2* after each delete of table_y. The col1 and col2 values for each row deleted by a single statement are accumulated in an internal temporary table and are passed together to *ruleproc2* with a single call:

```
create rule r2 after delete from table_y
        for each statement execute procedure ruleproc2 (q1 = old.col1, q2 =
old.col2);
```

Database Procedures

Proc_name is the name of the database procedure that is executed whenever the rule fires. The specified procedure must exist when the rule is created. Use Create Procedure (see page 366) to define a database procedure.

To execute a database procedure owned by another user, specify *schema.procedurename*, where *schema* is the user identifier of the owner of the procedure; you must have execute privilege for the procedure.

The *parameter* list allows values to be passed to the invoked procedure. The number and type of the parameters must be consistent with the number and type in the definition of the invoked procedure.

The *values* can include constants, expressions, or references to (old and new) values in the row that caused the rule to fire. (Old and new see values in the row before and after the specified change.) When the *value* is a constant, the keywords USER and NULL are acceptable values. A constant *value* cannot be a function expression, such as date('now').

Whenever value refers to a value in a row, it must be referenced by a correlation name. The referencing clause allows you to choose these correlation names. For example, the following statement establishes the correlation name, first, for referencing old values and, second, for referencing new values:

```
create rule r1 after update on table1
referencing old as first new as second
execute procedure p1
        (a = first.c1, b = second.c1);
```

Old and new correlation names can be specified in any order in the referencing clause. If correlation names are not chosen, the default is as follows:

```
referencing old as old new as new
```

If the name of the table is used as the correlation name, the DBMS Server assumes that the values referenced are new values.

If the *statement_type* in the table condition is INSERT, only new column values are available for the procedure. If the *statement_type* is DELETE, only old column values are available.

If both old and new correlation names are specified in a rule that includes an INSERT or a DELETE, or in the *statement_type* list, the DBMS Server assumes that both the old and new correlation names see the same set of values when the rule fires as a result of an INSERT or DELETE.

For example, assume the following rule:

```
create rule few_parts after update, delete
from parts
        execute procedure check_change
        (name = old.name, pre = old.quantity,
        post = new.quantity)
```

If an update fires the rule, the values passed to the procedure are the old and new values. However, if a DELETE fires the rule, the DBMS Server assumes that the correlation names are both old because the new value does not exist in a delete operation.

Table_Condition

The table_condition on a CREATE RULE statement defines the action that fires the rule.

The *table_condition* has the following format:

```
BEFORE|AFTER statement_type {, statement_type} ON|OFF|FROM|INTO
[schema.]table_name

                    [REFERENCING [OLD AS old_corr_name] [NEW AS new_corr_name]]
                    [WHERE qualification]
```

BEFORE|AFTER

Specifies that the rule be fired before or after the effective execution of the triggering statement.

statement_type

Specifies the type of statement that fires (triggers) the rule. Valid values include:

INSERT

UPDATE[(*column_name* {, *column_name*})]

DELETE

Note: Only one of each statement type can be included in a single *table_condition*.

table_name

Specifies the table against which the rule is created.

old_corr_name

Specifies the correlation name for old (prior to change) values in a row. The name specified is used to qualify references to old values in the parameter list and the *qualification*.

Default: Old

new_corr_name

> Specifies the correlation name for new (after the change) values in a row. The name specified is used to qualify new values in the parameter list and *qualification*.
>
> Default: New

qualification

> Indicates the specific change that must occur to the table to fire the rule. The qualification must evaluate to true or false.
>
> Any column references in *qualification* must be qualified by the correlation names defined in the referencing clause or by the default correlation names, old and new.
>
> The *qualification* cannot include a subselect or an aggregate (set) function such as count or sum.

The action that fires the rule can be:

- An INSERT, UPDATE, or DELETE performed on any column in the table.

  For example, the following rule fires whenever an INSERT is executed against the employee table:

  ```
  create rule insert_emp after insert into employee
      execute procedure new_emp (name = new.name,
      addr = new.address);
  ```

  In the above example, the AFTER keyword dictates that the rule fires after the INSERT statement is executed. This can be used, for example, to store auditing information in another table.

  If the rule uses the BEFORE keyword, it fires before the INSERT is executed. This can be used to alter the value of the name or addr columns (for example, after validating them against another table).

  Note: If a column name is not specified after update, the rule fires after an update to any column in the table.

- An update performed on specified columns in a table.

  For example, the following rule fires whenever the salary column in the employee table is changed.

  ```
  create rule emp_salary after update(salary)
      of employee
      execute procedure check_sal
      (name = old.name, oldsal = old.salary,
      newsal = new.salary);
  ```

  Up to 1024 columns can be specified in the UPDATE clause. The rule is fired whenever one or more of the columns is updated.

- A change to the table that satisfies the specified WHERE clause *qualification.*

For example, the following rule fires whenever an update to the quantity column of the parts table results in a quantity value of less than 50:

```
create rule few_parts after update(quantity)
    of parts
    where new.quantity < 50
    execute procedure issue_order
    (name = old.name,
    quantity = new.quantity);
```

Embedded Usage

In an embedded CREATE RULE statement, host language variables can be used to represent the parameters of the procedure.

Permissions

To create a rule against a table, you must own the table and have EXECUTE privileges for the procedure invoked by the rule.

Once a rule is applied to a table, any user who has permission to access that table using the operation specified by the rule has permission to fire the rule and consequently execute its associated procedure.

Locking

The CREATE RULE statement takes an exclusive lock on the specified table.

Related Statements

Delete (see page 470)

Execute Procedure (see page 515)

Insert (see page 572)

Update (see page 702)

Examples: Create Rule

The following are CREATE RULE statement examples:

1. The following two rules are applied to the employee table. The first rule fires whenever a change is made to an employee record, to log the action. The second rule fires only when a salary is increased. An UPDATE statement that increases the salary fires both the rules-in this case, the order of firing is not defined.

```
create rule emp_updates after delete, insert,
    update of employee
    execute procedure track_emp_updates
    (name = new.name);


create rule emp_salary after update(salary, bonus)
    of employee
    where new.salary > old.salary
    execute procedure check_sal
        (name = old.name,
        oldsal = old.salary,
        newsal = new.salary,
    oldbonus = old.bonus,
    newbonus = new.bonus);
```

2. The following two rules track changes to personnel numbers. When an employee is removed, an entry is made into the manager table, which in turn causes an entry to be made into the director table. Even if an entry is made directly into the manager table, the director table is still notified.

```
create procedure manager_emp_track
    (ename varchar(30), mname varchar(30)) as
begin
    update manager set employees = employees - 1
    where name = :mname;
    insert into mgrlog values
    ('Manager: ' + :mname +
        '. Deleted employee: ' + :ename);
end;


create rule emp_delete after delete from employee
    execute procedure manager_emp_track
    (ename = old.name, mname = old.manager);


create procedure director_emp_track
    (dname varchar(30)) as
begin
    update director set employees = employees - 1
    where name = :dname;
end;
```

```
create rule manager_emp_delete
    after update(employees) of manager
    where old.employees - 1 = new.employees
    execute procedure director_emp_track
    (dname = old.director);
```

3. The following example shows a rule that is fired before the insertion of rows into the customer table. The triggered procedure can verify the zip code (or other columns) of the inserted row, possibly changing it if the originally-assigned value is in error.

```
create rule cust_zip_replace
    before insert into customer
    execute procedure verify_zip
    (custno = c_no, zipcode = c_zip);
```

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). Each user has a maximum of one schema per database. If an error occurs within the CREATE SCHEMA statement, the entire statement is rolled back.

The statements within the CREATE SCHEMA statement must not be separated by semicolon delimiters; however, the CREATE SCHEMA statement must be terminated by placing a semicolon after the last object definition statement (CREATE TABLE, CREATE VIEW, or GRANT).

If object definitions are omitted, an empty schema is created.

To issue grant statements in a CREATE SCHEMA statement, you must have the required privileges. Specifically, to grant a privilege on an object you do not own, you must have been granted the privilege WITH GRANT OPTION (see page 547).

If an invalid GRANT statement is issued within a CREATE SCHEMA statement, the outcome is determined as follows:

- If you have no privileges whatsoever on the object against which you issue the GRANT statement, the entire CREATE SCHEMA statement is aborted.

- If you have any privilege whatsoever on the object, a warning is issued and the invalid portions of the grant do not succeed. The valid portions of the grant do succeed, and the CREATE SCHEMA statement is not aborted.

For example, if user andre has been granted SELECT WITH GRANT OPTION on table tony.mytable and issues the following GRANT statement within a CREATE SCHEMA statement:

```
grant select, insert on tony.mytable to fred
```

user fred is granted SELECT privilege but not INSERT privilege, and a warning is issued.

If a CREATE SCHEMA is issued specifying an existing schema (*schema_name*), the DBMS Server issues an error. To add objects to an existing schema, issue the required CREATE statements outside of a CREATE SCHEMA statement.

If no schema exists for the effective user identifier, one is implicitly created when any database object is created. If a CREATE SCHEMA statement is subsequently issued for the user, the DBMS Server returns an error.

If, within a CREATE SCHEMA statement, tables are created that have referential constraints, the order of CREATE TABLE statements is not significant. This differs from the requirements for creating tables with referential constraints outside of a CREATE SCHEMA statement, where the referenced table must exist before creating a constraint that references it. For details about referential constraints, see Create Table (see page 402).

Other users can reference objects in your schema if you have granted them the required permissions. To access an object in a schema other than the schema for the effective user of the session, specify the object name as follows:

```
schema.object
```

For example, user harry can select data from the employees table of the accounting group (if accounting has granted harry select permission). Harry can issue the following SELECT statement:

```
select lname, fname from accounting.employees
```

Embedded Usage

You cannot use host language variables in an embedded CREATE SCHEMA statement.

Permissions

This statement is available to all users.

Locking

The CREATE SCHEMA statement takes an exclusive lock on a page in the iischema catalog. Locks are acquired by the individual CREATE statements within the CREATE SCHEMA statement, but released only when the CREATE SCHEMA statement itself is committed. If the CREATE SCHEMA statement contains CREATE statements that acquire locks in excess of the maximum configured for the DBMS Server, the CREATE SCHEMA statement is aborted.

Related Statements

Create Table (see page 402)

Create View (see page 448)

Grant (privilege) (see page 534)

Example: Create Schema

Create a schema for the accounting user:

```
create schema authorization accounting
        create table employees (lname char(30) not null,
                fname char(30) not null,
                salary money,
                dname char(10)
                        references dept(deptname),
                primary key (lname, fname)

        create table dept(deptname char(10) not null unique,
        location char(15),
        budget money,
        expenses money 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 Security_Alarm

Valid in: SQL, ESQL

The CREATE SECURITY_ALARM statement specifies, for tables, databases, or the current installation, the events to be written to the security log.

Syntax

The CREATE SECURITY_ALARM statement has the following format:

```
[EXEC SQL] CREATE SECURITY_ALARM [alarm_name] ON
                [TABLE | DATABASE] [schema.]object_name {, [schema.]object_name} |
                   CURRENT INSTALLATION
                [IF SUCCESS | FAILURE | SUCCESS, FAILURE]
                [WHEN SELECT | DELETE | INSERT | UPDATE | CONNECT | DISCONNECT]
                [BY [USER | GROUP | ROLE] auth_id{, auth_id} | PUBLIC;]
                [RAISE DBEVENT [dbevent_owner.]dbevent_name [dbevent_text]]
```

object_name

Specifies the table or database for which security events are logged.

IF SUCCESS | FAILURE | SUCCESS, FAILURE

Specifies when logging occurs:

SUCCESS

Creates a log record when a user succeeds in performing the specified type of access.

FAILURE

Creates a log record when a user attempts to perform the specified type of access and fails (the query is aborted). Users can fail to gain access to a table because they lack the required permissions.

SUCCESS, FAILURE

Logs all attempts to access the tables.

WHEN clause

Specifies the types of access to be logged. Any combination of the access types shown in the syntax diagram can be specified, in a comma separated list.

BY clause

Specifies the user names of the users for whom logging is performed.

To log access attempts for all users, specify PUBLIC.

Default: PUBLIC

Embedded Usage

You cannot use host language variables in an embedded CREATE SECURITY_ALARM statement.

Permissions

You must be the owner of the table.

Locking

The CREATE SECURITY_ALARM statement locks the specified table, the iisecurity_alarms catalog, the iipermits catalog, and the iiprotect catalog.

Related Statements

Disable Security_Audit (see page 477)

Enable Security_Audit (see page 499)

Drop Security_Alarm (see page 494)

Help Security_Alarm (see page 551)

Examples: Create Security_Alarm

The following are CREATE SECURITY_ALARM statement examples:

1. Log all successful changes to the employee table.

    ```
    create security_alarm on table employee
        if success when insert, update, delete by public;
    ```

2. Specify alarms for a specific user group or application role.

    ```
    create security_alarm clerk_update on table secure_data if failure when
    update by group clerk
    ```

    These alarms are fired when a session connects as the specified group or role.

3. Specify alarm on a particular database or the current installation to raise an alarm when user, spy, connects to any database.

    ```
    create security_alarm secconnect on current installation when connect by user
    spy
    ```

4. Raise an optional database event, secconnect, as the result of an alarm firing when user, spy, connects to database sec1.

    ```
    create security_alarm secconnect on database sec1
    when connect by user spy
    raise dbevent secconnect 'user spy connected to sec1 database';
    ```

Create Sequence

Valid in: SQL, ESQL

The CREATE SEQUENCE statement creates new sequences. Sequences are defined database entities that are used to supply a set of integer values to an application in sequential order according to a set of definition parameters. These parameters are set by specifying sequence_options as explained below.

Syntax

The CREATE SEQUENCE statement has the following format:

```
[EXEC SQL] CREATE SEQUENCE [schema.]sequence_name [sequence_options]
```

sequence options

Define parameters that control how the sequence supplies data when requested by an application. Sequence options can be specified in any order, and none are required.

Any of the following options can be specified in a blank-space separated list:

AS *data type*

Specifies whether the data type is an integer or a decimal (with some precision and 0 scale).

Default: INTEGER

START WITH *number*

Specifies the start of the sequence as some integer constant. The default value is 1 for positive sequences (positive increment) and -1 for negative sequences (negative increment). (This option is valid with the CREATE SEQUENCE statement only.)

RESTART WITH *number*

Specifies a new start value for the sequence. (This option is valid with the ALTER SEQUENCE statement only.)

INCREMENT BY *number*

Specifies the increment value (positive or negative) that produces successive values of the sequence.

Default: 1

MAXVALUE *number*

Specifies the maximum value allowed for the sequence.

Defaults: For positive integer sequences: $2^{31}-1$

For positive decimal(n) sequences: $10^{(n+1)}-1$

For negative sequences: -1

NO MAXVALUE / NOMAXVALUE

Specifies that sequences can generate values with an upper bound equivalent to that of the data type chosen to hold the sequence (for example, $2^{31}-1$ for integers).

MINVALUE *number*

Specifies the minimum value allowed for the sequence.

Default: For positive sequences: 1

For negative integer sequences: -2^{31}

For negative decimal(n) sequences: $-(10^{(n+1)}-1)$

NO MINVALUE/NOMINVALUE

Specifies that sequences can generate values with a lower bound equivalent to that of the data type chosen to hold the sequence (for example, -2^{31} for integers).

CACHE *number*

Specifies the number of sequence values held in server memory. Once the supply of numbers is exhausted, Ingres requires a catalog access to acquire the next set.

Default: 20

NO CACHE / NOCACHE

Specifies that sequence values are not to be cached by the server. When this option is selected, a catalog access is required for each request for a sequence value. This can severely degrade application performance.

Default: CACHE 20 (when neither CACHE nor NOCACHE are specified), which ensures that the catalog overhead will be reasonably small.

CYCLE

Specifies that the sequence restarts at the beginning value once it reaches the minimum value (negative increment) or maximum value (positive increment).

Default: NO CYCLE

NO CYCLE / NOCYCLE

Specifies that the sequence is not cycled when the last valid value is generated. An error is issued to the requesting transaction.

ORDER

NO ORDER / NOORDER

These options are included solely for syntax compatibility with other DBMSes that implement sequences, and are not currently supported in Ingres.

Default: NOORDER

Permissions

You must have CREATE_SEQUENCE privileges. You must also have the NEXT privilege to retrieve values from a defined sequence. For information on the NEXT privilege, se (see page 534)e Grant (privilege).

Locking and Sequences

In applications, sequences use logical locks that allow multiple transactions to retrieve and update the sequence value while preventing changes to the underlying sequence definition. The logical lock is held until the end of the transaction.

Related Statements

Alter Sequence (see page 303)

Drop Sequence (see page 495)

Examples: Create Sequence

The following are CREATE SEQUENCE statement examples:

1. Define the start value for sequence "XYZ" as 10.

   ```
   create sequence XYZ start with 10
   ```

2. Define the increment value for sequence "XYZ" as 10 and the number of cached values as 500.

   ```
   create sequence XYZ increment by 10 cache 500
   ```

Create Synonym

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The CREATE SYNONYM statement defines a synonym for a table, view, or index. A *synonym* is an alias (alternate name) for an object.

References to synonyms in applications are resolved to their base objects at runtime. References to synonyms in definitions of database procedures, views, and permissions are resolved at the time the procedure, view, or permission is defined. For this reason, the synonym must be valid at definition time and at runtime, but can be dropped and recreated in between.

Syntax

The CREATE SYNONYM statement has the following format:

```
[EXEC SQL] CREATE SYNONYM synonym_name
              FOR [schema.]object
```

synonym_name

Specifies a valid object name and must not conflict with the names of other tables, views, indexes, or synonyms owned by the user issuing the statement. Synonyms can be used any place that table, view, or index identifiers are required.

Embedded Usage

You cannot use host language variables in an embedded CREATE SYNONYM statement.

Permissions

This statement is available to all users.

Locking

The CREATE SYNONYM statement locks the iisynonym system catalog, and takes an exclusive lock on the table, view, or index for which a synonym is being created.

Related Statements

Drop Synonym (see page 497)

Examples: Create Synonym

The following are CREATE SYNONYM statement examples:

1. Create a synonym for the authors table.

   ```
   create synonym writers for authors;
   ```

2. Create a synonym for the composers table, owned by another user.

   ```
   create synonym cmp for tony.composers;
   ```

Create Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE TABLE statement creates a base table.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

Syntax

The CREATE TABLE statement has the following format:

```
[EXEC SQL] CREATE TABLE [schema.] table_name
             (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

Defines the name of the new table, which must be a valid object name.

column_specification

Defines the characteristics of the column, as described in Column Specifications (see page 407).

table_constraint

Specifies the table-level constraint as described in Column-Level Constraints and Table-Level Constraints (see page 420).

with_clause

Consists of a comma-separated list of one or more of the following options, described in detail in With_Clause for Create Table (see page 430):

- LOCATION = (*location_name* {, *location_name*})

- [NO]JOURNALING

- [NO]DUPLICATES

- PAGE_SIZE = *n*

- LABEL_GRANULARITY = TABLE | ROW | SYSTEM_DEFAULT

- SECURITY_AUDIT = (*audit_opt* {, *audit_opt*})

- SECURITY_AUDIT_KEY = (*column*)

- NOPARTITION | PARTITION = (*partitioning-scheme*)

Additional options on the With_Clause for Create Table...as Select (see page 434) are as follows:

- STRUCTURE = HASH | HEAP | ISAM | BTREE

- KEY = (*column_name* {, *column_name*})

- FILLFACTOR = *n*

- MINPAGES = *n*

- MAXPAGES = *n*

- LEAFFILL = *n*

- NONLEAFFILL = *n*

- COMPRESSION[= ([[NO]KEY] [,[NO]DATA])] | NOCOMPRESSION

- ALLOCATION = *n*

- EXTEND = *n*

- PRIORITY = *n*

subselect

Specifies a SELECT clause, described in detail in Select (interactive) (see page 641). Also see Using Create Table...As Select (see page 421).

Note: Subselect cannot be used when creating a table in one or more raw locations—that is, CREATE TABLE *raw_table* AS SELECT... WITH LOCATION = (*raw_loc*).

Description

The CREATE TABLE statement creates a base table. A base table contains rows independently of other tables (unlike a view, which has no independent existence of its own). Rows in a base table are added, changed, and deleted by the user (unlike an index, which is automatically maintained by the DBMS Server).

The default page size is the smallest of either 8K (8192 bytes)—unless changed by the system administrator—or the page size configured in the installation that holds the record. For example, if 2K (2048 bytes), 4K (4096 bytes), and 8K (8192 bytes) page sizes are configured and the row size for a table to be created is 2500 bytes, the table is created with an 8K page size. Or, if 2K, 8K and 16K bytes page sizes are configured and the row size for the table to be created is 12,000 bytes, the table is created with a 16K page size.

Note: If the row is larger than any page size configured, or if a page size too small is specified with the page_size clause, Ingres creates the table, but the larger rows will span multiple pages.

The default storage structure for tables is either B-tree or heap, depending on the setting of the table_auto_structure configuration parameter in combination with the presence of constraint definitions in the CREATE TABLE statement. To create a table that has a different storage structure, specify the STRUCTURE option in the WITH clause.

To create a table that is populated with data from another table, specify CREATE TABLE...AS SELECT. The resulting table contains the results of the SELECT statement.

By default, tables are created without an expiration date. To specify an expiration date for a table, use the SAVE statement. To delete expired tables, use the verifydb utility. For details, see the *System Administrator Guide.*

A maximum of 1024 columns can be specified for a base table.

The following table shows the maximum row length when rows do not span pages.

Page Size	Max Row Length
2048 (2 KB)	2008 bytes
4096 (4 KB)	3988 bytes
8192 (8 KB)	8084 bytes
16384 (16 KB)	16276 bytes
32768 (32 KB)	32660 bytes
65536 (64 KB)	65428 bytes

You can create a table with row size greater than the maximum documented above, up to 256 KB. If the WITH PAGE_SIZE clause is not specified, the table is created with the default page size.

Note: Ingres is more efficient when row size is less than or equal to the maximum row size for the corresponding page size.

Long varchar and long byte columns can contain a maximum of 2 GB characters and bytes, respectively. The length of long varchar or long byte columns cannot be specified.

The following data types require space in addition to their declared size:

- A varchar or text column consumes two bytes (in addition to its declared length) to store the length of the string.

- Nullable columns require one additional byte to store the null indicator.

- In tables created with compression, c columns require one byte in addition to the declared length, and char columns require two additional bytes.

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:
```

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] DEFAULT default_spec | WITH DEFAULT | NOT DEFAULT]
[WITH NULL | NOT NULL]
[[CONSTRAINT constraint_name] column_constraint
{ [CONSTRAINT constraint_name] column_constraint}]
[COLLATE collation_name]
```

column_name

Assigns a valid name (see page 38) 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).

Note: For char and varchar columns, the column specification is in number of bytes (not number of characters).

DEFAULT clause

Specifies whether the column is mandatory, as described in Default Clause (see page 409).

NULL clause

Specifies whether the column accept nulls, as described in Null Clause (see page 411).

[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 414).

COLLATE *collation_name*

Specifies a column-level collation sequence, as one of the following.

Note: A default collation sequence can be specified for the database during database creation. The column-level collation overrides the default collation set for the database. For more information, see createdb command in the *Command Reference Guide*.

UNICODE

Specifies collation for columns containing Unicode data (nchar and nvarchar data types). This is the default collation for Unicode columns (nchar, nvarchar and long nvarchar) and for char, varchar, and long varchar columns if the instance is installed with the UTF8 character set.

UNICODE_CASE_INSENSITIVE

Specifies case insensitive collation for columns containing Unicode data.

UNICODE_FRENCH

Specifies French collation for columns containing Unicode data.

SQL_CHARACTER

Specifies the collation for columns containing char, C, varchar, and text data. This is the default collation for columns with non-Unicode data.

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 is mandatory (requires an entry).

WITH DEFAULT

Indicates that if no value is provided, the DBMS Server inserts 0 for numeric and money columns, or an empty string for character and date columns.

[WITH] DEFAULT *default_spec*

Indicates that if no value is provided (because none is required), the DBMS Server inserts the default value. The default value must be compatible with the data type of the column.

For character columns, valid default values include the following constants:

– USER

– CURRENT_USER

– SYSTEM_USER

The following is an example of using the DEFAULT clause:

```
CREATE TABLE DEPT(dname     CHAR(10),
    location       CHAR(10)   DEFAULT 'NY',
    creation       DATE       DEFAULT '01/01/03',
    budget         MONEY      DEFAULT 10000);
```

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.

■ For fixed-length string columns, if the column is wider than the default value, the default value is padded with blanks to the declared 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 money columns, the default value can be exact numeric (integer or decimal), approximate numeric (floating point), or a string specifying a valid money value. The decimal point and currency sign characters specified in the default value must match those in effect when the value is inserted.

- For date columns, the default value must be a string representing a valid date. If the time zone is omitted, the time zone defaults to the time zone of the user inserting the row.

- For user-defined data types (UDTs), the default value must be specified using a literal that makes sense to the UDT. A default value cannot be specified for a logical key column.

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. WITH NULL is the default for all data types except a system_maintained logical key.

NOT NULL

Indicates that the column does not accept nulls.

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, the DBMS Server inserts a null.

WITH NULL WITH DEFAULT

The column accepts null values. If no value is provided, the DBMS Server inserts a 0 or blank string, depending on the data type.

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 DBMS Server inserts 0 for numeric and money columns, or an empty string for character and date columns.

NOT NULL NOT DEFAULT (or NOT NULL)

The column is mandatory and does not accept nulls, which is typical for primary key columns.

System_Maintained Logical Keys

System_maintained logical key columns are assigned values by the DBMS Server, and cannot be assigned values by applications or end users. The following restrictions apply to logical keys specified as WITH SYSTEM_MAINTAINED:

- The only valid default clause is WITH DEFAULT. If the default clause is omitted, WITH DEFAULT is assumed.

- The only valid null clause for system_maintained logical keys is NOT NULL. If a column constraint or null clause is not specified, NOT NULL is assumed.

- No table constraint can include a system_maintained logical key column. For details about table constraints, see Column-Level Constraints and Table-Level Constraints (see page 420).

System_Maintained Clause	Null Clause	Valid?
WITH SYSTEM_MAINTAINED	NOT NULL	Yes
	WITH NULL	No
	NOT NULLWITH DEFAULT	Yes
	NOT NULL NOT DEFAULT	No
	(none specified)	Yes
NOT SYSTEM_MAINTAINED	NOT NULL	Yes
	WITH NULL	Yes
	NOT NULL WITH DEFAULT	Yes
	NOT NULL NOT DEFAULT	Yes

Sequence Defaults

The default value for a column can be the next value in a sequence.

The Sequence-operator can be in either form NEXT VALUE FOR sequence or sequence.NEXTVAL, where sequence is a sequence name, optionally specified as owner.sequence.

For example:

```
create sequence lineitemsequence;
create table lineitem
        (itemid integer not null,
         itemseq integer not null with default next value for
lineitemsequence);
insert into lineitem (itemid) values (4);
insert into lineitem values (8, next value for lineitemsequence);
insert into lineitem values (15, lineitemsequence.nextval);
insert into lineitem values (16, 23);
insert into lineitem (itemid) values (42);
```

Note: If the the schema of the sequence providing the default value is not specified, then it defaults to the schema (owner) of the table.

Sequence defaults are allowed on numeric columns only (integer, float, decimal). The column data type need not match the sequence data type exactly; sequence values are coerced to column values, if necessary.

If a row is inserted with some auto-incrementing columns defaulted, all relevant sequences are incremented once before the row is inserted, and the new sequence values are used whenever referenced in the row. This means that if two columns reference the same sequence for their default, they receive the same value, not two successive values.

Sequence defaulting is allowed in all contexts where column defaulting is allowed: INSERT, UPDATE (using SET COLUMN=DEFAULT), and COPY (including bulk copy).

Columns added or altered with the ALTER TABLE statement cannot use sequence defaults.

Defining a sequence default does not prevent you from explicitly assigning a value to the column upon insert or update. An explicitly assigned value may collide with a defaulted value from the sequence.

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 (see page 420).

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 char(10) unique not null, ...);
```

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 project (
    proj_id int not null not default,
    proj_dept_id int not null with default,
    proj_name char(25) not null,
    unique (proj_id) with structure = hash);
```

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. For example:

```
create table project (
    proj_id int not null not default,
    proj_dept_id int not null with default,
    proj_name char(25) not null unique,
    unique (proj_dept_id) with structure = hash,
    primary key (proj_id));
```

Note: Unique constraints create system indexes that cannot be explicitly dropped by the table owner. The 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 char(25), sal money,
constraint check_salary check (sal > 0));
```

The expression (see page 147) specified in the check constraint must be a Boolean expression.

To specify a check constraint for a group of columns, specify the check constraint at the table level (as opposed to 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 char(10),
    location char(10),
    budget money,
    expenses money,
constraint check_amount check (budget > 0 and
expenses <= budget));
```

Note: The way nullability is specified for a column determines whether you can change the nullability of the column. If CHECK...NOT NULL is specified for a column, use the ALTER TABLE...DROP CONSTRAINT statement to remove the constraint (because the column is created as nullable-that is, with an additional byte for the null indicator-and the check constraint is used to prevent nulls from being inserted). However, if NOT NULL is specified (as opposed to a CHECK...is NOT NULL CONSTRAINT), the constraint cannot be removed using the ALTER TABLE...DROP CONSTRAINT statement because the column was created without the additional byte for the null indicator, and the additional byte cannot be added.

Check constraints cannot include the following:

- Subselects
- Set functions (aggregate functions)
- Dynamic parameters
- Host language variables

Column-level check constraints cannot reference 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 nul),
      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 for individual columns can be specified as part of the column specification (column-level constraints) or for groups of columns as part of the table definition (table-level constraints).

The constraint has the following syntax:

```
[CONSTRAINT constraint_name] constraint
```

constraint

Is either a column-level constraint (*column_constraint*) or table-level constraint (*table_constraint*).

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]
```

table_constraint is 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]
```

constraint_name

Defines a name for the constraint. If the constraint name is omitted, the DBMS Server assigns one. The constraint name is used when dropping the constraint (using the ALTER TABLE statement).

Note: We recommend defining a name when creating a constraint; otherwise system catalogs must be queried to determine the system-defined name.

Examples

Here is an example of column-level constraints:

```
create table mytable(name char(10) not null,
        id integer references idtable(id),
        age integer check (age > 0));
```

Note: Multiple column constraints are separated by a space.

Here is an example of table-level constraints:

```
create table yourtable(firstname char(20) not null,
            lastname char(20) not null,
            unique(firstname, lastname));
```

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.

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.

Constraint With_Clause—Define 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).

- PAGE_SIZE = *n*

- 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:

```
alter table table_name add constraint constraint_name
primary key(column(s))
with (index = base table structure)
```

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.

Constraints and Integrities

The two types of integrities for tables and their error-handling characteristics are as follows:

Integrities created using the CREATE TABLE and ALTER TABLE statement options

These integrities are specified at the time the table is created or altered. An attempt to update the table with a row containing a value that violates the constraint causes the DBMS Server to abort the entire statement and issue an error.

Integrities created using the CREATE INTEGRITY statement

These integrities are specified after the table is created. An attempt to update the table with a row containing a value that violates the constraint causes the invalid row to be rejected. No error is issued.

The two types of integrities handle nulls differently: check constraints (created using the CREATE TABLE or ALTER TABLE statement) allow nulls by default, whereas integrities created using the CREATE INTEGRITY statement do not allow nulls by default.

In addition to table constraints, use rules to perform integrity checks when a table is updated.

Note: The CREATE TABLE and ALTER TABLE statements are the ANSI/ISO SQL-92-compliant methods for maintaining database integrity.

Partitioning Schemes

A table can be partitioned. Partitioning distributes the rows of a table among a number of sub-tables (partitions). A partitioning scheme determines which rows are sent to which partitions.

After the partitioning scheme is defined, partitioning is managed automatically by Ingres.

To define a table with partitions, use the PARTITION= option in the CREATE TABLE WITH clause.

When creating tables, NOPARTITION is the default.

Each dimension of a partitioning scheme defines a rule, or distribution scheme, for assigning rows to partitions. Conceptually, a dimension defines a set of logical partitions; each logical partition can then be subdivided according to the next dimension's rule. Dimensions are evaluated from left to right.

Four distribution scheme types are available: AUTOMATIC, HASH, LIST, and RANGE. Hash, list, and range are data-dependent and require the ON clause. Automatic distribution is not data dependent and does not allow the ON clause.

An automatic distribution is used when the only goal is to spread rows evenly among the partitions. Rows are arbitrarily assigned to random partitions.

A hash distribution is used to spread rows evenly among the partitions deterministically, by use of a hash value (instead of randomly). Given a value for the partitioning columns, a query can predict which partition contains the rows that have the matching value. Thus a query can restrict its search to a subset of partitions.

A list distribution is used to assign rows to partitions based on specific values in one or more columns. A row's partitioning column values are compared to each partition's list values, and when a match is found, the row is sent to that partition. Multiple list values per partition are allowed. If a row matches any of the list values, that partition is selected. One of the partitions must contain the default value in its list; this partition is selected if the row matches none of the list values.

A range distribution is used to assign ranges of values to partitions. The range containing a row's partitioning column values determines the partition in which the row is placed. The ranges must be defined in such a way that every possible value falls into exactly one range. Overlapping ranges are not allowed. Separate ranges cannot map to the same partition—that is, one range, one partition.

A partition defined with values < *range*va*lue* contains all possible values less than *rangevalue*, down to some smaller *rangevalue* in the scheme. Similarly, a partition defined with values > *rangevalue* contains all possible values greater than *rangevalue*, up to some larger *rangevalue* in the scheme. Because all values must be covered by some range, the smallest *rangevalue* must have the operator **<** (or **<=**), and the largest *rangevalue* must have the operator **>** (or **>=**). The partitions need not be defined in order of *rangevalue*.

Multi-column values are tested from left to right. For example, a three-column value (1, 10, 5) is greater than (1, 2, 3000).

While a null can be incorporated into a *rangevalue*, it is not recommended. The ordering of null relative to non-null values is defined by the SQL standard, so the resulting partitioning is dependent on server implementation.

The optional logical partition names must be unique for each table. The same partition name is allowed to occur in other partitioned tables. If a partition name is omitted, the system generates a name (of the form iipart*nn*).

If NO LOCATION= is specified for a partition, the location list is defined by the enclosing statement's with_clause (that is, the with_clause that contains the PARTITION**=** clause).

Note: If tables are defined with thousands of partitions, Ingres should be properly configured. Ensure that dmf_tcb_limit is large enough for the expected number of active tables+indexes+partitions. Also ensure that the DMF and RDF cache and memory settings are large enough. Having many partitions makes similar demands on the DBMS Server as does having many tables.

Partitioning Syntax

A table partition definition has the following format:

```
PARTITION = ( dimension )  |  (  ( dimension )  { SUBPARTITION (dimension)} )
```

The syntax for each partition dimension is:

```
dimension = rule [ ON column { , column } ]
                        partitionspec { , partitionspec }
```

rule

Defines the type of distribution scheme for assigning rows to partitions. Valid values are:

AUTOMATIC

Assigns rows arbitrarily to random partitions. The ON clause cannot be used with automatic distribution.

HASH

If rule is specified as automatic or hash, the syntax for *partitionspec* merely defines the number of partitions and optionally their names:

```
partitionspec = [nn] PARTITION[s] [ ( name {,  name} ) ] [with_clause]
```

The option number *nn* defaults to 1 if omitted.

LIST

Causes the syntax for *partitionspec* to define the list values to be mapped to each partition:

```
partitionspec = PARTITION [name] values ( listvalue {, listvalue} )
        [with_clause]
listvalue = single-constant-value
              |
              ( single-constant-value { , single-constant-value } )
              |
              default
```

The first form is used when there is only one partitioning column, while the second form is used when there are multiple partitioning columns.

RANGE

Causes the syntax for *partitionspec* to define the ranges that map to each partition:

```
partitionspec = PARTITION [name] values testing-op rangevalue
        [with_clause]
```

where *testing-op* is one of the operators: < , <= , > , or >=

```
rangevalue = single-constant-value
             |
             ( single-constant-value { , single-constant-value } )
```

Range values work like list values: parentheses are needed for multiple ON columns. Ranges do not allow the default keyword.

```
with_clause = WITH with-option
              | WITH   ( with-option { , with-option } )
with-option = LOCATION = ( location { , location } )
```

With_Clause for Create Table

The CREATE TABLE statement accepts the following with_clause options:

LOCATION = (*location_name* {, *location_name*})

Specifies the locations where the new table is created. To create locations, use the CREATE LOCATION statement. The location_names must exist and the database must have been extended to the corresponding areas. If the location option is omitted, the table is created in the default database location. If multiple location_names are specified, the table is physically partitioned across the areas. For details about defining location names and extending databases, see the *Database Administrator Guide*.

[NO]JOURNALING

Explicitly enables or disables journaling on the table. For details about journaling, see the *Database Administrator Guide*.

To set the session default for journaling, use the SET [NO]JOURNALING statement. The session default specifies the setting for tables created during the current session. To override the session default, specify the WITH [NO]JOURNALIING clause in the CREATE TABLE statement.

If journaling is enabled for the database and a table is created with journaling enabled, journaling begins immediately. If journaling is not enabled for the database and a table is created with journaling enabled, journaling begins when journaling is enabled for the entire database.

Note: To enable or disable journaling for the database and for system catalogs, use the ckpdb command. For information about ckpdb, see the *Command Reference Guide*.

[NO]DUPLICATES

Allows or disallows duplicate rows in the table. This option does not affect a table created as heap. Heap tables always accept duplicate rows regardless of the setting of this option. If a heap table is created and specified with NODUPLICATES, the heap table modified to a different table structure, the NODUPLICATES option is enforced. (By default, all structures accept duplicate rows.)

The DUPLICATES setting can be overridden by specifying a unique key for a table in the MODIFY statement.

PAGE_SIZE = *n*

Specifies a page size, in number of bytes. Valid values are described in Page_size Option (see page 431).

Default: 2048. The tid size is 4.

The buffer cache for the installation must also be configured with the page size specified in CREATE TABLE or an error occurs.

SECURITY_AUDIT = (*audit_opt* {, *audit_opt*})

Specifies row or table level auditing, as described in Security_audit Option (see page 432).

SECURITY_AUDIT_KEY = (*column*)

Writes an attribute to the audit log to uniquely identify the row in the security audit log. For example, an employee number can be used as the security audit key.

PARTITION =

Defines a partitioned table. For more information, see Partitioning Schemes (see page 426).

NOPARTITION

Indicates that the table is not to be partitioned. This is the default partitioning option.

Page_size Option

The PAGE_SIZE option on the WITH clause in the CREATE TABLE statement creates a table with a specific page size. This option has the following format:

PAGE_SIZE = *n*

where n is the number of bytes.

Valid values are shown in the Number of Bytes column in the following table:

Page Size	Number of Bytes	Page Header
2K	2,048	40
4K	4,096	76
8K	8,192	76
16K	16,384	76
32K	32,768	76
64K	65,536	76

Security_audit Option

The SECURITY_AUDIT option on the WITH clause specifies row- or table-level auditing.

This option has the following format:

```
SECURITY_AUDIT = (audit_opt {, audit_opt})
```

audit_opt

Specifies the level of security, as follows:

TABLE

(Default) Implements table-level security auditing on general operations (for example create, drop, modify, insert, or delete) performed on the table.

[NO]ROW

Implements row-level security auditing on operations performed on individual rows, such as insert, delete, update, or select. If NOROW is specified, the row-level security auditing is not implemented.

For example, an SQL delete statement that deleted 500 rows from a table with both table and row auditing generates the following audit events:

- One table-delete audit event, indicating the user issued a delete against the table.

- 500 row-delete audit events, indicating which rows were deleted.

Note: Either TABLE and ROW or TABLE and NOROW auditing can be specified. If NOROW is specified, row-level auditing is not performed. If either clause is omitted, the default installation row auditing is used. The default can be either ROW or NOROW depending on how your installation is configured.

WITH SECURITY_AUDIT_KEY Clause

The WITH SECURITY_AUDIT_KEY clause allows the user to specify an optional attribute to be written to the audit log to assist row or table auditing. For example, an employee number can be used as the security audit key:

```
create table employee (name char(60), emp_no integer)
with security_audit = (table, row),
        security_audit_key = (emp_no);
```

If no user-specified attribute is given and the table has row-level auditing, a new hidden attribute, _ii_sec_tabkey of type table_key system_maintained is created for the table to be used as the row audit key. Although any user attribute can be used for the security audit key (security_audit_key clause), we recommend that a short, distinctive value be used (such as a social security ID), allowing the user to uniquely identify the row when reviewing the security audit log. If an attribute longer than 256 bytes is specified for the security audit key, only the first 256 bytes are written to the security audit log.

With_Clause for Create Table...as Select

The CREATE TABLE...AS SELECT statement accepts the following options on the WITH clause:

ALLOCATION = *n*

Specifies the number of pages initially allocated for the table.

Limits: Integer between 4 and 8,388,607.

Default: 4

EXTEND = *n*

Specifies the number of pages by which the table is extended when more space is required.

Limits: Integer between 1 and 8,388,607

Default: 16

STRUCTURE = *structure*

Specifies the storage structure of the new table. Valid values include: BTREE, ISAM, HEAP, or HASH.

KEY = (*column_name {, column_name}*)

Specifies the columns on which your table is keyed. All columns in this list must also be specified in the subselect. Be advised that this option affects the way data is physically clustered on disk.

FILLFACTOR = *n*

Specifies the percentage of each primary data page that must be filled with rows (under ideal conditions). Fillfactor is not valid if a heap table is being created.

Note: Large fillfactors in combination with a non-uniform distribution of key values can cause a table to contain overflow pages, increasing the time required to access the table.

Limits: 1 to 100

MINPAGES = *n*

Specifies the minimum number of primary pages a hash table must have when created.

Limits: Minimum value is 1. Cannot exceed the value of MAXPAGES, if specified.

MAXPAGES = *n*

Specifies the maximum number of primary pages a hash table can have when created.

Limits: Minimum value is 1.

COMPRESSION [= ([[NO]KEY] [,[NO]DATA])] | NOCOMPRESSION

Specifies whether the key or data is to be compressed. If compression is specified, the structure clause must be specified.

LEAFFILL = *n*

Specifies the maximum percentage full for leaf index pages (B-tree tables only). Leaf pages are the index pages directly above the data pages.

Default: 70

NONLEAFFILL = *n*

Specifies the maximum percentage full for nonleaf index pages (B-tree tables only).

Default: 80

PRIORITY = *n*

Specifies cache priority. If an explicit priority is not set for an index belonging to a base table to which an explicit priority has been assigned, the index inherits the priority of the base table.

Limits: Integer between 0 and 8, with 0 being the lowest priority and 8 being the highest.

Default: 0. (Causes the table to revert to a normal cache management algorithm.)

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

All users. Using the grant statement, the DBA can control whether specific users, groups, or roles can create tables.

Locking

The DBMS Server takes an exclusive table lock when creating a table, which prevents other sessions—even those using READLOCK=NOLOCK—from accessing the table until the transaction containing the CREATE TABLE statement is committed.

Related Statements

Alter Table (see page 305)

Create Index (see page 353)

Create Integrity (see page 361)

Create Location (see page 363)

Drop (see page 482)

Grant (privilege) (see page 534)

Help (see page 551)

Modify (see page 579)

Select (interactive) (see page 641)

[No]Journaling (see page 674)

Examples: Create Table

The following are CREATE TABLE statement examples:

1. Create the employee table with columns eno, ename, age, job, salary, and dept, with journaling enabled.

```
create table employee
    (eno smallint,
     ename varchar(20) not null with default,
     age integer1,
     job smallint,
     salary float4,
     dept smallint)
  with journaling;
```

2. Create a table with some other data types.

```
create table debts
    (acct varchar(20) not null not default,
     owes money,
     logical_key object_key with system_maintained,
     due date not null with default);
```

3. 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);
```

4. Create a table that spans two locations. Specify number of pages to be allocated for the table.

```
create table emp as
      select eno from employee
      with location = (location1, location2),
      allocation = 1000;
```

5. Create a table specifying defaults.

```
create table dept (
  dname char(10)
  location char(10) default 'LA'
  creation_date      date default '1/1/93',
  budget money default 100000,
  expenses money default 0);
```

6. Create a table specifying check constraints. In the following 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 money default 100000,
    expenses money default 0,
    check (budget >= expenses));
```

7. Create a table specifying unique constraints and keys.

```
create table dept (
    deptno char(5) primary key,
    dname char(10) not null,
    dlocationchar(10) not null,
unique (dname, dlocation));
```

8. Create a table specifying null constraints.

```
create table emp (
    salary money with null not default,
    hiredate date with null not default,
    sickdays float with null with default 5.0);
```

9. Primary key constraint uses hash index structure instead of B-tree.

```
create table department (dept_id char(6) not null primary key with structure
= hash, dept_name char(20));
```

10. Base table structure is hash unique on dept_id.

```
create table department (dept_id char(6) not null, dept_name char(20));
modify department to hash unique on dept_id;
```

11. Force ANSI uniqueness semantics.

```
modify department to unique scope = statement;
```

12. Unique constraints use base table structure, not a generated index.

```
alter table department add primary key (dept_id)
    with index = base table structure;
```

13. Unique constraints generate 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));
```

14. 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);
```

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);
```

15. Create an automatically-distributed, partitioned table with four partitions. The last partition is placed in the database's default data location.

```
CREATE TABLE foo (
    i      INTEGER NOT NULL WITH DEFAULT,
    str    VARCHAR(10) NOT NULL WITH DEFAULT
) WITH JOURNALING,
    PARTITION = (AUTOMATIC
        PARTITION p1 WITH LOCATION=(ii_database),
        2 PARTITIONS WITH LOCATION=(loc1, loc2),
        PARTITION p4);
```

16. Create a list-distributed, partitioned table. Partition p1 is stored in location ii_database. Partition p2 is stored in locations loc1 and loc2. Partition p3 is stored in the database default data location.

```
CREATE TABLE table1 (
    i INTEGER NOT NULL WITH DEFAULT,
    str VARCHAR(10) NOT NULL WITH DEFAULT
) WITH PARTITION = (LIST ON i,str

  PARTITION p1 VALUES (1,'one')

      WITH LOCATION=(ii_database),
  PARTITION p2 VALUES ((2,'two'),(3,'three'))
      WITH LOCATION=(loc1, loc2),
  PARTITION p3 VALUES DEFAULT);
```

17. Create a range distribution table partition. Partition p1 contains all values less that A and is stored in location ii_database. Partition p2 contains all values between 'A' and 'Z' (inclusive) and is stored in locations loc1 and loc2. Partition p4 contains all values greater than 'Z' and is stored in the database default location.

```
CREATE TABLE range (
    i    INTEGER NOT NULL WITH DEFAULT,
    str   VARCHAR(10) NOT NULL WITH DEFAULT
) WITH PARTITION = (RANGE ON str
        PARTITION p1 VALUES < 'A'
WITH LOCATION=(ii_database),
        PARTITION p2 VALUES <= 'Z'
WITH LOCATION=(loc1, loc2),
        PARTITION p4 VALUES > 'Z');
```

18. Create a range distribution, sub-partitioned table using hash. In this example, the physical partitions are all stored in location loc2. There are 32 hash partitions, 8 for each of the ranges.

```
CREATE TABLE lineitems (
    shipdate  DATE NOT NULL WITH DEFAULT,
    partno    INTEGER NOT NULL WITH DEFAULT,
    stuff     VARCHAR(10) NOT NULL WITH DEFAULT
) WITH PARTITION = (
    (RANGE ON shipdate
     PARTITION p1 VALUES <= '31-Dec-2001',
     PARTITION p2 VALUES <= '31-Dec-2002',
     PARTITION p3 VALUES <= '31-Dec-2003',
     PARTITION p4 VALUES > '31-Dec-2003')
    SUBPARTITION
(HASH ON partno 8 PARTITIONS WITH LOCATION = (loc2)));
    );
```

Create User

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The CREATE USER statement defines a new user.

Syntax

The CREATE USER statement has the following format:

```
[EXEC SQL] CREATE USER user_name
[WITH with_item {, with_item}]
with_item = NOPRIVILEGES| PRIVILEGES = ( priv {, priv} )
                            | NOGROUP | GROUP = default_group
                            | SECURITY_AUDIT= ( audit_opt {,audit_opt})
                            | NOEXPIREDATE | EXPIRE_DATE = 'expire_date'
                            | DEFAULT_PRIVILEGES = (priv {,priv})| ALL
                                    | NODEFAULT_PRIVILEGES
                            | NOPROFILE | PROFILE= profile_name
                            | NOPASSWORD | PASSWORD = 'user_password'
                                    | PASSWORD = X'encrypted_role_password'
                            | EXTERNAL_PASSWORD
```

user_name

Specifies the user name to be created. Must be a valid object name.

priv

Specifies one of the following subject privileges, which apply to the user regardless of the database to which the user is connected. If the privileges clause is omitted, the default is NOPRIVILEGES.

CREATEDB

Allows users to create databases.

TRACE

Allows the user to use tracing and debugging features.

SECURITY

Allows the user to perform security-related functions (such as creating and dropping users).

OPERATOR

Allows the user to perform database backups and other database maintenance operations.

MAINTAIN_LOCATIONS

Allows the user to create and change the characteristics of database and file locations.

AUDITOR

Allows the user to register or remove audit logs and to query audit logs.

MAINTAIN_AUDIT

Allows the user to change the ALTER USER SECURITY_AUDIT and ALTER PROFILE SECURITY_AUDIT privileges. Also allows the user to enable, disable, or alter security audit.

MAINTAIN_USERS

Allows the user to perform various user-related functions, such as creating or altering users, profiles, group and roles, and to grant or revoke database and installation resource controls.

default_group

Specifies the default group to which the user belongs. Must be an existing group. For details about groups, see Create Group (see page 352).

Note: To specify that the user is not assigned to a group, use the NOGROUP option. If the group clause is omitted, the default is NOGROUP.

audit_opt

Defines security audit options:

ALL_EVENTS

All activity by the user is audited.

DEFAULT_EVENTS

Only default security auditing is performed, as specified with the ENABLE and DISABLE SECURITY_AUDIT statements. This is the default if the SECURITY_AUDIT clause is omitted.

QUERY_TEXT

Auditing of the query text associated with specific user queries is performed. Security auditing of query text must be enabled as a whole, using the ENALBE and DISABLE SECURITY_AUDIT statements with the QUERY_TEXT option (for example: ENABLE SECURITY_AUDIT QUERY_TEXT).

Default: DEFAULT_EVENTS if the security_audit clause is omitted.

expire_date

Specifies an optional expiration date associated with each user. Any valid date can be used. Once the expiration date is reached, the user is no longer able to log on. If the expire_date clause is omitted, the default is NOEXPIRE_DATE.

DEFAULT_PRIVILEGES =
(*priv* {, *priv*}) | ALL| NODEFAULT_PRIVILEGES

Defines the privileges initially active when connecting to Ingres. These must be a subset of those privileges granted to the user.

ALL

All the privileges held by the profile are initially active.

NODEFAULT_PRIVILEGES

No privileges are initially active. Allows default privileges to be removed.

profile_name

Allows a profile to be specified for a particular user. If the profile clause is omitted, the default is NOPROFILE.

user_password

Allows users to change their own password. If the oldpassword clause is missing or invalid the password is unchanged. In addition, users with the maintain_users privilege can change or remove any password.

EXTERNAL_PASSWORD

Allows a user's password to be authenticated externally to Ingres. The password is passed to an external authentication server for authentication.

Embedded Usage

In an embedded CREATE USER statement, specify the WITH clause using a host string variable (with :*hostvar*).

Permissions

You must have the MAINTAIN_USERS privilege and be connected to the iidbdb database.

Additional privileges are required to perform certain operations, as summarized here:

Action	Privilege Required
Set security audit attributes	maintain_audit

Locking

The CREATE USER statement locks pages in the iiuser system catalog.

Related Statements

Alter Profile (see page 294)

Alter User (see page 312)

Create Profile (see page 376)

Drop Profile (see page 491)

Drop User (see page 498)

Examples: Create User

The following are CREATE USER statement examples:

1. Create a new user, specifying group and privileges.

```
create user bspring with
    group = publishing,
    privileges = (createdb, security);
```

2. Create a new user, group and no privileges.

```
create user barney with
    group = sales,
    noprivileges;
```

3. Define user expiration date.

```
create user bspring
    with expire_date = '6-jun-1995'
```

4. Define an expiration date relative to the date the statement is executed.

```
create user bspring
    with expire_date = '1 month'
```

5. Specify no expiration date for a user.

```
create user bspring
    with noexpire_date
```

6. Create a user with a password.

```
create user bspring
    with password='mypassword';
```

7. Create a user with several privileges, and a smaller set of default privileges.

```
create user bspring
    with privileges=(write_down, write_fixed, trace,
      default_privileges = (trace);
```

8. Specify a profile for a particular user.

```
create user bspring with profile = dbop
```

where dbop is an existing profile.

9. Specify a user with an externally verified password.

```
create user bspring
    with external_password;
```

Declare

Valid in: DBProc

The DECLARE statement describes a list of local variables for use in a database procedure.

This statement is used only in a database procedure definition to declare a list of local variables for use in the procedure. If this statement is to be used, place it before the begin clause of the database procedure definition.

Nullable variables are initialized to null. Non-nullable variables are initialized to the default value according to data type: character data types are initialized to blank, and numeric data types are initialized to zero. Any non-nullable variables declared without an explicit default value are initialized to the default value.

The following table lists the effects of the null and default specifications on the default value of a column.

Nullability Option	Default Option	Results
with null	(none specified)	The variable can be null; default value is null.
not null	(none specified)	The default is 0 or blank (according to data type).
(none specified)	with default	Not valid without a null clause.
(none specified)	not default	Not valid without a null clause.
with null	with default	Not valid.
with null	not default	Not valid.
not null	with default	The variable defaults to 0 or blank, according to its data type.
not null	not default	The variable defaults to 0 or blank, according to its data type.

Syntax

The DECLARE statement has the following format:

```
DECLARE    var_name {, var_name} [=] var_type
           [NOTE NULL [WITH | NOT DEFAULT] | WITH NULL];
           {var_name {, var_name} [=] var_type
           [NOT NULL [WITH | NOT DEFAULT] | WITH NULL];}
```

var_name

Specifies the name of the local variable. A variable name must be unique within the procedure; it cannot match the name of any other procedure variable or parameter.

var_type

Is the data type of the variable. A local variable can be any data type except a system_maintained table_key or object_key.

Permissions

This statement is available to all users.

Related Statements

Create Procedure (see page 366)

Prepare (see page 605)

Example: Declare

The following example demonstrates some declarations and uses of local variables:

```
create procedure variables (vmny money not null) as
      declare
      vi4    integer not null;
      vf8    float;
      vc11   char(11) not null;
      vdt    date;
      begin
      vi4 = 1234;
      vf8 = null;
      vc11 = '26-jun-1957';
      vdt = date(:vc11);
      vc11 = :vmny;--data type conversion error
      vmny = :vf8;--null to non-null conversion
             error
      return :vi4;
end;
```

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 select_stmt
            [WITH CHECK OPTION]
```

view_name

Defines the name of the view. It must be a valid object name.

select_stmt

Is a SELECT statement, as described in the SELECT statement description in this chapter.

WITH CHECK OPTION (see page 450)

Prevents an INSERT or UPDATE to a view that creates a row that does not comply with the view definition.

Description

The CREATE VIEW statement uses a SELECT statement to define the contents of a virtual table. The view definition is stored in the system catalogs. When the view is used in an statement, the statement operates on the underlying tables. When a table or view used in the definition of a view is dropped, the view is also dropped.

Data can be selected from a view the same way data is selected from a base table. However, updates, inserts, and deletes on views are subject to several restrictions. Updates, inserts, and deletes are allowed only if the view meets all the following conditions:

- The view is based on a single updateable table or view.

- All columns see columns in the base table or view (no aggregate functions or derived columns are allowed).

- The SELECT statement omits distinct, group by, union, and having clauses.

Inserts are not allowed if a mandatory (not null not default) column in a base table is omitted from the view.

A maximum of 1024 columns can be specified for a view.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

With Check Option Clause

The WITH CHECK OPTION clause prevents you from executing an insert or update to a view that creates a row that does not comply with the view definition (the qualification specified in the WHERE clause).

For example, if the following view is defined WITH CHECK OPTION:

```
create view myview
    as select *
    from mytable
    where mycolumn = 10
    with check option;
```

And the following update is attempted:

```
update myview set mycolumn = 5;
```

The update to the mycolumn column is rolled back, because the updated rows fail the mycolumn = 10 qualification specified in the view definition. If the WITH CHECK OPTION is omitted, 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.

The WITH CHECK OPTION is valid only for updateable views. The WITH CHECK OPTION clause cannot be specified if the underlying base table is used in a subselect in the SELECT statement that defines the view. You cannot update or insert into a view defined on top of a view specified with WITH CHECK OPTION if the resulting rows violate the qualification of the underlying view.

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. Specify the WITH clause using a host string variable (with :*hostvar*).

Permissions

You must have all privileges required to execute the SELECT statements that define the view.

Locking

The CREATE VIEW statement requires an exclusive lock on the view's base tables.

Related Statements

Drop (see page 482)

Insert (see page 572)

Select (interactive) (see page 641)

Examples: Create View

The following are CREATE VIEW statement examples:

1. Define a view of employee data including names, salaries, and name of the manager.

```
create view empdpt (ename, sal, dname)
    as select employee.name, employee.salary,
    dept.name
    from employee, dept
    where employee.mgr = dept.mgr;
```

2. Define a view that uses aggregate functions to display the number of open orders and the average amount of the open orders for sales representative that has orders on file. This view is not updateable (because it uses aggregate functions).

```
create view order_statistics
    (sales_rep, order_count, average_amt)
      as select salesrep, count(*), avg(ord_total)
      from open_orders
      group by sales_rep;
```

3. Define an updateable view showing the employees in the southern division. Specify check option to prevent any update that changes the region or adds an employee from another region.

```
create view southern_emps
    as select * from employee
    where region = 'South'
    with check option;
```

Declare Cursor

Valid in: ESQL, OpenAPI

The DECLARE CURSOR statement associates a cursor name with a SELECT statement (see page 641).

Syntax

The DECLARE CURSOR statement has the following format:

```
EXEC SQL DECLARE cursor_name CURSOR
          FOR SELECT [ALL | DISTINCT] result_expression {, result_expression}
          FROM [schema.]table [correlation] {, [schema.]table [correlation]}
          [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 [DEFERRED | DIRECT] 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 32 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. Rows can be retrieved in any sequence if a scrollable cursor is declared.

4. Close the cursor by issuing the CLOSE statement.

You can use SELECT * in a cursor SELECT statement.

Scrollable Cursors

Note: Scrollable cursors are supported only in OpenAPI and JDBC.

Scrollable cursors are database query result sets that are maintained in an internal temporary table as long as the cursor is open and that allow the user to retrieve rows of a result set in any sequence.

Scrollable cursors allow an application to move backward and forward through the query results faster. Because scrollable cursors require more overhead than non-scrollable cursors, you should deploy scrollable cursors only where your application requires it.

The scrollable cursor can be declared as either a static or keyset type.

Static scrollable cursors are read only. The complete result set is stored in an internal temporary table.

Keyset scrollable cursors are updateable. Only the key columns and tids of the result set are stored in an internal temporary table.

Cursor Updates

Unless the cursor is explicitly opened in readonly mode or if table level lock granularity is in effect or if the transaction isolation level is read uncommitted, an update mode lock is obtained at the row or page level granularity, as appropriate. The granularity of locking used depends on many factors, including the effects of several set options, the estimated selectivity of the select criteria, various lock configuration parameters, and the page size used by the table. For a complete explanation, see the chapter on the locking system in the *Database Administrator Guide*.

If an update is performed, this lock is converted to an exclusive lock. If the cursor moves off the page or row without performing an update, the lock is converted to share mode if the isolation level is repeatable read or serializable, or the lock is released if the isolation level is read committed.

If the isolation level is read uncommitted, updates are implicitly forbidden, and no logical locks are taken on the table.

If isolation level is not read, uncommitted and table level lock granularity is used and the cursor was not opened in readonly, a single exclusive lock at the table level is taken.

If updates are not to be performed with the cursor, cursor performance can be improved by specifying for readonly when the cursor is opened (see page 601). (If the SELECT statement of the cursor contains one or more aggregate functions, the cursor is read-only.)

For details about updating or deleting table rows using a cursor, see Update (see page 702) and Delete (see page 470).

A cursor cannot be declared for update 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;
```

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 updateable column has been assigned a result column name using the syntax:

result_name **=** *column_name*

or:

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.

Cursor Modes

There are two update modes for cursors: DEFERRED and DIRECT.

DEFERRED

Specifies that cursor updates take effect when the cursor is closed. Only thereafter are the updates visible to the program that opened the cursor. The actual committal of the changes does not override or interfere with commit or rollback statements that can be executed subsequently in the program. Transaction semantics, such as the release of locks and external visibility to other programs, are not changed by using the deferred mode of update.

Only one update or delete against a row fetched by a cursor opened in the deferred mode can be executed. If an attempt to update such a row is made more than once, or if the row is updated and deleted, the DBMS Server returns an error indicating that an ambiguous update operation was attempted.

Only one cursor can be open at a time in the deferred mode.

DIRECT

Specifies that updates associated with the cursor take effect on the underlying table when the statement is executed, and 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 subsequently executed 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 can result in fetching the same row again at a later point.

Multiple update statements can be issued against a row that was fetched from a cursor opened in the direct mode. This enables a row to be updated and deleted.

Note: The default cursor mode is specified at the time the DBMS Server is started. For compliance with the ANSI/ISO SQL-92 standard, the default cursor mode must be DIRECT.

Embedded Usage

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, 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 OPEN statement of the cursor, because the SELECT is evaluated at that time; the variables do not need to have defined values at the point of the DECLARE CURSOR statement. Host language variables cannot be used to specify table, correlation, or column names.

Use the dynamic SQL syntax and specify a prepared statement name (see page 605) 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.

A source file can have multiple cursors, but the same cursor cannot be declared twice. To declare several cursors using the same host language variable to represent *cursor_name*, it is only necessary to declare the cursor once, because DECLARE CURSOR is a compile-time statement. Multiple declarations of the same *cursor_name*, even if its actual value is changed between declarations, cause a preprocessor error.

For example, the following statements cause a preprocessor error:

```
exec sql declare :cname[i] cursor for s1;
i = i + 1
/* The following statement causes a preprocessor
** error */
exec sql declare :cname[i] cursor for s2;
```

Declare the cursor only once; the value assigned to the host language variable *cursor_name* is determined when 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 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. 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, because the scope of a cursor declaration is the source file. If the cursor is re-declared in another file with the same associated query, it *still* does not identify the same cursor, not even at runtime. For example, if a cursor c1 is declared in source file, file1, 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 Server 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. In languages that do not allow empty control blocks, (for example, COBOL, which does not allow empty IF blocks), the DECLARE CURSOR statement must not be the only statement in the block.

Usage in OpenAPI

In OpenAPI, Declare Cursor functionality is achieved through query parameters to the IIapi_query() function.

Permissions

This statement is available to all users.

Locking

See the explanation in Cursor Updates (see page 705).

Related Statements

Close (see page 320)

Fetch (see page 523)

Open (see page 601)

Select (interactive) (see page 641)

Update (see page 702)

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
           [with_clause]
```

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
           [with_clause]
```

table_name

Defines the name of the temporary table. The SESSION schema qualifier is optional, as described in SESSION Schema Qualifier (see page 465).

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 641).

with_clause

Specifies parameters on the WITH clause. Multiple WITH clause parameters must be specified as a comma-separated list. For details about these parameters, see Create Table (see page 402).

Valid parameters for the *with_clause* are:

- LOCATION = (*locationname* {, *locationname*}**)**

- [NO]DUPLICATES

- ALLOCATION = *initial_pages_to_allocate*

- EXTEND = *number_of_pages_to_extend*

For temporary tables created using a subselect, the following additional parameters can be specified in the *with_clause*:

- STRUCTURE = HASH | HEAP | ISAM | BTREE

- KEY = (*columnlist*)

- FILLFACTOR = *n*

- MINPAGES = *n*

- MAXPAGES = *n*

- LEAFFILL = *n*

- NONLEAFFILL = *n*

- COMPRESSION[= ([[NO]KEY] [,[NO]DATA])] | NOCOMPRESSION

- PAGE_SIZE = *n*

- PRIORITY = *cache_priority*

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.

Temporary tables have the following characteristics:

- Temporary tables are visible only to the session that created them.

- Temporary tables are deleted when the session ends (unless deleted explicitly by the session). They do not persist beyond the duration of the session.

- Temporary tables can be created, deleted, and modified during an online checkpoint.

The SESSION schema qualifier on the table name is optional, as described in SESSION Sch (see page 465)ema Qualifier.

If the LOCATION parameter is omitted, the temporary table is located on the default database location (if the temporary table requires disk space). If the subselect is omitted, the temporary table is created as a heap.

When a transaction is rolled back, any temporary table that was in the process of being updated is dropped (because the normal logging and recovery processes are not used for temporary tables).

Note: If II_DECIMAL is set to comma, be sure that when SQL syntax requires a comma (such as 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 ti:
```

SESSION Schema Qualifier

Two syntaxes can be used for declaring and referencing global temporary tables:

With the SESSION Schema Qualifier

If the DECLARE GLOBAL TEMPORARY TABLE statement defines the table with the SESSION schema qualifier, then subsequent SQL statements that reference the table must use the SESSION qualifier.

When using this syntax, the creation of permanent and temporary tables with the same name is allowed.

Without the SESSION Schema Qualifier

If the DECLARE GLOBAL TEMPORARY TABLE statement defines the table without the SESSION schema qualifier, then subsequent SQL statements that reference the table can optionally omit the SESSION qualifier. This feature is useful when writing portable SQL.

When using this syntax, the creation of permanent and temporary tables with the same name is not allowed.

Note: In both modes, a session table is local to the session, which means that two sessions can declare a global temporary table of the same name and they do not conflict with each other.

Note: Syntaxes cannot be mixed in a single session. For example, if the table is declared with SESSION the first time, all declarations must use SESSION.

Embedded Usage

In an embedded DECLARE GLOBAL TEMPORARY TABLE statement:

- Host language variables can be used to specify constant expressions in the *subselect* of a CREATE TABLE...AS statement.

- *Locationname* can be specified using a host language string variable.

- The preprocessor does not validate the syntax of the WITH clause.

- 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.

Permissions

This statement is available to all users.

Restrictions

Temporary tables are subject to the following restrictions:

- Temporary tables cannot be used within database procedures.
- Temporary tables cannot be used in view definitions.
- Integrities, constraints, or user-defined defaults cannot be created for temporary tables. (The with|not null and with|not default can be specified.)
- A temporary table cannot be modified to use a different page size.
- Repeat queries referencing temporary tables cannot be shared between sessions.

All SQL statements can be used with temporary tables *except* the following:

- ALTER TABLE
- CREATE INTEGRITY
- CREATE PERMIT
- CREATE RULE
- CREATE SECURITY_ALARM
- CREATE SYNONYM
- CREATE VIEW
- GRANT
- HELP
- REVOKE
- SAVE
- SET JOURNALING
- SET LOCKMODE

The following commands cannot be issued with a temporary table name:

- auditdb
- copydb
- optimizedb
- statdump
- verifydb
- usermod
- genxml
- xmlimport

Related Statements

Create Table (see page 402)

Delete (see page 470)

Drop (see page 482)

Insert (see page 572)

Select (interactive) (see page 641)

Update (see page 702)

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
    emps
    (name char(20) , empno char(5))
    on commit preserve rows
    with norecovery,
    location = (personnel),
    [no]duplicates,
    allocation=100,
    extend=100;
```

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
    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 prepared SQL 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
```

Related Statements

Prepare (see page 605)

Example: Declare

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 [schema.]table_name TABLE
            (column_name data_type [WITH NULL | NOT NULL [WITH DEFAULT]]
            {, 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.

Permissions

This statement is available to all users.

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, DBProc, OpenAPI, ODBC, JDBC, .NET

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.

The DELETE statement does not automatically recover the space in a table left by the deleted rows. However, if new rows are added later, the empty space can be reused. To recover lost space after deleting many rows from a table, modify the table. To delete all rows from a table, use MODIFY...TO TRUNCATED (see page 579).

Syntax

The DELETE statement has the following formats:

Interactive and database procedure version:

```
[EXEC SQL] DELETE FROM [schema.]table_name [corr_name]
            [WHERE search_condition];
```

Embedded non-cursor version:

```
[EXEC SQL] [REPEATED] DELETE FROM [schema.]table_name [corr_name]
            [WHERE search_condition];
```

Embedded cursor version:

```
[EXEC SQL] DELETE FROM [schema.]table_name
            WHERE CURRENT OF cursor_name;
```

table_name

Specifies the table for which the constraint is defined.

corr_name

Specifies a correlation name for the table for use in the *search_condition*.

Embedded Usage

In an embedded DELETE statement, specify the cursor name with a string constant or a host language variable.

If the DELETE statement does not delete any rows, the sqlcode variable in the SQLCA structure is set to 100. Otherwise, the sqlerrd(3) variable in the SQLCA structure contains the number of rows deleted.

There are two embedded versions of the DELETE statement: the first is similar to the interactive version of the statement, and the second is for use with cursors.

Non-Cursor Delete

The non-cursor version of the embedded SQL 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 DBMS Server to save the execution plan of the DELETE statement 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 SQL 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 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, the DBMS Server generates an error.

To position the cursor to a row, issue a FETCH statement. After a deletion, the cursor points to a position after the deleted row, but before the next row, if any.

If the cursor is opened for direct update, the deletion takes effect immediately. If the cursor is opened for deferred update, the deletion takes effect when the cursor is closed. If the cursor is opened for deferred update, a row cannot be deleted after it has been updated. If an attempt is made to do so, the DBMS Server returns an error indicating an ambiguous update operation.

Both 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 continue in a loop to repeat the process. This process fails because the first commit closes the cursor.

A cursor delete can be executed dynamically using the PREPARE and EXECUTE statements. However, a cursor delete can only be prepared after the referenced cursor is opened. The prepared cursor delete remains valid while the cursor is open. If the named cursor is closed and reopened, the corresponding DELETE statement must be prepared again. If an attempt is made to execute the DELETE statement associated with the previously open cursor, the DBMS Server returns an error.

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.

Permissions

You must own the table or have delete permission. If the DELETE statement contains a WHERE clause, select and delete permissions are required; otherwise, delete permission alone is sufficient.

Locking

The DELETE statement locks pages as follows:

- If row level locking is in effect, victim rows are X locked, with a weaker IX lock held on the data page. Also X locks are held on any secondary index pages affected by the delete.

- If page level locking is in effect, the DELETE statement locks the pages containing the rows in the table that are evaluated against the WHERE clause of the statement. If secondary indexes exist, DELETE also locks the pages in the secondary indexes that have pointers to the deleted rows.

- If table level locking is in effect, a single lock on the base table is all that is needed. For a complete explanation of lock usage, see the *Database Administrator Guide*.

Related Statements

Declare Cursor (see page 451)

Fetch (see page 523)

Select (interactive) (see page 641)

Open (see page 601)

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.

For examples of the DESCRIBE statement and information about using the information it returns, see the *Embedded SQL Companion Guide*.

Usage in OpenAPI, ODBC, JDBC, .NET

The OpenAPI, ODBC, JDBC, and .NET interfaces cannot send a Describe statement to the DBMS. Each has its own mechanism for obtaining this information from the database. For example, JDBC uses ResultSet.getMetaData and OpenAPI uses IIapi_getDescriptor().

Permissions

This statement is available to all users.

Related Statements

Describe Input (see page 476)

Execute (see page 505)

Prepare (see page 605)

Describe Input

Valid in: ESQL

The DESCRIBE INPUT statement returns information about the input parameter markers of a prepared statement. (Input parameter markers are denoted by "?" in the prepared statement.)

When there is a direct correspondence to a table column, such as positional mapping or direct comparison, the type of the table column is returned. Otherwise, a best guess of the type based on local context (such as the nearest resolvable operand) is returned.

Note: Best-guess types are typically returned as nullable. If no type guess is possible because of lack of context, a zero (illegal) type is returned, but the DESCRIBE INPUT operation succeeds.

Syntax

The DESCRIBE INPUT statement has the following format:

```
EXEC SQL DESCRIBE INPUT statement_name
        USING [SQL] DESCRIPTOR :descriptor_name
        [WITHOUT NESTING]
```

statement_name

Specifies a valid prepared statement. Specify the *statement_name* using a string literal or a host language string variable. If the statement is prepared but has no input parameters, the DESCRIBE INPUT succeeds and returns zero for the returned SQLDA's sqld field.

descriptor-name

Identifies the name of the receiving descriptor area, formatted as 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. For information about the structure of an SQLDA and its allocation and inclusion in an embedded program, see the *Embedded SQL Companion Guide*.

WITHOUT NESTING

This optional noise phrase is included for Standards conformance. It has no effect on the operation of the statement.

The DESCRIBE INPUT 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 INPUT statement are complete and valid only if the number of the statement parameter markers is less than or equal to the number of allocated sqlvar elements.

Disable Security_Audit

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DISABLE SECURITY_AUDIT statement enables the security administrator to turn off security logging for the specified type of security event. To turn security logging on, use the enable security_audit statement.

Syntax

The DISABLE SECURITY_AUDIT statement has the following format:

```
[EXEC SQL] DISABLE SECURITY_AUDIT audit_type | ALL;
```

audit_type

Specifies the type of information to log, as follows:

ALARM

Disables logging of all security events generated by create security_alarm statements issued on tables.

DATABASE

Disables logging of all types of access by all users to all database objects, including use of the ckpdb, rollforwarddb, and auditdb utilities.

DBEVENT

Disables logging of all create dbevent, raise dbevent, register dbevent, remove dbevent, and drop dbevent statements.

LOCATION

Disables logging of all access to location objects (create location, alter location and drop location statements) by all users.

PROCEDURE

Disables logging of all access to database procedures (create procedure and drop procedure statements and procedure execution) by all users.

ROLE

Disables logging of role events (set role statement with -r flag)

RULE

Disables logging of rule events (create rule, drop rule, and firing of rules)

SECURITY

Disables logging of all types of access by all users to all security-related objects.

TABLE

Disables logging of all types of access by all users to all tables.

USER

Disables logging of all changes to user and group information, including runtime verification of user and group names.

VIEW

Disables logging of all types of access by all users to all views.

LEVEL ('*security_label*')

Disables logging of all types of access by all users to all events that dominate specific security audit labels.

ROW

Disables logging of all types of access by all users to all row-level events.

QUERY_TEXT

Disables logging of all types of access by all users to all the detail information for querytext events.

RESOURCE

Disables logging of all types of access by all users to violations of resource limits.

ALL

Disables logging of all types of security events.

After disabling security logging, the DBMS Server continues to log security events for users that have the AUDIT_ALL privilege. (To disable auditing for users that are assigned the AUDIT_ALL privilege, use Ingres configuration tools, described in the *System Administrator Guide*.)

Embedded Usage

You cannot use host language variables in an embedded DISABLE SECURITY_AUDIT statement.

Permissions

You must have the maintain_audit privilege and be working in a session that is connected to the iidbdb.

Locking

The DISABLE SECURITY_AUDIT statement locks pages in the iisecuritystate system catalog.

Related Statements

Create Security_Alarm (see page 395)

Drop Security_Alarm (see page 494)

Enable Security_Audit (see page 499)

Example: Disable Security_Audit

The following example turns off logging of database events:

```
disable security_audit dbevent;
```

Disconnect

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The DISCONNECT statement terminates a session connected to a database. The disconnect statement implicitly closes any open cursors, and commits any pending updates.

Syntax

The DISCONNECT statement has the following format:

```
EXEC SQL DISCONNECT [CURRENT] | connection_name |[SESSION session_identifier |
ALL];
```

CURRENT

Terminates the current session.

SESSION *session_identifier*

Terminates the specified session, which is other than the current session in a multi-session application. To determine the numeric session identifier for the current session, use the INQUIRE_SQL(:*session_id* = SESSION) statement.

connection_name

Terminates the specified connection, which is other than the current session in a multi-session application. To determine the connection name for the current session, use the INQUIRE_SQL(*connection_name*) statement.

ALL

Disconnects all open sessions.

If an invalid session is specified, the DBMS Server issues an error and does not disconnect the session.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, Disconnect is supported through interface calls. For example, JDBC uses Connection.close() and OpenAPI uses IIapi_disconnect().

Permissions

This statement is available to all users.

Locking

When the DISCONNECT statement is issued, all locks held by the session are dropped.

Related Statements

Connect (see page 325)

Set (see page 671)

Examples: Disconnect

The following are DISCONNECT statement examples:

1. Disconnect from the current database.

    ```
    exec sql disconnect;
    ```

2. Disconnect a session in a multi-session application by specifying the connection name.

    ```
    exec sql disconnect accounting;
    ```

3. Disconnect a session by specifying its session identifier.

    ```
    exec sql disconnect session 99;
    ```

4. On an error, roll back pending updates, 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.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

Syntax

The DROP statement has the following format:

```
[EXEC SQL] DROP objecttype [schema.]objectname {, [schema.]objectname};
```

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.

Description

The DROP statement removes the specified tables, indexes, and views from the database. Any synonyms and comments defined for the specified table, view, or index are also dropped. If the object is a table, any indexes, views, privileges, and integrities defined on that table are automatically dropped.

If the keyword indicating the object type is specified, the DBMS Server checks to make sure that the object named is the specified type. If more than one object is listed, only objects of the specified type are dropped. For example, if employee is a base table and emp_sal is a view on the base table salary, the following statement:

```
drop table employee, emp_sal;
```

drops only the employee base table (because the keyword TABLE was specified and emp_sal is a view, not a base table).

To drop a combination of table, views, and indexes in a single statement, omit the *objecttype* keyword. For example:

```
drop employee, emp_sal;
```

If an object that is used in the definition of a database procedure is dropped, all permits on the procedure are dropped (the procedure is not dropped). The procedure cannot be executed, nor can the execute privilege be granted on the procedure until all the objects required by its definition exist.

All temporary tables are deleted automatically at the end of the session.

Embedded Usage

You cannot use host language variables in an embedded DROP statement. However, the DROP statement can be used in an embedded EXECUTE IMMEDIATE statement.

Permissions

You must be the owner of a table, view, or index.

Locking

The DROP statement takes an exclusive lock on the specified table.

Related Statements

Create Table (see page 402)

Declare Global Temporary Table (see page 462)

Create View (see page 448)

Examples: Drop

The following are DROP statement examples:

1. Drop the employee and dept tables.

   ```
   drop table employee, dept;
   ```

2. Drop the salary table and its index, salidx, and the view, emp_sal.

   ```
   drop salary, salidx,
       accounting.emp_sal;
   ```

3. In an embedded program, drop two views.

   ```
   exec sql drop view tempview1, tempview2;
   ```

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;
```

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.

Permissions

You must be the owner of a 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.

Related Statements

Create Dbevent (see page 350)

Grant (privilege) (see page 534)

Raise Dbevent (see page 615)

Register Dbevent (see page 620)

Remove Dbevent (see page 625)

Example: Drop Location

The following example deletes the specified location:

```
drop location extra_work_disk;
```

Drop Group

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP GROUP statement removes the specified group identifiers from the installation. The DROP GROUP statement If any of the specified identifiers does not exist, the DBMS Server returns an error but does not abort the statement. Other valid existing *group_ids* in the statement are deleted.

A group identifier must be empty, that is, have no users in its user list, before it can be dropped. If an attempt is made to drop a group identifier that still has members in its user list, the DBMS Server returns an error and does not delete the identifier. However, the statement is not aborted. Other group identifiers in the list, if they are empty, are deleted. (Use the ALTER GROUP statement to drop all the users from a group's user list.)

Any session using a group identifier when the identifier is dropped continues to run with the privileges defined for that group.

For more information about group identifiers, see the *Database Administrator Guide*.

Syntax

The DROP GROUP statement has the following format:

```
[EXEC SQL] DROP GROUP group_id {, group_id};
```

Embedded Usage

In an embedded DROP GROUP statement, g*roup_id* cannot be specified using a host language variable.

Permissions

You must have the maintain_users privilege and be working in a session connected with the iidbdb.

Locking

The DROP GROUP statement locks pages in the iiusergroup catalog of the iidbdb.

Related Statements

Alter Group (see page 290)

Create Group (see page 352)

Examples: Drop Group

The following are DROP GROUP statement examples:

1. Drop the group identifier, acct_clerk.

   ```
   drop group acct_clerk;
   ```

2. In an application, drop the group identifiers, tel_sales and temp_clerk.

   ```
   exec sql drop group tel_sales, temp_clerk;
   ```

Drop Integrity

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP INTEGRITY statement removes the specified integrity constraints from a table.

When integrities are dropped from a table, the DBMS Server updates the date and timestamp of that table.

After integrities are dropped from a table, the DBMS Server recreates query plans for repeat queries and database procedures when an attempt is made to execute the repeat query or database procedure.

Note: The DROP INTEGRITY statement does not remove constraints defined using the CREATE TABLE and ALTER TABLE statements.

Syntax

The DROP INTEGRITY statement has the following format:

```
[EXEC SQL] DROP INTEGRITY ON table_name all | integer {, integer};
```

table_name

Specifies the name of the table for which integrity constraints are to be dropped.

ALL

Removes all the constraints currently defined for the specified table.

integer (,integer)

Removes individual constraints. To obtain the integer equivalents for integrity constraints, execute the HELP INTEGRITY statement.

Embedded Usage

In an embedded DROP INTEGRITY statement, t*able_name* or *integer* cannot be represented with host language variables.

Permissions

You must own the specified table.

Related Statements

Create Integrity (see page 361)

Examples: Drop Integrity

The following are DROP INTEGRITY statement examples:

1. Drop integrity constraints 1, 4, and 5 on job.

   ```
   drop integrity on job 1, 4, 5;
   ```

2. In an application, drop all the constraints against the exhibitions table.

   ```
   exec sql drop integrity on exhibitions all;
   ```

Drop Location

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP LOCATION statement deletes a name that was assigned to a physical disk location using the CREATE LOCATION statement.

Note: You can drop only a data or work location if no currently existing database has been extended to it. You cannot drop other types of locations.

Syntax

The DROP LOCATION statement has the following format:

```
[EXEC SQL] DROP LOCATION location_name;
```

Embedded Usage

You cannot use host language variables in an embedded DROP LOCATION statement.

Permissions

You must have the maintain_locations privilege and be working in a session that is connected to the iidbdb.

Locking

The DROP LOCATION statement locks pages in the iilocation_info catalog.

Related Statements: Drop Location

Alter Location (see page 292)

Create Location (see page 363)

Modify (see page 579)

Drop Procedure

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP PROCEDURE statement removes a database procedure definition from the database. Sessions that are executing the procedure are allowed to complete before the procedure's query plan is removed from memory.

If a procedure that is executed from another procedure is removed, the calling procedure is retained but marked dormant, and cannot be executed until the called procedure is restored.

Syntax

The DROP PROCEDURE statement has the following format:

```
[EXEC SQL] DROP PROCEDURE proc_name;
```

proc_name

Specifies the name of the procedure to be removed.

Embedded Usage

In an embedded DROP PROCEDURE statement, a host language variable cannot be used to represent *proc_name*.

Permissions

You must be the owner of a database procedure.

Related Statements

Create Procedure (see page 366)

Execute (see page 505)

Grant (privilege) (see page 534)

Example: Drop Procedure

The following example removes the procedure named salupdt:

```
drop procedure salupdt;
```

Drop Profile

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP PROFILE statement drops a user profile that is no longer needed.

User profiles are a set of subject privileges and other attributes that can be applied to a user or set of users. Each user can be given a profile, which is used to provide the default attributes for that user. A default profile, changeable by the system administrator, is provided to determine the default user attributes when no profile is explicitly specified.

This statement is available in dynamic SQL. It is not available in database procedures. There are no dynamic parameters in embedded SQL.

Syntax

The DROP PROFILE statement has the following format:

```
[EXEC SQL] DROP PROFILE profile_name [CASCADE | RESTRICT]
```

CASCADE

Specifies that any users with this profile have their profile reset to the default profile.

RESTRICT

(Default) Specifies that if any users have this profile the statement is rejected.

Permissions

You must have the maintain_user privilege and be connected to the iidbdb database.

Locking

The DROP PROFILE statement locks iiprofile.

Related Statements

Alter Profile (see page 294)

Create Profile (see page 376)

Example: Drop Profile

The following example drops the myprofile profile:

```
drop profile myprofile cascade
```

Drop Role

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP ROLE statement removes the specified role identifiers from the installation. Any session using a role identifier when the identifier is dropped continues to run with the privileges defined for that identifier. For more information about role identifiers, see the *Database* Administrator *Guide.*

Syntax

The DROP ROLE statement has the following format:

```
[EXEC SQL] DROP ROLE role_id {, role_id};
```

role_id

Must be an existing role identifier. If the list of *role_id*s contains any that do not exist, the DBMS Server returns an error for each non-existent *role_id*, but does not abort the statement. Others in the list that are valid, existing role identifiers, are removed.

Embedded Usage

In an embedded DROP ROLE statement, r*ole_id* cannot be represented with a host language variable.

Permissions

You must have the maintain_user privilege and be connected to the iidbdb database.

Locking

The DROP ROLE statement locks pages in the iirole catalog in the iidbdb.

Related Statements

Create Role (see page 380)

Alter Role (see page 298)

Example: Drop Role

The following example drops the sales_report role identifier:

```
drop role sales_report;
```

Drop Rule

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP RULE statement removes the specified rule from the database. (A rule is dropped automatically if the table on which the rule is defined is dropped.)

Syntax

The DROP RULE statement has the following format:

```
[EXEC SQL] DROP RULE [schema.]rulename;
```

Embedded Usage

In an embedded DROP RULE statement, a host language variable cannot be used to represent the rule name.

Permissions

You must be the owner of a rule.

Related Statements

Create Rule (see page 384)

Create Procedure (see page 366)

Example: Drop Rule

The following example drops the chk_name rule:

```
drop rule chk_name;
```

Drop Security_Alarm

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP SECURITY_ALARM statement deletes security alarms for a table, database, or current installation.

Syntax

The DROP SECURITY_ALARM statement has the following format:

```
[EXEC SQL] DROP SECURITY_ALARM ON [TABLE] table_name |
           DATABASE dbname | CURRENT INSTALLATION
ALL | integer {, integer} | alarm_name {, alarm_name}
```

ALL

Deletes all security alarms for the specified table, database, or current installation.

integer (, integer)

Deletes the security alarms identified by the specified numeric ID. To see the numeric ID, use the HELP SECURITY_ALARM statement.

alarm_name (, alarm_name)

Deletes the security alarms identified by the specified alarm name.

Embedded Usage

You cannot use host language variables in an embedded DROP SECURITY_ALARM statement.

Permissions

You must be the owner of the tables. To drop database or installation security alarms, you must have security privilege and be connected to the iidbdb.

Locking

The DROP SECURITY_ALARM statement locks the tables on which the security alarms were created, and the iirelation, iiqrytext, and iiprotect system catalogs.

Related Statements

Disable Security_Audit (see page 477)

Create Security_Alarm (see page 395)

Enable Security_Audit (see page 499)

Help Security_Alarm (see page 551)

Examples: Drop Security_Alarm

The following are DROP SECURITY_ALARM statement examples:

1. Delete a security alarm for the employee table.

   ```
   drop security_alarm employee 1;
   ```

2. Drop a table security alarm and an installation alarm.

   ```
   drop security_alarm on emp 2;
   drop security_alarm on current installation bad_update ;
   ```

Drop Sequence

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP SEQUENCE statement deletes a sequence from the database catalog.

For more information about sequences, see the *Database Administrator Guide.*

Syntax

The DROP SEQUENCE statement has the following format:

```
[EXEC SQL] DROP SEQUENCE [schema.]sequence_name;
```

sequence_name

Specifies an existing sequence.

The DBMS Server returns an error for each non-existent *sequence name* in a list, but does not abort the statement. Existing sequences for valid names in the list are removed.

Permissions

You must have the create_sequence privilege.

Locking and Sequences

In applications, sequences use logical locks that allow multiple transactions to retrieve and update the sequence value while preventing changes to the underlying sequence definition. The logical lock is held until the end of the transaction.

Related Statements

Alter Sequence (see page 303)

Create Sequence (see page 398)

Examples: Drop Sequence

The following example deletes sequence "XYZ":

```
drop sequence XYZ
```

Drop Synonym

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP SYNONYM statement deletes one or more synonyms from a database. A synonym is an alias (alternate name) for a table, view, or index; synonyms are created using the Create Synonym (see page 401)

Dropping a synonym causes the DBMS Server to re-compile any repeat query or database procedure that references the alias. Dropping a synonym has no effect on views or permissions defined using the synonym.

When a table, view, or index is dropped (using the drop statement), all synonyms that have been defined for it are dropped.

Syntax

The DROP SYNONYM statement has the following format:

```
[EXEC SQL] DROP SYNONYM [schema.]synonym_name {, [schema.]synonym_name};
```

Embedded Usage

You cannot use host language variables in an embedded DROP SYNONYM statement.

Permissions

To drop a synonym that resides in a *schema* owned by the session's effective user, omit the *schema* parameter. To drop a synonym that resides in a schema owned by the session's effective group or role, specify the *schema* parameter.

Locking

The DROP SYNONYM statement takes an exclusive lock on the object for which the synonym was defined.

Related Statements

Create Synonym (see page 401)

Example: Drop Synonym

The following example deletes a synonym for the authors table:

```
drop synonym writers;
```

Drop User

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The DROP USER statement deletes an existing user.

For details about users, see the *Database Administrator Guide.*

Users that own databases cannot be dropped. If a user that owns database objects is dropped, the objects are not dropped.

Syntax

The DROP USER statement has the following format:

```
[EXEC SQL] DROP USER user_name;
```

Embedded Usage

You cannot use host language variables in an embedded DROP USER statement.

Permissions

You must have the maintain_user privilege and be connected to the iidbdb database.

Locking

The DROP USER statement locks pages in the iiuser system catalog in the iidbdb.

Related Statements

Alter User (see page 312)

Create User (see page 440)

Grant (privilege) (see page 534)

Example: Drop User

The following example drops a user:

```
drop user betsy;
```

Enable Security_Audit

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The ENABLE SECURITY_AUDIT statement enables the security administrator to turn on security logging for the specified type of security event.

This statement cannot be issued from within a multi-statement transaction.

Syntax

The ENABLE SECURITY_AUDIT statement has the following format:

```
[EXEC SQL] RESTRICTIONS:ENABLE SECURITY_AUDIT audit_type | ALL;
```

audit_type

Specifies the type of information to log, as follows:

ALARM

Logs all security events generated by create security_alarm statements issued on tables.

DATABASE

Logs all types of access by all users to all database objects, including use of the ckpdb, rollforwarddb, and auditdb utilities.

DBEVENT

Logs all CREATE DBEVENT, RAISE DBEVENT, REGISTER DBEVENT, REMOVE DBEVENT, and DROP DBEVENT statements.

LOCATION

Logs all access to location objects (CREATE LOCATION, ALTER LOCATION, and DROP LOCATION statements) by all users.

PROCEDURE

Logs all access to database procedures (CREATE PROCEDURE and DROP PROCEDURE statements and procedure execution) by all users.

ROLE

Logs role events (SET ROLE -r statement)

RULE

Logs rule events (CREATE RULE, DROP RULE, and firing of rules)

SECURITY

Logs all types of access by all users to all security-related objects.

TABLE

Logs all types of access by all users to all tables.

USER

Logs all changes to user and group information, including runtime verification of user and group names.

VIEW

Logs all types of access by all users to all views.

LEVEL ('*security_label*')

Logs all types of access by all users to all events that dominate specific security audit labels.

ROW

Logs all types of access by all users to all row-level events.

QUERY_TEXT

Logs all types of access by all users to all the detail information for querytext events.

RESOURCE

Logs all types of access by all users to violations of resource limits.

ALL

Logs all types of security events.

For users that are assigned the AUDIT_ALL privilege (using the CREATE USER or GRANT statement), all security events are logged, regardless of the types of security logging enabled using the ENABLE SECURITY_AUDIT statement.

Embedded Usage

You cannot use host language variables in an embedded ENABLE SECURITY_AUDIT statement.

Permissions

You must have the maintain_audit privilege and be working in a session that is connected to the iidbdb.

Locking

The ENABLE SECURITY_AUDIT statement locks pages in the iisecuritystate system catalog.

Related Statements

Example: Enable Security_Audit

The following example turns on all forms of auditing:

```
enable security_audit all;
```

Enddata

Valid in: ESQL

The ENDDATA statement terminates retrieval of long varchar or long byte data in a data handler routine. Long varchar and long byte data is retrieved using the GET DATA statement.

Syntax

The ENDDATA statement has the following format:

```
EXEC SQL ENDDATA;
```

Permissions

This statement is available to all users.

Examples: Enddata

For examples of the ENDDATA statement in the context of a data handler routine, see the chapter "Working with Embedded SQL."

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;
```

Permissions

This statement is available to all users.

Related Statements

Begin Declare (see page 316)

Declare Table (see page 469)

Include (see page 560)

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 SQL select loops. A select loop (see page 664) 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.

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 code block that is syntactically nested within a select loop, the statement ends the code block as well as the select loop.

The statement must be terminated according to the rules of the host language.

Note: To find out how many rows were retrieved before the ENDSELECT statement was issued, check the sqlerrd(3) variable of the SQLCA. For details about the SQLCA, see the chapter "Working with Embedded SQL."

Permissions

This statement is available to all users.

Locking

If autocommit is off (default behavior), the ENDSELECT statement does not affect locking. All locks held before the ENDSELECT statement remain. If autocommit is on, the ENDSELECT statement ends the query and locks are dropped.

Related Statements

Select Loops (see page 664)

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 SQL 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 regular or delimited identifier or a host language variable. It must identify a valid prepared statement.

If the statement identified by *statement_name* is invalid, the DBMS 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 named statement.) Similarly, if the statement name refers to a cursor update or delete whose associated cursor is no longer open, the DBMS Server issues an error. For more information, see Update (see page 702) and Delete (see page 470).

USING *variable*

Must be used if question marks (?) are used in the prepared statement as placeholders for parameters to be specified at runtime. If the number and data types of the expressions specified by question marks in the prepared statement are known, use the USING *variable_list* alternative. The number of the variables listed must correspond to the number of question marks in the prepared statement, and each must be type-compatible with its usage in the prepared statement.

USING DESCRIPTOR *descriptor_name*

Must be used if the number and data types of the parameters in the prepared statement are not known until runtime.

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 (see page 510) statement.

To use long varchar columns as variables in the USING clause, specify a DATAHANDLER clause in place of the host language variable. For details about data handler routines, see the *Embedded SQL Companion Guide*.

The syntax for the datahandler clause is as follows:

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

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 parameters in the prepared statement are not known until runtime, the USING DESCRIPTOR alternative must be used. 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, as it is in this case, your application program must set the sqlvar array element type, length, and data area for each portion of the prepared statement that is specified by question marks prior to executing the statement. Your application program can use one of the following methods to 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 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 an 4GL 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.

In addition, the program must also correctly set the sqld field in the SQLDA structure.

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, 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 that allows a host structure variable when issued non-dynamically.

This statement must be terminated according to the rules of the host language.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, the applications must use the interface-specific mechanism for executing a prepared statement, rather than sending an EXECUTE statement.

Permissions

This statement is available to all users.

Locking

The locking behavior of the EXECUTE statement depends on the statement that is executed.

Related Statements

Describe (see page 474)

Prepare (see page 605)

Examples: Execute

The following are EXECUTE statement examples:

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 can 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;
    ```

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 the following statements:

```
exec sql prepare statement_name
     from :statement_buffer;
exec sql execute statement_name;
'Forget' the statement_name;
```

The EXECUTE IMMEDIATE can be used:

- If a dynamic statement needs to be executed just once in your program

- When a dynamic SELECT statement is to be executed and the result rows are to be processed with a select loop

- In DROP statements, where the name of the object to be dropped is not known at the time the program is compiled

If the statement string is to be executed 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 "Working with Embedded SQL."

The EXECUTE IMMEDIATE statement must be terminated according to the rules of the host language. If the statement string is blank or empty, the DBMS Server returns a runtime syntax error.

The following SQL statements cannot be executed using EXECUTE IMMEDIATE:

- CALL

- CLOSE

- CONNECT

- DECLARE

- DISCONNECT

- ENDDATA

- FETCH

- GET DATA

- GET DBEVENT

- HELP

- INCLUDE

- INQUIRE_SQL

- OPEN

- PREPARE TO COMMIT

- PUT DATA

- SET_SQL

- WHENEVER

- Other dynamic SQL statements

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 a string that includes quotes as a string constant is to be specified, remember that quoted characters within the statement string must follow the SQL string delimiting rules.

If your host language delimits strings with double quotes, the quoted characters within the statement string must be delimited by the SQL 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. Use this option when the program knows the data types and lengths of the result columns before the SELECT executes. The data type of the variables must be compatible with the associated result columns. For information about the compatibility of host language variables and SQL data types, see the *Embedded SQL Companion Guide*.

Note: To use long varchar variables in the INTO clause, specify a DATAHANDLER clause in place of the host language variable. The syntax for the DATAHANDLER clause is as follows:

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

If the program does not know the types and lengths of the result columns until runtime, specify the USING clause. 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 specifying the USING clause, the result column values are placed in the variables to which the sqlvar elements point.

If the USING clause is to be used, 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 use that information to allocate the necessary variables before executing the select.

If the SELECT statement returns more than one row, include the BEGIN and END statement block. This block defines a select loop. The DBMS Server processes each row that the select returns using the program code that you specify 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 a select loop is not supplied, the application receives only the first row and an error to indicate that others were returned but unseen.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, the EXECUTE IMMEDIATE statement is equivalent to executing an SQL statement in a string variable without preparing the statement first, so the functionality is supported while the syntax is not.

Permissions

This statement is available to all users.

Locking

The locking behavior of the EXECUTE IMMEDIATE statement is dependent on which statement is executed.

Related Statements

Execute (see page 505)

Prepare (see page 605)

Examples: Execute Immediate

The following are EXECUTE IMMEDIATE statement examples:

1. The following example saves a table until the first day of the next year. Next_year and current_year are integer variables.

```
/* There is no need for a FROM clause in this
** SELECT
*/
exec sql select date_part('year', date('now'))
    into :current_year;

next_year = current_year + 1;

statement_buffer = 'save ' + table_name +
    ' until Jan 1 ' + next_year;
exec sql execute immediate :statement_buffer;
```

2. The following example reads an SQL statement from the terminal into a host string variable, statement_buffer. If the statement read is "quit," the program ends. If an error occurs upon execution, 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})] |
              [(parm = SESSION.global temporary table_name)]
              [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.

global temporary table_name

> Is the name of a global temporary table already declared in the session in which the execute procedure is issued; it *must* be preceded by the SESSION qualifier.

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, specify *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, specify the parameter as *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.

Temporary Table Parameter

The temporary table must have been declared prior to procedure execution. However, it does not have to be populated (because the procedure itself can place rows into the table). Upon invocation of the procedure, Ingres binds the executing procedure unambiguously to the global temporary table instance of the invoking session. This permits any number of users, each with their own temporary table instance, to execute the procedure concurrently.

Example:

```
execute procedure gttproc (parm1 = session.mygtt1);
```

This statement invokes the procedure gttproc, passing the global temporary table session.mygtt1 as its parameter. (The name used for the actual parameter is inconsequential.)

Limitations of Temporary Table Parameter

When a global temporary table is passed as a procedure parameter, it must be the *only* parameter in both the calling and called parameter list (that is, in both the EXECUTE PROCEDURE and CREATE PROCEDURE statements).

The columns of the temporary table declaration and the elements in the set of parameter definition must exactly match in degree (number), name, type, and nullability. A check is performed during the execute procedure compile to assure that this constraint is met.

Temporary table parameters *cannot* be used in nested procedure calls. Global temporary tables cannot be declared within a procedure; hence no locally created temporary table can be passed in an EXECUTE PROCEDURE statement nested in another procedure. Likewise, a set of parameter cannot be specified in a nested EXECUTE PROCEDURE statement.

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;"
```

Usage in OpenAPI, ODBC, JDBC, .NET

The applications should use the interface-specific mechanism, rather than sending EXECUTE PROCEDURE in SQL.

Permissions

To execute a procedure that you do not own, you must specify the *schema* parameter, and must have execute privilege for the procedure.

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.

Related Statements

Create Procedure (see page 366)

Drop Procedure (see page 490)

Grant (privilege) (see page 534)

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 [FROM] cursor_name
            INTO variable[:indicator_var] {, variable[:indicator_var]};
```

Dynamic version:

```
EXEC SQL FETCH [FROM] cursor_name USING DESCRIPTOR descriptor_name;
```

FROM *cursor_name*

> Specifies the name of 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."

Note: To retrieve long varchar columns, specify a DATAHANDLER clause in place of the host language variable. For more information, see Data Handlers for Large Objects (see page 207) and the *Embedded SQL Companion Guide*. The syntax for the DATAHANDLER clause is as follows:

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

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.

Readonly Cursors and Performance

The performance of the FETCH statement is improved if the cursor associated with the statement is opened as a read-only cursor.

For read-only cursors, the DBMS Server prefetches rows to improve performance. To disable prefetching or specify the number of rows that are prefetched, use the SET_SQL(PREFETCHROWS) statement.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, the fetch concept is supported through interface-specific calls.

Permissions

This statement is available to all users.

Related Statements

Delete Cursor (see page 451)

Open (see page 601)

Close (see page 320)

Select (interactive) (see page 641)

Update (see page 702)

Examples: Fetch

The following are FETCH statement examples:

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 for readonly;

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 can be written as:

```
exec sql fetch cursor1
    into :emprec;
```

The preprocessor interprets 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:salary_ind;
```

For-EndFor

Valid in: DBProc

The FOR - ENDFOR statements repeat a series of statements while a specified condition is true.

Syntax

The FOR - ENDFOR statement has the following format:

```
[label:] FOR select_stmt DO
            statement; {statement;}
            ENDFOR;
```

Description

The FOR - ENDFOR statement define a program loop driven by the rows retrieved by the *select_stmt*. These statements can only be used inside a database procedure. The SELECT statement must have an INTO clause so that it can return result values into local variables in the procedure. The statement list can include any series of legal database procedure statements, including another for statement. The statement list is executed once for each row returned by the SELECT statement. After the last row from the result set of the SELECT statement is processed through the statement list, the for loop is terminated and execution continues with the first statement following the ENDFOR.

The ENDLOOP statement also terminates a FOR loop. When ENDLOOP is encountered, the loop is immediately closed, the remaining rows in the result set of the SELECT statement (if any) are discarded, and execution continues with the first statement following ENDFOR. For example,

```
        FOR select_1 DO

        statement_list_1

        IF condition_1 THEN

        ENDLOOP;

        ENDIF;

        statement_list_2;
ENDFOR;
```

In this case, *statement_list_1* and *statement_list_2* are executed once for each row returned by *select_1*. As soon as *condition_1* is true, *statement_list_2* is not executed in that pass through the loop, *select_1* is closed and the entire loop is terminated."

A FOR statement can be labeled. The label enables the ENDLOOP statement to break out of a nested series of FOR statements to a specified level. The label precedes FOR and is specified by a unique alphanumeric identifier followed by a colon, as in the following:

```
A: for...
```

The label must be a legal object name. The ENDLOOP statement uses the label to indicate which level of nesting to break out of. If no label is specified after ENDLOOP, only the innermost loop currently active is closed.

The following example illustrates the use of labels in nested FOR statements:

```
label_1:        FOR select_1 DO
                    statement_list_1
label_2:            FOR select_2 DO
                        statement_list_2
                        IF condition_1 THEN
                            ENDLOOP label_1;
                        ELSEIF condition_2 THEN
                            ENDLOOP label_2;
                        ENDIF;
                        statement_list_3
                    ENDFOR;
                    statement_list_4
                ENDFOR;
```

In this example, there are two possible breaks out of the inner loop. If *condition_1* is true, both loops are closed, and control resumes at the statement following the outer loop. If *condition_1* is false but *condition_2* is true, the inner loop is exited and control resumes at *statement_list_4*.

If an error occurs during the evaluation of a FOR statement, the database procedure terminates and control returns to the calling application.

Permissions

Anyone with permission to create procedures.

Example: For-EndFor

The following database procedure, avgsal_by_dept, returns rows containing the department name, average salary in the department and count of employees in the department. Any unexpected error from the SELECT statement terminates the loop:

```
create procedure avgsal_by_dept
       result row (char(15), float, int) as
declare
       deptname char(15);
       avgsal   float;
       empcount int;
       err      int;
begin
       err = 0;
       for select d.dept, avg(e.salary), count(*) into :deptname, :avgsal,
:empcount
       from department d, employee e
       where e.deptid = d.deptid
       group by d.deptid do
       if iierrornumber > 0 then
       err = 1;
       endloop;
       endif;
       return row(:deptname, :avgsal, :empcount);
       endfor;
return :err;
end"
```

Get Data

Valid in: ESQL

The GET DATA statement reads a segment of a long varchar, long nvarchar, or long byte column from a table to an embedded SQL program.

The GET DATA statement is used in data handler routines. For details about data handler routines, see the chapter "Working with Embedded SQL" and the *Embedded SQL* Companion *Guide*.

Syntax

The GET DATA statement has the following format:

```
EXEC SQL GET DATA(:col_value = SEGMENT
                [, :length_value = SEGMENTLENGTH]
                [, :dataend_value = DATAEND])
                [WITH MAXLENGTH = maxlength_value;]
```

col_value

Specifies the variable to which the value from the column is assigned. The maximum length of a long varchar, long nvarchar, or long byte column is 2 GB.

length_value

(Optional) Signed 4-byte integer to which the length of the data segment is assigned when the segment is read. If the MAXLENGTH parameter is specified, the value returned in the SEGMENTLENGTH variable, when the last segment is read, is less than MAXLENGTH . If the MAXLENGTH parameter is omitted, the value returned in the SEGMENTLENGTH variable is either the length of the segment variable or the number of remaining bytes, whichever is smaller.

dataend_value

(Optional) Signed 4-byte integer returns 1 if the segment is the last segment, 0 if the segment is not the last.

maxlength_value

(Optional) Signed 4-byte integer specifying the number of bytes to be returned. This value must not exceed the length of the SEGMENT variable. *Maxlength_value* can be specified using a literal or a host language variable.

The host language variables for *col_value*, *length_value*, *dataend_value* and *maxlength_value* must be declared in a BEGIN DECLARE SECTION/END DECLARE SECTION block to the ESQL preprocessor.

If a data handler routine attempts to exit without issuing an ENDDATA statement, the DBMS Server issues a runtime error.

Permissions

This statement is available to all users.

Related Statements

Put Data (see page 614)

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.

Usage in OpenAPI

In OpenAPI, the Get Dbevent functionality is supported through IIapi_catchEvent() and IIapi_getEvent().

Permissions

This statement is available to all users.

Related Statements

Create Dbevent (see page 350)

Drop Dbevent (see page 484)

Raise Dbevent (see page 615)

Register Dbevent (see page 620)

Remove Dbevent (see page 625)

Grant (privilege)

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The GRANT (privilege) statement grants privileges on the database as a whole or on individual tables, views, sequences or procedures. It controls access to database objects, roles, and DBMS resources. Details about using the GRANT statement with role objects is described in Grant (role) (see page 550).

To remove privileges, use the REVOKE statement. To determine the privileges in effect for a session, use the dbmsinfo function. In some cases granting a privilege imposes a restriction, and revoking the privilege removes the restriction. For example, GRANT NOCREATE_TABLE prevents the user from creating tables.

Note: The GRANT statement is the ISO/ANSI-compliant method for controlling access to database objects and resources.

To display granted database privileges, select data from the iidbprivileges system catalog. For details about system catalogs, see the *Database Administrator Guide.*

Syntax

The GRANT (privilege) statement has the following format:

```
[EXEC SQL] GRANT ALL [PRIVILEGES] | privilege {, privilege}
             [ON [object_descriptor] [schema.]object_name
                                      {, [schema.]object_name}]
             TO PUBLIC | [auth_type] auth_id {, auth_id} [WITH GRANT OPTION];
```

privilege

Specifies the type of privilege, as described in Types of Privileges (see page 536).

object_descriptor

Specifies the type of object on which you are granting privileges. *Object_descriptor* must be one of the following:

- TABLE

- PROCEDURE

- DATABASE

- DBEVENT

- SEQUENCE

- CURRENT INSTALLATION

Default: TABLE

The *object_descriptor* must agree with the privilege being granted (for example, execute privilege cannot be granted on a table).

Privileges cannot be defined for more than one type of object in a single GRANT statement. If *object_descriptor* is current installation, *object_name* must be omitted.

object_name

Specifies the name of the table, view, procedure, database event, sequence or database for which the privilege is being defined. The object must correspond to the *object_descriptor*. For example, if *object_descriptor* is table, *object_name* must be the name of an existing table or view.

auth_type

Specifies the type of authorization to which you are granting privileges. A GRANT statement cannot contain more than one *auth_type*. Valid *auth_types* are:

- USER

- GROUP

- ROLE

The *auth_ids* specified in the statement must agree with the specified *auth_type*. For example, if you specify *auth_type* as GROUP, all *auth_ids* listed in the statement must be group identifiers.

Default: USER

PUBLIC

Grants a privilege to all users. The *auth_type* parameter can be omitted.

auth_id

Specifies the name of the users, groups, or roles to which you are granting privileges, or PUBLIC. Both PUBLIC and a list of *auth_ids* can be specified in the same GRANT statement. If the privilege is subsequently revoked from PUBLIC, the individual privileges still exist.

Types of Privileges

Privileges are classed according to the type of objects they affect:

Table Privileges	Control access to individual tables or views
Database Privileges	Control access to database resources
Database Procedure Privileges	Control who can execute individual database procedures
Database Event Privileges	Control who can register for and raise specific database events
Database Sequence Privileges	Control who can retrieve values from individual database sequences

Table Privileges

Table privileges control access to tables and views. By default, only the owner of the table has privileges for the table. To enable others to access the table, the owner must grant privileges to specific authorization IDs or to public. Table privileges must be granted explicitly. The following table lists valid table privileges:

Privilege	Description
Select	Allows grantee to select rows from the table.
Insert	Allows grantee to add rows to the table.

Privilege	Description
Update	Allows grantee to change existing rows. To limit the columns that the grantee can change, specify a list of columns to allow or a list of columns to exclude.
	To grant the privilege for specific columns, use the following syntax after the update keyword in the GRANT statement:
	(column_name {, *column_name*)
	To grant the privilege for all columns except those specified, use the following syntax after the UPDATE keyword in the GRANT statement:
	EXCLUDING (*column_name* {, *column_name*})
	If the column list is omitted, update privilege is granted to all columns of the table or, for views, all updateable columns.
Delete	Allows grantee to delete rows from the table.
References	Allows grantee to create referential constraints that reference the specified tables and columns. For details about referential constraints, see Create Table (see page 402). A list of columns to allow or to exclude can optionally be specified.
	To grant the privilege for specific columns except those specified, use the following syntax after the REFERENCES keyword in the GRANT statement:
	(*column_name* {, *column_name*}**)**
	To grant the privilege for all columns except those specified, use the following syntax after the REFERENCES keyword in the GRANT statement:
	excluding (*column_name* {, *column_name*}**)**
	If the column list is omitted, references privilege is granted to all columns of the table. The references privilege cannot be granted on a view.
Copy_into	Allows the grantee to issue the COPY...INTO statement on a table. This privilege can be granted on tables only.
Copy_from	Allows the grantee to issue the COPY...FROM statement on a table. This privilege can be granted on tables only.

Privilege	Description
All [privileges]	All grants the subset of select, insert, update, delete, and references privileges for which the grantor has grant option. For details, see Grant All Privileges Option (see page 545).

When privileges are granted against a table, the date and timestamp of the specified table is updated, and the DBMS Server recreates query plans for repeat queries and database procedures that see the specified table.

Table Privileges for Views

The following table lists the privileges required to enable the owner of a view to grant privileges on the view:

Privilege	Description
Select	View owner must own all tables and views used in the view definition, or view owner or public must have grant option for select for the tables and views used in the view definition.
Insert	View owner must own all tables and views used in the view definition, or view owner or public must have grant option for insert for the tables and views used in the view definition.
Update	View owner must own all tables and updateable columns in views used in the view definition, or view owner or public must have grant option for update for the tables and updateable columns in views used in the view definition.
Delete	View owner must own all tables and views used in the view definition, or view owner or public must have grant option for delete for the tables and views used in the view definition.

To grant privileges for views the grantor does not own, the grantor must have been granted the specified privilege with grant option.
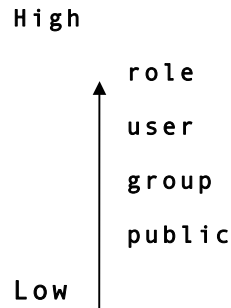
Database Privileges

Database privileges control the consumption of computing resources. Valid database privileges are:

- [no]access
- [no]create_procedure
- [no]create_sequence
- [no]create_table
- [no]db_admin
- [no]lockmode
- [no]query_cost_limit
- [no]query_cpu_limit
- [no]query_io_limit
- [no]query_page_limit
- [no]query_row_limit
- [no]update_syscat
- [no]select_syscat
- [no]connect_time_limit
- [no]idle_time_limit
- [no]session_priority
- [no]table_statistics

To override the default for a database privilege, grant a specific value to public. For example, by default, everyone (public) has the privilege to create database procedures. To override the default, grant nocreate_procedure to public, and grant the create_procedure privilege to any user, group, or role that you want to have this privilege. (Users, groups, and roles are referred to collectively as *authorization IDs*.)

The database privileges in effect for a session are determined by the values that were granted to the authorization IDs in effect for the session, according to the following hierarchy:

```
High

           role

           user

           group

           public

Low
```

For example, if different values for query_row_limit are granted to public, and to the user, group, and role that are in effect for a session, the value for the role of the session prevails.

This table describes database privileges in detail:

Privilege	Description
access	Access allows the specified authorization IDs to connect to the specified database. Noaccess prevents the specified authorization IDs from connecting.
create_procedure	Create_procedure allows the specified authorization IDs to create database procedures in the specified database. Nocreate_procedure prevents the specified users, groups, or roles from creating database procedures. By default, all authorization IDs can create database procedures.
create_sequence	Create_sequence allows the specified authorization IDs to create, alter and drop sequences in the specified database. Nocreate_sequence prevents the specified authorization IDs from creating sequences. By default, all authorization IDs can create, alter and drop sequences.
create_table	Create_table allows the specified authorization IDs to create tables in the specified database. Nocreate_table prevents the specified authorization IDs from creating tables. By default, all authorization IDs can create tables.

Privilege	Description
db_admin	Db_admin confers unlimited database privileges for the specified database and the ability to specify effective user (using the -u flag). A session that has the db_admin privilege does not have all the rights that a DBA has; some utilities can be run only by a DBA. The DBA of a database and users with the security privilege have the db_admin privilege by default. For all other users, the default is nodb_admin.
lockmode	Lockmode allows the specified authorization IDs to issue the set lockmode statement. Nolockmode prevents the specified users, groups, or roles from issuing the set lockmode statement. By default, everyone can issue the set lockmode statement.
query_cost_limit	Specifies the maximum cost per query on the database, in terms of disk I/O and CPU usage. By default, authorization identifiers are allowed an unlimited cost per query.
query_cpu_limit	Specifies the maximum CPU usage per query on the database. By default, authorization identifiers are allowed unlimited CPU usage per query.
query_io_limit	Query_io_limit specifies the maximum estimated number of I/O requests allowed for a single query for the specified authorization IDs when connected to the specified database. Integer must be a non-negative integer (or 0 to specify that no I/O is performed). Noquery_io_limit grants an unlimited number of I/O requests per query. Noquery_io_limit is the default.
query_page_limit	Specifies the maximum number of pages per query on the database. By default, authorization identifiers are allowed an unlimited number of pages per query.
query_row_limit	Query_row_limit *integer* specifies the maximum estimated number of rows returned by a single query for the specified authorization IDs when connected to the specified database. *Integer* must be a positive number (or 0 to specify that no rows are returned). Noquery_row_limit allows a single query to return an unlimited number of rows. Noquery_row_limit is the default.
update_syscat	Update_syscat allows the specified authorization IDs to update system catalogs when working in a session connected to the iidbdb.

Privilege	Description
select_syscat	Select_syscat allows a session to query system catalogs to determine schema information. When connected to the iidbdb database, this includes the master database catalogs such as iiuser and iidatabase. Select_syscat can be granted to user, group, role or public, and can only be issued when connected to the iidbdb database.
	This privilege restricts user queries against the core DBMS catalogs containing schema information, such as iirelation and iiattribute. Standard system catalogs such as iitables can still be queried.
connect_time_limit	Connect_time_limit limits the total connect time that a session can consume. The default is no limit, that is, a session can remain connected indefinitely. The connect time is checked periodically by the DBMS Server and if the limit has been exceeded for a session, it is disconnected, rolling back any open database transactions.
	The units are seconds. The maximum connection time limit is approximately 130 years. The minimum connection time limit is 1 second.
	As with other database privileges this can be granted to user, group, role or public, and can only be issued when connected to the iidbdb database.
Idle_time_limit	Idle_time_limit specifies the time that a session can take between issuing statements. The default is no limit, that is, a session can remain idle indefinitely without being disconnected. The idle time for each session is checked periodically by the DBMS Server, and if a session exceeds its idle time limit it is disconnected, rolling back any open database transactions.
	The units are seconds. The maximum idle time limit is approximately 130 years. The minimum idle time limit is 1 second. Idle_time_limit can be granted to user, group, role or public, and can only be issued when connected to the iidbdb database.

Privilege	Description
session_priority	Session_priority determines whether or not a session is allowed to change its priority, and if so what its initial and highest priority can be.
	If nosession_priority (the default) is specified, users can not alter their session priority.
	If session_priority is specified, users can alter their session priority, up to the limit determined by the privilege.
table_statistics	Table_statistics allows users to view (by way of SQL and statdump) and create (by way of optimizedb) database table statistics.
	If statistics exist in the database catalogs the DBMS Server automatically uses them when processing queries, even if the user does not possess this privilege.

Note: About query_cost_limit, query_cpu_limit, query_io_limit, query_page_limit, and query_row_limit: restrictions are enforced based on estimates from the DBMS query optimizer. If the optimizer predicts that a query consumes more I/O's than allowed by the session, the query is aborted prior to execution. The accuracy of the optimizer's estimates can be impeded by out-of-date or insufficient statistics about the contents of tables. For details about table statistics, see the description of the optimizedb command in the *Command Reference Guide* and the information on the query optimizer in the *Database Administrator Guide*.

Database Procedure Privileges

The execute privilege allows the grantee to execute the specified database procedures. To grant the execute privilege on database procedures, the owner of the procedure must have grant option for all the privileges required to execute the procedure. To grant the execute privilege on database procedures that the grantor does not own, the grantor must have execute privilege with grant option for the database procedure.

Database Event Privileges

Valid database event privileges are as follows:

- **Raise** - Allows the specified authorization IDs to raise the database event (using the RAISE DBEVENT statement).

- **Register** - Allows the specified authorization IDs to register to receive a specified database event (using the REGISTER DBEVENT statement).

Database Sequence Privileges

The next privilege allows the grantee to execute the next value and current value functions on the specified sequences. To grant the next privilege on sequences, the grantor must either own the sequence or have next privilege with grant option for the sequence.

Privilege Defaults

This table lists privilege defaults:

Privilege	Default
Select Insert Delete Update	Only the owner can perform select, insert, delete, or update operations on objects it owns.
References	Only the table owner can create referential constraints that see its tables.
Execute	Only the owner of a database procedure can execute the procedure.
Raise	Only the owner of a database event can raise the event.
Register	Only the owner of a database event can register to receive the event.
Next	Only the owner of a database sequence can execute the next value and current value operators on the sequence.

This table lists database privileges:

Privilege	Description
query_io_limit	Any user can perform unlimited I/O (noquery_io_limit)
query_row_limit	Any user can obtain unlimited rows (noquery_row_limit).
create_table	Any user can create tables (create_table).
create_procedure	Any user can create database procedures (create_procedure).

Privilege	Description
create_sequence	Any user can create database sequences (create_sequence).
lockmode	Any user can issue the set lockmode statement (lockmode).
db_admin	For a specified database, the DBA of the database and users that have the security privilege have the db_admin privilege. All other users of the database have nodb_admin privilege.

Grant All Privileges Option

The following sections describe the results of the GRANT ALL PRIVILEGES option.

Installation and Database Privileges

If GRANT ALL PRIVILEGES ON DATABASE or GRANT ALL PRIVILEGES ON CURRENT INSTALLATION is specified, the grantees receive the following database privileges:

- noquery_io_limit
- noquery_row_limit
- create_table
- create_procedure
- lockmode
- raise dbevent
- register dbevent

Privileges granted on a specific database override privileges granted on current installation.

Other Privileges

The requirements for granting all privileges on tables, views, database procedures, and database events depend on the type of object and the owner. To grant a privilege on an object owned by another user, the grantor or public must have been granted the privilege with grant option. Only the privileges for which the grantor or public has grant option are granted.

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 char(25), department char(5),
        salary money)...
```

and, 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.

Granting All Privileges on Views

The results of granting all privileges on a view you do not own are determined as shown in this table:

Privilege	Results
Select	Granted if the grantor can grant select privilege on all tables and views in the view definition.
Update	Granted for all columns for which the grantor can grant update privilege. If the grantor was granted update...with grant option on a subset of the columns of a table, update is granted only for those columns.
Insert	Granted if the grantor can grant insert privilege on all tables and views in the view definition.
Delete	Granted if the grantor can grant delete privilege on all tables and views in the view definition.
References	The references privilege is not valid for views.

Grant Option Clause

To enable an authorization ID to grant a privilege to another authorization ID, specify the WITH GRANT OPTION clause. The owner of an object can grant any privilege to any authorization ID (or to public). The authorization ID to whom the privilege is granted WITH GRANT OPTION can grant only the specified privilege. Any authorization ID can grant privileges that were granted to PUBLIC WITH GRANT OPTION to any other authorization ID.

The grant option cannot be specified for database privileges.

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 user laura did not specify the WITH GRANT OPTION clause, user evan cannot authorize another user to select data from tony.mytable. User laura can grant select privilege, but cannot grant, for example, insert privilege. If user tony revokes select permission from user laura (using the REVOKE statement), user tony must specify how the DBMS must handle any dependent privileges that user laura has issued.

The choices are:

- **REVOKE...CASCADE** - Revokes all dependent privileges. In the preceding example, select permission is revoked from user evan.

- **REVOKE...RESTRICT** - Do not revoke specified privileges if there are dependent privileges. In the preceding example, select privileges are not revoked from user laura because her grant to user evan depends on the privileges she received from user tony.

For more details, see "Ensuring Access Security" in the *Database Administrator Guide*.

Embedded Usage

In an embedded GRANT (privilege) statement, the WITH clause can be specified using a host string variable (with :*hostvar*).

Permissions

Database privileges are not enforced if the user has the security privilege or is the DBA of the current database. The GRANT statement can be executed by a user who is either the owner of the target object, or has been granted permission (using WITH GRANT OPTION) to use the statement on the specific target object by another user.

Locking

Granting privileges on a table takes an exclusive lock on that table. Granting privileges on the database as a whole locks pages in the iidbpriv catalog of the iidbdb.

Related Statements

Create Dbevent (see page 350)

Create Group (see page 352)

Create Procedure (see page 366)

Create Table (see page 402)

Create User (see page 440)

Create View (see page 448)

Grant (role) (see page 550)

Revoke (see page 630)

Examples: Grant (privilege)

The following are GRANT (privilege) statement examples:

1. Grant select and update privileges on the salary table to the group, acct_clerk.

   ```
   grant select, update on table salary
       to group acct_clerk;
   ```

2. 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;
   ```

3. Grant permission to the public to execute the monthly_report procedure.

   ```
   grant execute on procedure monthly_report
       to public;
   ```

4. Define a query_row_limit privilege of 100 rows on the new_accts database for users in the group, new_emp.

   ```
   grant query_row_limit 100 on database new_accts
       to group new_emp;
   ```

5. Grant unlimited rows to the role identifier, update_emp, which allows unlimited rows to be returned to any application that is associated with the role identifier, update_emp.

   ```
   grant noquery_row_limit on database new_acct
       to role update_emp;
   ```

6. Enable the inventory_monitor role to register for and raise the stock_low database event.

   ```
   grant register, raise on dbevent stock_low
       to role inventory_monitor
   ```

7. Enable any employee in accounting to change columns containing salary information.

   ```
   grant update on employee.salary, employee.socsec
       to group accounting;
   ```

8. Enable the accounting manager, rickr, complete access to salary information and to grant permissions to other user.

   ```
   grant all on employee to rickr with grant option;
   ```

9. Enable any user to create a table constraint that references the employee roster.

   ```
   grant references on emp_roster to public;
   ```

Grant (role)

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The GRANT (role) statement controls additional access to role objects created by the CREATE ROLE command. (When a role is created, an implicit grant is issued on the role to the user creating the role.) Role access can only be granted to specific users or to public.

Syntax

The GRANT (role) statement has the following format:

```
[EXEC SQL] GRANT role {, role}
            TO PUBLIC | [USER] auth_id {, auth_id};
```

Permissions

If the indicated roles have security audit attributes, the session must also have maintain_audit privilege.

Related Statements

Create Role (see page 380)

Grant (privilege) (see page 534)

Example: Grant (role)

The following example enables the user, bspring, to access sysop role:

```
grant sysop to bspring
```

Help

Valid in: SQL

The HELP statement displays information about database objects and SQL statements, including object name, owner, and type for all tables, views, and indexes to which the user has access, and all synonyms owned by the user. System tables and temporary tables are not listed. Information is displayed in a one-line-per-object format.

In general, to display high-level information, specify HELP *objectname* (for example, HELP mytable). To display detailed information, specify HELP *objecttype objectname* (for example, HELP TABLE mytable).

The asterisk wildcard character can be used in object name specifications to display information about a selected set of objects. For details, see Wildcards and Help (see page 554).

Syntax

The HELP statement has the following format:

```
HELP [[schema.]objectname {, [schema.]objectname}]
HELP COMMENT COLUMN [schema.]table column_name {, column_name}
HELP COMMENT TABLE [schema.]table {, [schema.]table }
HELP CONSTRAINT [schema.]table_name
                {, [schema.]constraint_name}
HELP DEFAULT [schema.]table_name
HELP HELP
HELP INDEX [schema.]indexname {, [schema.]indexname}
HELP INTEGRITY [schema.]table_name {, [schema.]table_name}
HELP PERMIT ON DBEVENT
                [schema.]objectname  {, [schema.]objectname}
HELP PERMIT ON PROCEDURE | TABLE | VIEW
                [schema.]object_name  {, [schema.]object_name}
HELP PROCEDURE [schema.]procedurename
                {, [schema.]procedurename}
HELP REGISTER [schema.]objectname
HELP RULE [schema.]rulename, {[schema.]rulename}
HELP SECURITY_ALARM [ON TABLE] table_name
HELP SECURITY_ALARM ON DATABASE database_name
HELP SECURITY_ALARM ON CURRENT INSTALLATION
HELP SQL [sql_statement]
HELP SYNONYM [schema.]synonym {, [schema.]synonym}
HELP TABLE [schema.]table_name {, [schema.]table_name}
HELP TABLE|INDEX name
HELP VIEW [schema.]view_name {, [schema.]view_name}
```

objectname

Specifies the name of a table, view, or index. Display format is block-style.

COMMENT COLUMN

Displays any comments defined for the specified columns.

COMMENT TABLE

Displays any comments defined for the specified tables.

CONSTRAINT

Displays any constraints defined on columns of the specified table or on the entire table. For details about table constraints, see Create Table (see page 402).

These constraints are not the same as the integrities displayed by the help integrities statement.

DEFAULT

Displays any user-defined defaults defined on columns of the specified table

HELP

Displays valid help statements.

INDEX

Displays detailed information about the specified indexes.

INTEGRITY

Displays any integrity constraints defined on the specified tables or indexes. Integrity constraints are defined using the create integrity statement.

PERMIT ON DBEVENT

Displays information about the specified database event.

PERMIT ON PROCEDURE | TABLE | VIEW

For tables, displays the permit text. For other objects, displays the values required by the corresponding drop permit statement.

PROCEDURE

Displays detailed information about the specified procedure.

REGISTER

Displays information about registered objects. For details about registering objects, see Register Table (see page 621).

RULE

Displays the text of the create rule statement that defined the rule.

SECURITY_ALARM [ON TABLE]

Displays all security alarms defined for the specified table. The information includes an index number that can be used to delete the security alarm (using the drop security_alarm statement).

SECURITY_ALARM ON DATABASE

Displays all security alarms defined for the specified database. The information includes an index number that can be used to delete the security alarm (using the drop security_alarm statement).

SECURITY_ALARM ON CURRENT INSTALLATION

Displays all security alarms defined for the current installation. The information includes an index number that can be used to delete the security alarm (using the drop security_alarm statement).

SQL

If the sql_statement parameter is omitted, a list of SQL statements displays for which help information is available. If the sql_statement parameter is specified, information displays about the specified statement.

SYNONYM

Displays information about the specified synonyms. To display all the synonyms you own, specify help synonym *. To display all the synonyms you own plus all the synonyms to which you have access, specify help synonym *.*.

TABLE

Displays detailed information about the specified tables.

TABLE | INDEX

Displays the cache priority.

VIEW

Displays detailed information about the specified views.

Wildcards and Help

The asterisk (*) wildcard can be used to specify all or part of the owner or object name parameters in a HELP statement. The HELP statement displays only objects to which the user has access, which are:

- Objects owned by the user

- Objects owned by other users that have granted permissions to the user

- Objects owned by the DBA to which you have access

If wildcards are specified for both the owner and object name (*.*), HELP displays all objects to which you have access. To display help information about objects you do not own, specify the owner name (using the *schema.objectname* syntax). If the owner name wildcard is omitted (that is, * is specified instead of *.*), HELP displays the objects that can be accessed without the owner prefix.

The following examples illustrate the effects of the wildcard character:

Wildcard	Description
HELP *	Display objects owned by the effective user of the session.
HELP davey.*	Display all objects owned by davey.
HELP *.mytable	Display all objects named, mytable, regardless of owner.
HELP d*.*	Display all objects owned by users beginning with d.
HELP *.d*	Display all objects beginning with d regardless of owner.
HELP *.*	Display all objects regardless of owner.

Permissions

This statement is available to all users.

Locking

The HELP statement does not take read locks on system catalogs. As a result, if the HELP statement is issued while a CREATE SCHEMA or CREATE TABLE AS SELECT statement is executing, the HELP statement can display results that do not reflect the final results of the CREATE statements.

Related Statements

Examples: Help

The following are HELP statement examples:

1. Display a list of all tables, views, and indexes to which the user has access.

   ```
   help;
   ```

2. Display help about all tables starting with "e" to which the user has access.

   ```
   help *.e*;
   ```

3. Display help about the employee and dept tables.

   ```
   help employee, dept;
   ```

4. Display the definition of the view, highpay.

   ```
   help view highpay;
   ```

5. Display all permits issued on the job and employee tables.

   ```
   help permit on table job, employee;
   ```

6. Display all integrity constraints issued on the dept and employee tables.

   ```
   help integrity dept, employee;
   ```

7. Display information on the SELECT statement.

   ```
   help sql select;
   ```

If-Then-Else

Valid in: DBProc

The IF-THEN-ELSE statement chooses between alternate execution paths inside a database procedure.

Syntax

The IF-THEN-ELSE statement has the following format:

```
IF boolean_expr THEN statement; {statement;}
            {ELSEIF boolean_expr THEN statement; {statement;}}
            [ELSE statement;{statement;}]
ENDIF
```

Description

The IF-THEN-ELSE statement can only be issued from within the body of a database procedure.

A Boolean expression (*boolean_expr*) must always evaluate to true or false. A Boolean expression can include comparison operators ('=', '<>', and so on) and the logical operators and, or, not. Boolean expressions involving nulls can evaluate to unknown. Any Boolean expression whose result is unknown is treated as if it evaluated to false.

If an error occurs during the evaluation of an if statement condition, the database procedure terminates and control returns to the calling application. This is true for both nested and non-nested if statements.

If Statement

The simplest form of the IF statement performs an action if the Boolean expression evaluates to true. The syntax follows:

```
IF boolean_expr THEN
        statement; {statement;}
ENDIF
```

If the Boolean expression evaluates to true, the list of statements is executed. If the expression evaluates to false (or unknown), the statement list is not executed and control passes directly to the statement following the ENDIF statement.

If...Then Statement

The second form of the IF statement includes the ELSE clause. The syntax follows:

```
IF boolean_expr THEN
        statement; {statement;}
ELSE
        statement; {statement;}
ENDIF
```

In this form, if the Boolean expression is true, the statements immediately following the keyword THEN are executed. If the expression is false (or unknown), the statements following the keyword ELSE are executed. In either case, after the appropriate statement list is executed, control passes to the statement immediately following ENDIF.

If...Then...Elseif Statement

The third IF variant includes the ELSEIF clause. The ELSEIF clause allows the application to test a series of conditions in a prescribed order. The statement list corresponding to the first true condition found is executed and all other statement lists connected to conditions are skipped. The ELSEIF construct can be used with or without an ELSE clause, which must follow all the ELSEIF clauses. If an ELSE clause is included, one statement list is guaranteed to be executed, because the statement list connected to the ELSE is executed if all the specified conditions evaluate to false.

The simplest form of this variant is:

```
IF boolean_expr THEN
        statement; {statement;}
ELSEIF boolean_expr THEN
        statement; {statement;}
ENDIF
```

If the first Boolean expression evaluates to true, the statements immediately following the first THEN keyword are executed. In such a case, the value of the second Boolean expression is irrelevant. If the first Boolean expression proves false, however, the next Boolean expression is tested. If the second expression is true, the statements under it are executed. If both Boolean expressions test false, neither statement list is executed.

A more complex example of the ELSEIF construct is:

```
IF boolean_expr THEN
        statement; {statement;}
ELSEIF boolean_expr THEN
        statement; {statement;}
ELSEIF boolean_expr THEN
        statement; {statement;}
ELSE
        statement; {statement;}
ENDIF
```

In this case, the first statement list is executed if the first Boolean expression evaluates to true. The second statement list is executed if the first Boolean expression is false and the second true. The third statement list is executed only if the first and second Boolean expressions are false and the third evaluates to true. Finally, if none of the Boolean expressions is true, the fourth statement list is executed. After any of the statement lists is executed, control passes to the statement following the ENDIF.

Nesting IF Statements

Two or more IF statements can be nested. In such cases, each IF statement must be closed with its own ENDIF.

This example illustrates nested IF statements in outline form:

```
if (condition1)

        code for condition1

            if (condition2)

                code for condition2

            endif /* condition2 */

        more code for condition1

endif /* condition1 */
```

Permissions

This statement is available to all users.

Example: If-Then-Else

The following IF statement performs a delete or an insert and checks to make sure the statement succeeded:

```
if (id > 0) and (id <= maxid) then
        delete from emp where id = :id;
        if iierrornumber > 0 then
                message 'Error deleting specified row';
                return 1;
        elseif iirowcount = 0 then
                message 'Specified row does not exist';
                return 2;
        endif;
elseif (id < maxid) then
        insert into emp values (:name, :id, :status);
        if iierrornumber > 0 then
                message 'Error inserting specified row';
                return 3;
        endif;
else
         message 'Invalid row specification';
         return 4;
endif;
```

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.

Permissions

This statement is available to all users.

Related Statements

Declare (see page 446)

Examples: Include

The following are INCLUDE statement examples:

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;
    ```

Inquire_sql

Valid in: ESQL

The INQUIRE_SQL statement provides an application program with a variety of run-time information.

Note: INQUIRE_SQL is a synonym for INQUIRE_INGRES.

Syntax

The INQUIRE_SQL statement has the following format:

```
EXEC SQL INQUIRE_SQL (variable = object {, variable = object});
```

Description

The INQUIRE_SQL statement enables an embedded SQL program to retrieve a variety of runtime information, such as:

- Information about the last executed database statement

- The message number and text from a message statement executed by a database procedure

- 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

- Information about the last event removed from the event queue

- The value of the last single logical key inserted into the database by the application

- Provides an application program with a variety of runtime information. (INQUIRE_SQL is a synonym for INQUIRE_INGRES.)

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). *variable* is the name of a program variable.

To retrieve error or message information about database procedure statements, issue the INQUIRE_SQL statement inside an error or message handler called by the WHENEVER SQLERROR or WHENEVER SQLMESSAGE statement.

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 Logical Keys

To obtain the last logical key added by an INSERT statement, issue the following INQUIRE_SQL statement:

```
exec sql inquire_sql

        (:key_variable:null_indicator = key_type)
```

where:

key_type is object_key or table_key.

This inquiry must be issued after an insert that adds a single row containing a logical key. In the case of the INSERT...AS SELECT statement, INQUIRE_SQL (:row_variable=rowcount) can be used to determine the number of rows added. INQUIRE_SQL cannot be used to obtain individual logical key values for multiple rows inserted as the result of an INSERT...AS SELECT statement.

A null indicator variable must be specified when inquiring about logical keys. INQUIRE_SQL returns the following values:

Null Indicator	Key Variable	Returned When..
0	Logical key value	Inquiry is issued after an insert statement that added a single row containing a system_maintained table or object key column.
-1	Unchanged	Inquiry is issued after: Inserting a row that did not contain the specified type of logical key A non-insert database statement An insert that failed, or added either 0 or more than 1 rows.

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 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	Description
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.

Object	Data Type	Description
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_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.

Object	Data Type	Description
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, 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."

Object	Data Type	Description
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) 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.

Object	Data Type	Description
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 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.

Object	Data Type	Description
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.

Permissions

This statement is available to all users.

Related Statements

Delete (see page 470)

Insert (see page 572)

Message (see page 577)

Raise Dbevent (see page 615)

Select (interactive) (see page 641)

Set_sql (see page 698)

Update (see page 702)

Examples: Inquire_sql

The following are INQUIRE_SQL statement examples:

1. 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 'INGRES error: ', sqlca.sqlcode;
print err_msg;
exec sql rollback;
```

2. The following example demonstrates the use of the WHENEVER statement for intercepting trace messages from a database procedure. The messages are written to a trace file.

```
exec sql whenever sqlerror stop;
exec sql whenever sqlmessage call trace_message;

exec sql execute procedure proc1 into :retstat;
...

/* Inside the "trace_message" host language
    procedure */
exec sql inquire_sql (:msgnum = messagenumber,
    :msgtxt = messagetext);

if (msgnum = 0) then

    print logfile, msgtxt;
```

```
        else

            print logfile, msgnum, '-'msgtxt;

        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 [schema.]table_name
            [(column {, column})]
            [VALUES (expr{, expr})] | [subselect];
```

Description

The INSERT statement inserts new rows into the specified table. Use 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*. For the syntax of *subselect*, see Select (interactive) (see page 641).

The *column* list identifies the columns of the specified table into which the values are placed. When including the column list, the DBMS Server places the result of the first expression 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 under the following circumstances:

- A subselect is specified 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 expressions 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.

A value cannot be specified to be inserted into a system_maintained table_key or object_key column (because the values for these data types are system-generated). For this reason, the column names must be specified when inserting into a table that has logical key columns.

When including the column list, any columns in the table that are not specified in the column list are assigned a default value or a null, depending on how the column was defined when the table was created. For details about column defaults, see Create Table (see page 402).

Expressions in the values list can only be constants (including the null constant), scalar functions on constants, or arithmetic operations on constants.

Note: To insert long varchar or long byte columns, specify a DATAHANDLER clause in place of the host language variable in the VALUES clause. For details about data handler routines, see the chapter "Working with Embedded SQL" and the *Embedded SQL Companion Guide*. The syntax for the datahandler clause is as follows:

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

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:
```

An INSERT/SELECT into a heap table that satisfies the bulk load criteria (not journaled and no secondary indexes) is executed with one bulk-load operation instead of multiple row-by-row inserts. This style of INSERT is faster and does much less transaction logging than row-by-row inserting. The inserted rows start on a new page, however, so many bulk-loaded INSERT/SELECTs of a few rows each may result in unexpected space usage in the result table.

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 columns in the subselect must correspond in sequence to the columns into which values are being inserted. For example:

```
insert into emps (entryclerk, sicktime, ename)
       select :yourname, 0, newename from newemps
```

In the previous example, the entryclerk column is filled from the form field, yourname, the sicktime column is initialized to 0 using a constant; and the ename column is filled from the names in the newename column of the newemps table.

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.

Permissions

You must own the table or have insert privilege. To insert into a view you must be able to insert into the underlying base tables or views. If you do not own the view, you must have insert privilege for the view.

Repeated Queries

The keyword repeated directs the DBMS Server to encode the insert and save its execution plan when it is first executed. This encoding can account for significant performance improvements on subsequent executions of the same insert.

Do not specify the repeated option for INSERT statements that are constructed using dynamic SQL. 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.

Error Handling

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), the sqlcode variable of the SQLCA is set to 100.

If an INSERT statement attempts to add a duplicate key to a column that has a unique constraint, a duplicate key error is returned, the current transaction is aborted, and any open cursors are closed.

Locking

Pages affected by the insert are locked in exclusive mode. When necessary, locking is escalated to the table level.

Related Statements

Delete (see page 470)

Select (interactive) (see page 641)

Update (see page 702)

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;
```

Message

Valid in: DBProc

The MESSAGE statement returns message text and a message number from a database procedure to an application program. The message statement can only be issued from a database procedure.

Syntax

The MESSAGE statement has the following format:

```
MESSAGE message_text | message_number | message_number message_text
              [WITH DESTINATION =([SESSION][, ERROR_LOG] [, AUDIT_LOG]]);
```

message_text

Specifies a string literal or a non-null host string variable.

Message_number

Specifies an integer literal or a non-null host integer variable.

Neither *message_text* nor *message_number* can be expressions. The values for these parameters do not correspond to DBMS Server error codes or messages; the message statement simply returns the specified values to the receiving application. If the *message_number* parameter is omitted, the DBMS Server returns a value of 0.

WITH DESTINATION

Changes the default destination of the message, which is a window at the bottom of the screen. Alternate destinations are as follows:

AUDIT_LOG

Directs the output of the message statement to the security audit log.

For example: WITH DESTINATION = (AUDIT_LOG)

ERROR_LOG

Directs the output of the message statement to the error log.

The message number and text is written to errlog.log with message identifier E_QE0300.

SESSION

Restores the default behavior.

To both log and return messages to the application, specify WITH DESTINATION = (SESSION, ERROR_LOG).

To specify an action to be performed when an application receives a message from a database procedure, use the WHENEVER SQLMESSAGE statement. For details, see Whenever (see page 708).

To specify a routine that is called when an application receives a message from a database procedure, use the SET_SQL(messagehandler) statement. For details, see Set_sql (see page 698).

Permissions

This statement is available to all users.

Related Statements

Create Procedure (see page 366)

Inquire_sql (see page 562)

Examples: Message

The following are MESSAGE statement examples:

1. The following example turns debugging text to the application.

```
message 'Inserting new row';
    insert into tab values (:val);
    message 'About to commit change';
    commit;
    message 'Deleting newly inserted row';
    delete from tab where tabval = :val;
    message 'Returning with pending change';
    return;
```

2. The following example returns a message number to the application. The application can extract the international message text out of a message file.

```
if iierrornumber > 0 then
        message 58001;
    elseif iirowcount <> 1 then
        message 58002;
    endif;
```

3. The following example sends a message to the error log file.

```
message 'User attempting to update payroll table'
    with destination = (error_log);
```

Modify

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The MODIFY statement changes properties of a table or index.

Syntax

The MODIFY statement has the following format:

```
[EXEC SQL] MODIFY [schema.]table_name|[schema.]indexname |
            [ schema.]table-name PARTITION partition-name { . partition-name }
            TO modify-action [UNIQUE]
            [ON column_name [ASC|DESC]{, column_name [ASC|DESC]}]
            [with_clause]
```

PARTITION *partition-name*

Allows the modify action to be applied specifically to the named partitions.

The PARTITION *partition-name* clause can be used with the following *modify-actions* only:

RECONSTRUCT, RELOCATE, REORGANIZE, MERGE, ADD_EXTEND, and TABLE_DEBUG. Use of TABLE_DEBUG against a partitioned table requires the PARTITION clause and a logical partition name for each dimension. (That is, the table debug operation can only operate on one specific physical partition.)

For a partitioned table with multiple dimensions, partition names are listed in the same order that dimensions were defined. It is not necessary to name a logical partition for every dimension; if a dimension is omitted, it is translated as "all logical partitions in this dimension."

Note: All other variants of the MODIFY statement can be applied only to the partitioned table as a whole, not to individual partitions.

The *modify-action* can be any one of the following action keywords:

ISAM

Modifies the table storage structure to the ISAM structure.

HASH

Modifies the table storage structure to the HASH structure

HEAP

Modifies the table storage structure to the HEAP structure

HEAPSORT

Modifies the table storage structure to the HEAP structure, and additionally sort the rows in the table as directed

BTREE

Modifies the table storage structure to the BTREE structure

RECONSTRUCT

Modifies the table storage structure to what it currently is (the table is physically rebuilt)

TRUNCATED

Truncates the table, deleting all data

REORGANIZE

Moves the data to a different location

RELOCATE

Moves the table to a different location

MERGE

Shrinks a btree index

ADD_EXTEND

Adds disk pages to the table

[NO]READONLY

Marks the table read only or not read only

PHYS_[IN]CONSISTENT

Marks the table physically consistent or inconsistent

LOG_[IN]CONSISTENT

Marks the table logically consistent or inconsistent

TABLE_RECOVERY_[DIS]ALLOWED

Allows or disallow table level rollforward

[NO]PERSISTENCE

Marks the index to be recreated automatically as needed (secondary indexes only)

UNIQUE_SCOPE = ROW | STATEMENT

Defines when uniqueness must be checked

TABLE_DEBUG

Displays internal table data structures

PRIORITY=*n*

Sets the table's buffer cache priority

The additional *action_keywords* CHEAP, CHASH, CISAM, and CBTREE are accepted. CHEAP is a synonym for HEAP with compression=(data), and the others similarly. These forms are deprecated; the WITH COMPRESSION= clause should be used instead.

One of the storage structure actions (HEAP, HASH, ISAM, BTREE) can be used instead of RECONSTRUCT.

UNIQUE

Requires each key value in the restructured table to be unique. This clause is used only with the ISAM, HASH, or BTREE *modify-actions*.

ON *column-name*

Determines the storage structure keys of the table. This clause is used only with ISAM, HASH, BTREE, or HEAPSORT actions.

with_clause

Specifies WITH clause parameters consisting of the word WITH followed by a comma-separated list of any number of the following items:

- ALLOCATION = *n*
- EXTEND = *n*
- FILLFACTOR = *n* (isam, hash, and btree only)
- MINPAGES = *n* (hash only)
- MAXPAGES = *n* (hash only)
- LEAFFILL = *n* (btree only)
- NONLEAFFILL = *n* (btree only)
- BLOB_EXTEND = *n* (btree only)
- NEWLOCATION = (*locationname* {, *locationname*})
- OLDLOCATION = (*locationname* {, *locationname*})
- LOCATION = (*locationname* {, *locationname*})
- COMPRESSION [= ([[no]key] [,[no|hi]data])] | nocompression
- [NO]PERSISTENCE
- UNIQUE_SCOPE = ROW | STATEMENT
- PAGE_SIZE = *n*
- PRIORITY = *cache_priority*
- NOPARTITION
- PARTITION = (*partitioning-scheme*)
- CONCURRENT_UPDATES

Use the syntax shown below to perform the listed operation:

- Reorganize a btree table's index:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO MERGE
  ```

- Move a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO RELOCATE
      WITH oldlocation = (locationname {, locationname}),
          newlocation = (locationname {, locationname}),
  ```

- Change locations for a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO REORGANIZE
      WITH LOCATION = (locationname {, locationname})
  ```

- Delete all data in a table:
  ```
  [EXEC SQL] modify table_name|indexname  to truncated
  ```

- Add pages to a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO ADD_EXTEND
      [WITH EXTEND = number_of_pages]
  ```

  where:

  number_of_pages is 1 to 8,388,607.

- Add pages to blob extension table:
  ```
  [EXEC SQL] MODIFY table_name|indexname WITH BLOB_EXTEND
      [WITH EXTEND = number_of_pages]
  ```

  where:

  number_of_pages is 1 to 8,388,607.

- Mark a table as physically consistent/inconsistent:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO PHYS_CONSISTENT|PHYS_INCONSISTENT
  ```

- Mark a table as logically consistent/inconsistent:
  ```
  [EXEC SQL] MODIFY table_name|indexname  TO LOG_CONSISTENT|LOG_INCONSISTENT
  ```

- Mark a table as allowed or disallowed for table-level recovery:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO
      TABLE_RECOVERY_ALLOWED|TABLE_RECOVERY_DISALLOWED
  ```

- Defer the uniqueness check until the end of statement execution:
  ```
  [EXEC SQL] MODIFY table_name TO UNIQUE_SCOPE = statement
  ```

- Mark a table as readonly:
  ```
  [EXEC SQL] MODIFY table_name TO [NO]READONLY
  ```

- Assign a table fixed cache priority:
  ```
  [EXEC SQL] MODIFY table_name TO PRIORITY = cache_priority
  ```

- Change a table's partitioning scheme:
  ```
  [EXEC SQL] MODIFY table_name TO RECONSTRUCT
  WITH PARTITION = ( partitioning-scheme )
  ```

- Enable a table modification to be performed online:
  ```
  [EXEC SQL] MODIFY table_name WITH CONCURRENT_UPDATES
  ```

Description

The MODIFY statement enables the following operations to be performed:

- Change the storage structure of the specified table or index

- Specify the number of pages allocated for a table or index, and the number of pages by which it grows when it requires more space

- Add pages to a table

- Reorganize a btree index

- Move a table or index, or portion thereof, from one location to another

- Spread a table over many locations or consolidate a table onto fewer locations

- Delete all rows from a table and release its file space back to the operating system

- Specify whether an index is recreated when its base table is modified

- Specify how unique columns are checked during updates: after each row is inserted or after the UPDATE statement is completed

- Mark table as physically or logically consistent or inconsistent

- Mark table as allowed/disallowed for table-level recovery

- Defer uniqueness check until the end of statement execution

- Mark a table as read only

- Assign a table fixed cache priority

- Change a table's partitioning scheme

- Enable modify table to be performed online

You can change a table's location and storage structure in the same MODIFY statement.

The MODIFY statement operates on existing tables and indexes. When modifying a table to change, truncate, or reconstruct the storage structure, the DBMS Server destroys any indexes that exist for the specified table (unless the index was created with persistence, or the table is a btree and the table being modified to reorganize its index).

Syntax for Modify Operations

Use the syntax shown below to perform the listed operation:

- Reorganize a btree table's index:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO MERGE
  ```

- Move a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO RELOCATE
      WITH oldlocation = (locationname {, locationname}),
          newlocation = (locationname {, locationname}),
  ```

- Change locations for a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO REORGANIZE
      WITH LOCATION = (locationname {, locationname})
  ```

- Delete all data in a table:
  ```
  [EXEC SQL] modify table_name|indexname  to truncated
  ```

- Add pages to a table:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO ADD_EXTEND
      [WITH EXTEND = number_of_pages]
  ```

  where:

  number_of_pages is 1 to 8,388,607.

- Add pages to blob extension table:
  ```
  [EXEC SQL] MODIFY table_name|indexname WITH BLOB_EXTEND
      [WITH EXTEND = number_of_pages]
  ```

  where:

  number_of_pages is 1 to 8,388,607.

- Mark a table as physically consistent/inconsistent:
  ```
  [EXEC SQL] MODIFY table_name|indexname TO PHYS_CONSISTENT|PHYS_INCONSISTENT
  ```

- Mark a table as logically consistent/inconsistent:
  ```
  [EXEC SQL] MODIFY table_name|indexname  TO LOG_CONSISTENT|LOG_INCONSISTENT
  ```

- Mark a table as allowed or disallowed for table-level recovery:

    ```
    [EXEC SQL] MODIFY table_name|indexname TO
        TABLE_RECOVERY_ALLOWED|TABLE_RECOVERY_DISALLOWED
    ```

- Defer the uniqueness check until the end of statement execution:

    ```
    [EXEC SQL] MODIFY table_name TO UNIQUE_SCOPE = statement
    ```

- Mark a table as readonly:

    ```
    [EXEC SQL] MODIFY table_name TO [NO]READONLY
    ```

- Assign a table fixed cache priority:

    ```
    [EXEC SQL] MODIFY table_name TO PRIORITY = cache_priority
    ```

- Change a table's partitioning scheme:

    ```
    [EXEC SQL] MODIFY table_name TO RECONSTRUCT
    WITH PARTITION = ( partitioning-scheme )
    ```

- Enable a table modification to be performed online:

    ```
    [EXEC SQL] MODIFY table_name WITH CONCURRENT_UPDATES
    ```

Storage Structure Specification

Changing the storage structure of a table or index is typically done to improve the performance when accessing the table. For example, to improve the performance of COPY, change the structure of a table to heap before performing a bulk copy into the table.

The *storage_structure* parameter must be one of the following:

ISAM

Indexed Sequential Access Method structure, duplicate rows allowed unless the WITH NODUPLICATES clause is specified when the table is created.

HASH

Random hash storage structure, duplicate rows allowed unless the WITH NODUPLICATES clause is specified when the table is created

HEAP

Unkeyed and unstructured, duplicated rows allowed, even if the WITH NODUPLICATES clause is specified when the table is created.

HEAPSORT

Heap with rows sorted and duplicate rows allowed unless the WITH NODUPLICATES clause is specified when the table is created (sort order not retained if rows are added or replaced).

BTREE

Dynamic tree-structured organization with duplicate rows allowed unless the WITH NODUPLICATES clause is specified when the table is created.

An index cannot be modified to HEAP, HEAPSORT, or RTREE.

The DBMS Server uses existing data to build the index (for isam and btree tables), calculate hash values (for hash tables) or for sorting (heapsort tables).

To optimize the storage structure of heavily used tables (tables containing data that is frequently added to, changed, or deleted), modify those tables periodically.

The optional keyword UNIQUE requires each key value in the restructured table to be unique. (The key value is the concatenation of all key columns in a row.) If UNIQUE is specified on a table that contains non-unique keys, the DBMS Server returns an error and does not change the table's storage structure. For the purposes of determining uniqueness, a null is considered to be equal to another null. Use UNIQUE only with isam, hash, and btree tables.

The optional ON clause determines the table's storage structure keys. This clause can only be specified when modifying to one of the following storage structures: isam, hash, heapsort, or btree. When the table is sorted after modification, the first column specified in this clause is the most significant key, and each successive specified column is the next most significant key.

If the ON clause is omitted when modifying to isam, hash, or btree, the table is keyed, by default, on the first column. When a table is modified to heap, the ON clause must be omitted.

When modifying a table to heapsort, specify the sort order as ASC (ascending) or DESC (descending). The default is ASC. When sorting, the DBMS Server considers nulls greater than any non-null value.

In general, any MODIFY.. TO <storage_structure> ... of a table or index assigned to a raw location must include WITH LOCATION=(...) syntax to move the table or index to another set of locations because modify semantics involve a create, rename, and delete file, which works efficiently for cooked locations, but does not adapt to raw locations.

If UNIQUE is used with a partitioned table, the new storage structure must be compatible with the table's partitioning scheme. This means that given a value for the unique storage structure key columns, it must be possible to determine that a row with that key will be in one particular physical partition. In practice, this rule means that the partitioning scheme columns must be the same as (or a subset of) the storage structure key columns. It also means that unique cannot be used with AUTOMATIC partitioning. A MODIFY TO UNIQUE that violates the partitioning rule will be rejected with an error.

Note: It is still possible to enforce an arbitrary uniqueness constraint on a partitioned table, regardless of the partitioning scheme, by adding a UNIQUE or PRIMARY KEY constraint, or a unique secondary index, to the table.

Modify...to Reconstruct

To rebuild the existing storage structure for a table or partition, use the MODIFY...TO RECONSTRUCT option.

The reconstruct action allows the table or partitions to be rebuilt, maintaining the existing storage structure, key columns, and storage attributes. Any overrides specified in the MODIFY statement WITH clause are applied.

The reconstruct variant provides a simple means for partitioning or unpartitioning a table without affecting its other attributes. Partitioning or unpartitioning a table requires rewriting its storage structure, which is why partitioning is limited to restructuring variants of the MODIFY statement.

The heapsort structure is not really a storage structure, in the sense that the sort criteria are not remembered in any system catalog. Therefore reconstruction of a table originally modified to heapsort simply remodifies the table to heap with no additional sorting.

When operating on specific logical partitions instead of an entire table, the reconstruct modify does not permit any override with-attributes except for the location option.

The partition name clause allows the modify statement to operate on specific named logical partitions. Partition names must be listed from outer dimension to inner dimension.

Note: MODIFY...TO RECONSTRUCT does not work on secondary indexes.

Modify...to Merge

To shrink a btree index, use the MODIFY...TO MERGE option. When data is added to a btree table, the index automatically expands. However, a btree index does not shrink when rows are deleted from the btree table.

MODIFY...TO MERGE affects only the index, and therefore usually runs a good deal faster than the other modify variants. MODIFY...TO MERGE does not require any temporary disk space to execute.

Modify...to Relocate

To move the data without changing the number of locations or storage structure, specify MODIFY...TO RELOCATE.

For example, to relocate the employee table to three different areas:

```
modify employee to relocate
        with    oldlocation = (area1, area2, area3),
                newlocation = (area4, area5, area6);
```

The data in area1is moved to area4, the data in area2 is moved to area5, and the data on area3 is moved to area6. The number of areas listed in the oldlocation and newlocation options must be equal. The data in each area listed in the oldlocation list is moved "as is" to the corresponding area in the newlocation list. The oldlocation and newlocation options can only be used when relocate is specified.

To change some but not all locations, specify only the locations to be changed. For example, you can move only the data in area1 of the employee table:

```
modify employee to relocate
    with    oldlocation = (area1),
            newlocation = (area4);
```

Areas 2 and 3 are not changed.

The DBMS Server is very efficient at spreading a table or index across multiple locations. For example, if a table is to be spread over three locations:

```
create table large (wide varchar(2000),
        with location = (area1, area2, area3);
```

rows are added to each location in turn, in 16 page (approximately 32K for the 2K page size) chunks. If it is not possible to allocate 16 full pages on an area when it is that area's turn to be filled, the table is out of space, even if there is plenty of room in the table's other areas.

Modify...to Reorganize

To move the data and change the number of locations without changing storage structure, specify MODIFY...TO REORGANIZE. For example, to spread an employee table over three locations:

```
modify employee to reorganize
        with location = (area1, area2, area3);
```

When specifying reorganize, the only valid WITH clause option is LOCATION.

Modify...to Truncated

To delete all the rows in the table and release the file space back to the operating system, specify MODIFY...TO TRUNCATED. For example, the following statement deletes all rows in the acct_payable table and releases the space:

```
modify acct_payable to truncated;
```

Using TRUNCATED converts the storage structure of the table to heap. The *with_clause* options cannot be specified when using MODIFY...TO TRUNCATED.

Modify...to Add_extend

To add pages to a table, specify MODIFY...TO ADD_EXTEND. To specify the number of pages to be added, use the extend=*number_of_pages* option. If the with extend=*number_of_pages* option is omitted, the DBMS Server adds the default number of pages specified for extending the table. To specify the default, use the MODIFY...WITH EXTEND statement. If no default has been specified for the table, 16 pages are added.

The MODIFY...TO ADD_EXTEND option does not rebuild the table or drop any secondary indexes.

Modify...with Blob_extend

To add pages to a blob extension table, specify MODIFY...WITH BLOB_EXTEND. To specify the number of pages to be added, use the extend=*number_of_pages* option. If the WITH *extend=number_of_pages* option is omitted, the DBMS Server adds the default number of pages specified for extending the table. To specify the default, use the MODIFY...WITH EXTEND statement. If no default has been specified for the table, 16 pages are added.

The MODIFY...WITH BLOB_EXTEND option does not rebuild the table or drop any secondary indexes.

Modify...to Phys_consistent|Phys_inconsistent

A physically inconsistent table has some form of physical corruption. A table is marked physically inconsistent if rollforwarddb of that table fails for some reason, or if the table is a secondary index and point-in-time recovery has occurred only on the base table.

The MODIFY...TO PHYS_CONSISTENT command marks the table as physically consistent but does not change the underlying problem.

Modify...to Log_consistent|Log_inconsistent

A logically inconsistent table is out-of-step in some way with one or more logically related tables. A table is marked logically inconsistent if the table is journaled and the user enters rollforwarddb +c -j on the table, or if the table is journaled and the user rolls forward to a point-in-time. For example, if two tables are logically related through referential constraints, and only one is moved to a specific point-in-time, the table is marked logically inconsistent.

The MODIFY TO LOG_CONSISTENT command marks the table as logically consistent but does not fix the underlying problem.

Modify...to Table_recovery_allowed|Table_recovery_disallowed

To prevent the possibility of logically or physically inconsistent tables due to table-level recovery, table-level recovery can be disallowed for any table with the MODIFY...TO TABLE_RECOVERY_DISALLOWED command.

Modify...to Unique_scope = Statement|Row

To defer the uniqueness check until the end of statement execution (for unique indexes or base table structures), specify the MODIFY...TO UNIQUE_SCOPE = STATEMENT|ROW option. This statement allows more than one row to contain the same value for a declared unique column during the processing of an update statement, as long as the column values are unique at the end of the statement. These semantics are required to support ANSI uniqueness constraints.

This MODIFY statement is necessary when using a pre-existing secondary index or the base table structure for a unique constraint. When an index is constructed specifically for a unique constraint definition (even when the definition includes a constraint WITH clause), the UNIQUE_SCOPE attribute is set automatically.

Modify…to [No]Readonly

This option marks the table READONLY and returns an error if insert, delete, or update operations are performed. Additionally, all transactions that use the read only table take a shared table lock.

Modify…to Priority=n

This MODIFY statement permits the assignment of a cache priority without having to also change the storage structure. This must be an integer between 0 and 8, with 0 being the lowest priority and 8 being the highest. A specification of 0 causes the table to revert to a normal cache management algorithm and is the default value. If an explicit priority is not set for an index belonging to a base table to which an explicit priority has been assigned, the index inherits the priority of the base table.

With Clause Options

The remaining WITH clause options for the MODIFY statement are described below.

Fillfactor, Minpages, and Maxpages

FILLFACTOR specifies the percentage (from 1 to 100) of each primary data page that must be filled with rows, under ideal conditions. For example, if you specify a fillfactor of 40, the DBMS Server fills 40% of each of the primary data pages in the restructured table with rows. You can specify this option with the isam, hash, or btree structures. Take care when specifying large fillfactors because a non-uniform distribution of key values can later result in overflow pages and thus degrade access performance for the table.

MINPAGES specifies the minimum number of primary pages a hash table must have. MAXPAGES specifies the maximum number of primary pages a hash table can have. Minpages and maxpages must be at least 1. If both minpages and maxpages are specified in a modify statement, minpages must not exceed maxpages.

For best performance, the values for minpages and maxpages must be a power of 2. If a number other than a power of 2 is chosen, the DBMS Server can change the number to the nearest power of 2 when the modify executes. To ensure that the specified number is not changed, set both minpages and maxpages to that number.

By default, modify to *storage-structure* resets these attributes back to their defaults (listed below). The modify to reconstruct operation does not affect these attributes.

Default values for fillfactor, minpages and maxpages are listed in this table:

	Fillfactor	Minpages	Maxpages
hash	50	16	no limit
compressed hash	75	1	no limit
Isam	80	n/a	n/a
compressed isam	100	n/a	n/a
btree	80	n/a	n/a
compressed btree	100	n/a	n/a

Leaffill and Nonleaffill

For btree tables, the LEAFFILL parameter specifies how full to fill the leaf index pages. Leaf index pages are the index pages that are directly above the data pages. NONLEAFFILL specifies how full to fill the non-leaf index pages; non-leaf index pages are the pages above the leaf pages. Specify leaffill and nonleaffill as percentages. For example, if you modify a table to btree, specifying NONLEAFFILL=75, each non-leaf index page is 75% full when the modification is complete.

The LEAFFILLand NONLEAFFILL parameters can assist with controlling locking contention in btree index pages. If some open space is retained on these pages, concurrent users can access the btree with less likelihood of contention while their queries descend the index tree. Strike a balance between preserving space in index pages and creating a greater number of index pages. More levels of index pages require more I/O to locate a data row.

By default, modify to *storage-structure* resets these attributes back to their defaults.

Default: LEAFFILL=70; NONLEAFFILL=80

The MODIFY TO RECONSTRUCT operation does not affect these attributes.

Allocation Option

Use the WITH ALLOCATION option to specify the number of pages initially allocated to the table or index. By pre-allocating disk space to a table, runtime errors that result from running out of disk space can be avoided. If the table is spread across multiple locations, space is allocated across all locations.

The number of pages specified must be between 4 and 8,388,607 (the maximum number of pages in a table). If the specified number of pages cannot be allocated, the modify statement is aborted.

A table can be modified to a smaller size. If the table requires more pages that you specify, the table is extended and no data is lost. A table can be modified to a larger size to reserve disk space for the table.

If not specified, a modify does not change a table's allocation.

Extend

To specify the number of pages by which a table or index grows when it requires more space, use the WITH EXTEND clause. The number of pages specified must be between 1 and 8,388,607 (the maximum number of pages in a table). If the specified number of pages cannot be allocated when the table must be extended (for example, during an insert operation), the DBMS Server aborts the statement and issues an error. By default, tables and indexes are extended by groups of 16 pages.

If not specified, a modify does not change a table's extend attribute.

Compression

To specify data and key compression, use the WITH COMPRESSION clause. Compression removes the string trim from variable character data. The following table lists valid compression options:

Storage Structure	Base Table or Secondary Index	Can Compress Data?	Can Compress Key?
hash	Base Table	Yes	No
	Secondary Index	Yes	No
heap	Base Table	Yes	No
	Secondary Index	No	No
btree	Base Table	Yes	Yes
btree	Secondary Index	No	Yes
Isam	Base Table	Yes	No
	Secondary Index	Yes	No

To specify an uncompressed storage structure, specify WITH NOCOMPRESSION.

To compress both key and data for tables where this is valid (primarily btree), specify WITH COMPRESSION, omitting the KEY and DATA clause. To compress data or keys independently of one another, specify WITH COMPRESSION = (KEY|DATA). To compress data using bit compression, specify WITH COMPRESSION = HIDATA. To explicitly suppress compression of data or keys, specify WITH COMPRESSION = (NOKEY | NODATA).

If not specified, modify to *storage-structure* removes compression, unless the c-prefix variants are used (cbtree and so on). Other variants of MODIFY preserve the table's compression type.

Location

To change the location of a table when modifying its storage structure, specify the LOCATION option. This option specifies one or more new locations for the table. The locations specified must exist when the statement executes and the database must have been extended to those locations. For information about areas and extending databases, see the *Database Administrator Guide*.

Unique_scope

The UNIQUE_SCOPE option specifies, for tables or indexes with unique storage structures, how uniqueness is checked during an update option.

There are two options:

UNIQUE_SCOPE = ROW

    Checks uniqueness as each row is inserted.

UNIQUE_SCOPE = STATEMENT

     Checks uniqueness after the UPDATE statement is completed.

Default: UNIQUE_SCOPE = ROW, when first imposing uniqueness on a table.

Specify the UNIQUE_SCOPE option only when modifying to a unique storage structure. For example:

```
modify mytable to btree unique with unique_scope = row;
```

If not otherwise specified, a MODIFY does not change the UNIQUE_SCOPE setting.

(No)persistence Option

The [NO]PERSISTENCE option specifies whether an index is recreated when its related base table is modified. This option is valid only for indexes.

There are two options:

WITH PERSISTENCE

Recreates the index when its base table is modified.

WITH NOPERSISTENCE

Drops the index when its base table is modified.

Default: A MODIFY to a storage structure sets an index to NOPERSISTENCE.

Other MODIFY actions (including MODIFY TO RECONSTRUCT) do not change an index's persistence.

Page_size

Specify page size using PAGE_SIZE = *n* where *n* can be the page size in the following table:

Page Size	Number of Bytes
2K	2,048
4K	4,096
8K	8,192
16K	16,384
32K	32,768
64K	65,536

The default page size is 2,048. The tid size is 4. The buffer cache for the installation must also be configured with the page size you specify in create table or an error occurs.

The page size of a session temporary table cannot be changed by a modify.

Nopartition | Partition=

The PARTITION= clause allows the partitioning scheme of a table to be changed. The table does not have to be partitioned initially. The NOPARTITION clause removes partitioning from a table. For the syntax of a PARTITION= specification, see Partitioning Schemes (see page 426).

The with_clause options PARTITION= or NOPARTITION are permitted in a MODIFY statement only if the MODIFY statement specifies a storage structure which includes the RECONSTRUCT action. Other forms of the MODIFY statement (for example, MODIFY TO TRUNCATED) do not allow either PARTITION= or NOPARTITION clauses.

The default for the MODIFY statement is to not change the partitioning scheme of a table.

Concurrent_updates

The CONCURRENT_UPDATES option specifies that a table modify is to be performed online. Unlike a regular modify, which locks out all other access to the table for the entire duration, an online modify permits normal read and update access to the table for most of the modify. There is a brief period at the end of the modify operation where exclusive access to the table is still required.

Online modification of tables cannot be accomplished in the following:

- Ingres clusters
- Temporary tables
- System catalogs
- Partitioned tables
- Secondary indices

Note: To use online modification, a database must be journaled.

Nodependency_check

A table cannot be modified if it destroys indexes needed for constraints. The operation can be forced, however, by using the NODEPENDENCY_CHECK option.

Important! If you use this option, you must preserve or recreate the table structure necessary to enforce the constraints.

Embedded Usage

When using the MODIFY statement in an application, the DBMS Server returns the number of rows modified in the SQLCA's sqlerrd(3) field. If the statement does not modify any rows, the DBMS Server sets the SQLCA's sqlcode to 100.

The preprocessor does not verify the syntax of the *with_clause*. The values in the *with_clause* options can be specified using host language variables. Any other parameters cannot be specified using host language variables.

Permissions

You must own the table or have the security privilege and connect as the owner.

Locking

The MODIFY statement requires an exclusive table lock. Other sessions, even those using readlock=nolock, cannot access the table until the transaction containing the MODIFY statement is committed.

Two exceptions are the MODIFY TO TABLE_DEBUG variant, which takes a shared table lock, and the CONCURRENT_UPDATES option, which takes only a brief exclusive lock at the end of the modify operation.

Related Statements

Copy (see page 331)

Create Index (see page 353)

Create Location (see page 363)

Create Table (see page 402)

Examples: Modify

The following are MODIFY statement examples:

1. Modify the employee table to an indexed sequential storage structure with eno as the keyed column.

   ```
   modify employee to isam on eno;
   ```

   If eno is the first column of the employee table, the same result can be achieved by:

   ```
   modify employee to isam;
   ```

2. Redo the isam structure on the employee, but request a 60% occupancy on all primary pages.

   ```
   modify employee to reconstruct
           with fillfactor = 60;
   ```

3. Modify the job table to compressed hash storage structure with jid and salary as keyed columns.

   ```
   modify job to hash on jid, salary
           with compression;
   ```

4. Perform the same modify, but also request 75% occupancy on all primary pages, a minimum of 7 primary pages and a maximum of 43 primary pages.

   ```
   modify job to hash on jid, salary
           with compression, fillfactor = 75,
           minpages = 7, maxpages = 43;
   ```

5. Perform the same modify again but only request a minimum of 16 primary pages.

   ```
   modify job to hash on jid, salary
           with compression, minpages = 16;
   ```

6. Modify the dept table to a heap storage structure and move it to a new location.

   ```
   modify dept to heap with location=(area4);
   ```

7. Modify the dept table to a heap again, but have rows sorted on the dno column and have any duplicate rows removed.

   ```
   modify dept to heapsort on dno;
   ```

8. Modify the employee table in ascending order by ename, descending order by age, and have any duplicate rows removed.

   ```
   modify employee to heapsort on ename asc,
           age desc;
   ```

9. Modify the employee table to btree on ename so that data pages are 50% full and index pages are initially 40% full.

```
modify employee to btree on ename
            with fillfactor = 50, leaffill = 40;
```

10. Modify a table to btree with data compression, no key compression. This is the format used by the (obsolete) cbtree storage structure.

```
modify table1 to btree
        with compression=(nokey, data);
```

11. Modify an index to btree using key compression.

```
modify index1 to btree with compression=(key);
```

12. Modify an index so it is retained when its base table is modified. Its current table structure is unchanged.

```
modify empidx to reconstruct with persistence;
```

13. Modify a table, specifying the number of pages to be initially allocated to it and the number of pages by which it is extended when it requires more space.

```
modify inventory to btree
        with allocation = 10000, extend = 1000;
```

14. Modify an index to have uniqueness checked after the completion of an UPDATE statement.

```
modify empidx to btree unique on empid
        with unique_scope = statement;
```

15. Move all physical partitions of the table in the Create Table Example 17 (see page 437) that contain 2001 and earlier ship-dates to the history_loc location.

```
MODIFY lineitems PARTITION p1 TO REORGANIZE
```

```
WITH LOCATION = (history_loc);
```

16. Remove partitioning from a table.

```
modify lineitems to reconstruct with nopartition;
```

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];
```

The FOR READONLY clause indicates that, though the cursor can have been declared for update, the cursor is being opened for reading only. This clause improves the performance of data retrieval, and can be used whenever appropriate.

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 605).

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' + table_name + 'where low < ?
    and high > ?';
exec sql prepare sel_stmt from :statement_buffer;
```

When opening the cursor for this prepared SELECT statement, values for the question marks must be provided 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 Server evaluates the SELECT statement accordingly. If an SQLDA is used, the values that replace the question marks are taken from variables to which the sqlvar elements point. Before using the descriptor in an OPEN CURSOR statement, allocate the SQLDA and the variables to which the sqlvar elements point, and place values in the variables. For more information about the SQLDA and its sqlvar elements, see the chapter "Working with Embedded SQL."

More than one cursor can be opened at the same time, but only one cursor that has been declared for update in deferred mode can be open at a time. If a cursor that has been declared for update in deferred mode is open, all other open cursors must have been declared for readonly or for update in direct mode.

The same cursor can be opened and closed (with the CLOSE statement) successive times in a single program. An open cursor must be closed before it can be reopened.

A string constant or a host language variable can be used to specify the *cursor_name*. The open statement must be terminated according to the rules of your host language.

Permissions

This statement is available to all users.

Locking

If for readonly is not specified, the DBMS Server can take exclusive locks while the cursor is open.

Related Statements

Close (see page 320)

Declare Cursor (see page 451)

Fetch (see page 523)

Update (see page 702)

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 SQL 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 SQL 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. The maximum length of a statement name is 32 characters. If the statement string is blank or empty, the DBMS Server returns a runtime syntax error.

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 not accurate 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 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 must be used on the statement string. If the prepared statement is to be executed repeatedly, the prepare and execute sequence must be used.

The following statements cannot be prepared and executed dynamically:

- call
- close
- connect
- create procedure
- declare
- disconnect
- enddata
- execute immediate
- execute procedure
- execute
- fetch
- get data
- get dbevent
- include
- inquire_sql
- open
- prepare to commit
- prepare
- put data
- set
- whenever

In addition, the following types of statements cannot be prepared and dynamically executed:

- Dynamic SQL statements
- SQL 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 specifying a string that includes quotes as a string constant, remember that quoted characters within the statement string must follow the SQL string delimiting rules.

If your host language delimits strings with double quotes, the quoted characters within the statement string must be delimited by the SQL 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 following 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 contains the data types, lengths, and names of the result columns. If the statement was not a SELECT, the descriptor's sqld field contains a zero. For more information about the results of describing a statement, see the chapter "Working with Embedded SQL" and Describe (see page 474).

This statement must be terminated according to the rules of your host language.

Usage in OpenAPI, ODBC, JDBC, .NET

The applications should use the interface-specific mechanism, rather than sending PREPARE in the SQL.

Permissions

This statement is available to all users.

Related Statements

Describe (see page 474)

Execute (see page 505)

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;
```

Prepare to Commit

Valid in: ESQL, OpenAPI, ODBC, JDBC, .NET

The PREPARE TO COMMIT statement polls the local DBMS server to determine the commit status of the local transaction associated with the specified distributed transaction. The distributed transaction is identified by its distributed transaction ID, a unique, 8-byte integer that is generated by the coordinator application.

This statement provides support for the two-phase commit functionality. For a discussion of two phase commit, see the chapter "Working with Transactions and Handling Errors."

Dynamic SQL cannot be used to execute this statement. This statement must be terminated according to the rules of your host language.

Note: The only SQL statements that can follow the PREPARE TO COMMIT statement are COMMIT or ROLLBACK.

Syntax

The PREPARE TO COMMIT statement has the following format:

```
EXEC SQL PREPARE TO COMMIT
            WITH HIGHDXID = value, LOWDXID = value;
```

value

Can be an integer constant or integer variable. The *value* associated with HIGHDXID must be the high-order 4 bytes of the distributed transaction ID. The *value* associated with LOWDXID must be the low-order 4 bytes of the distributed transaction ID.

Usage in OpenAPI, ODBC, JDBC, .NET

For OpenAPI, Prepare to Commit functionality is achieved through IIapi_prepareCommit(). For ODBC, JDBC, and .Net, it is supported through interface-specific XA.

Permissions

This statement is available to all users.

Related Statements

Commit (see page 323)

Rollback (see page 636)

Example: Prepare to Commit

The following example shows a portion of a banking application that uses the
PREPARE TO COMMIT statement:

```
...
exec sql begin declare section;
        from_account            integer;
        to_account              integer;
        amount                  integer;
        high                    integer;
        low                     integer;
        acc_number              integer;
        balance                 integer;
exec sql end declare section;

define          sf_branch 1
define          bk_branch 2
define          before_willing_commit 1
define          willing_commit 2

exec sql whenever sqlerror stop;

/* connect to the branch database in s.f */

exec sql connect annie session :sf_branch;

/* program assigns value to from_account,
** to_account, and amount
*/

/* begin a local transaction on s.f branch to
** update the balance in the from_account.
*/

exec sql update account
        set balance = balance - :amount
        where acc_number = :from_account;

/* connect to the branch database in berkeley. */

exec sql connect aaa session :bk_branch;

/* begin a local transaction on berkeley branch
** to update the balance in the to_account.
*/

exec sql update account
        set balance = balance + :amount
        where acc_number = :to_account;
```

```
/* ready to commit the fund transfer transaction.
** switch to s.f branch to issue the prepare to
** commit statement.*/

exec sql set_sql (session = :sf_branch);

/* store the transaction state information */

store_state_of_xact(sf_branch,
       before_willing_commit, high, low, "annie"

exec sql prepare to commit with highdxid = :high,
       lowdxid = :low;

/* store the transaction state information */
store_state_of_xact(sf_branch, willing_commit,
       high, low, "aaa");

/* switch to berkeley branch to issue the prepare
** to commit statement.*/

exec sql set_sql (session = :bk_branch);

/* store the transaction state information */

store_state_of_xact(bk_branch,
       before_willing_commit, high, low, "aaa");

exec sql prepare to commit with highdxid = :high,
       lowdxid = :low;

       /* store the transaction state information */

store_state_of_xact(bk_branch, willing_commit,
       high, low, "aaa");

/* both branches are ready to commit; commit the
** fund transfer transaction. */
/* switch to s.f branch to commit the
** local transaction. */

exec sql set_sql (session = :sf_branch);

exec sql commit;

/* switch to berkeley branch to commit the
** local transaction. */
```

```
exec sql set_sql (session = :bk_branch);

exec sql commit;

/* distributed transaction complete */
```

Put Data

Valid in: ESQL

The PUT DATA statement writes a segment of a long varchar, long nvarchar, or long byte column from an embedded program to a table. The PUT DATA statement is valid only in data handler routines. For details about data handler routines, see the chapter "Working with Embedded SQL" and the *Embedded SQL* Companion *Guide*.

Syntax

The PUT DATA statement has the following format:

```
EXEC SQL PUT DATA(SEGMENT = col_value
                [, SEGMENTLENGTH = length_value]
                 [, DATAEND = dataend_value]);
```

col value

Specifies the value to be assigned to the column. The maximum length of a long varchar, long nvarchar, or long byte column is 2 GB.

length_value

(Optional) Signed 4-byte integer specifying the length of the data segment being written.

dataend_value

(Optional) Signed 4-byte integer specifying whether the segment is the last segment to be written. To indicate end-of-data, specify 1. To indicate that the segment is not the last, specify 0.

The host language variables for *col_value*, *length_value*, and *dataend_value* must be declared in a BEGIN DECLARE SECTION/END DECLARE SECTION block to the ESQL preprocessor.

The data handler must issue a PUT DATA statement with DATAEND set to 1 before exiting; otherwise, the DBMS Server issues a runtime error.

Permissions

This statement is available to all users.

Related Statements

Get Data (see page 531)

Raise Dbevent

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The RAISE DBEVENT statement enables an application to notify other applications of its status.

Syntax

The RAISE DBEVENT statement has the following format:

```
[EXEC SQL] RAISE DBEVENT [schema.]event_name [event_text]
            [WITH [NO]SHARE];
```

event_name

Specifies an existing database event name.

Description

The RAISE DBEVENT statement enables a session to communicate status information to other sessions that are registered to receive *event_name*.

If *schema* is omitted, the DBMS Server checks first for the specified database event owned by the effective user of the session. If the current effective user does not own the database event, the DBMS Server seeks the specified database event in the database events owned by the DBA.

Use the optional *event_text* parameter to pass a (maximum 256 character) string to receiving applications; to obtain the text, receiving applications must use the INQUIRE_SQL(DBEVENTTEXT) statement.

To restrict database event notification to the session that raised the database event, specify WITH NOSHARE. To notify all registered sessions, specify WITH SHARE or omit this clause. The default is SHARE.

If a database event is raised from within a transaction and the transaction is subsequently rolled back, the database event notification is not rolled back.

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.

Permissions

To raise a database event you do not own, specify the *schema* parameter and have raise privilege for the database event. To assign the raise privilege to another user, use the GRANT statement.

Related Statements

Create Dbevent (see page 350)

Get Dbevent (see page 533)

Inquire_sql (see page 562)

Register Dbevent (see page 620)

Remove Dbevent (see page 625)

Raise Error

Valid in: DBProc

The Raise Error statement notifies the DBMS Server and the application that a database procedure has encountered an error. The raise error statement can only be issued inside a database procedure. This statement is particularly useful when using a rule and its associated database procedure to apply an integrity constraint.

When this statement is issued, the DBMS Server responds as if the database procedure has encountered an error. If the procedure was invoked by a rule, the DBMS Server rolls back any changes made to the database by the original user statement and any made by the database procedure and issues an error to the application. If the raise error statement is issued in a procedure that is executed directly by the execute procedure statement, the error is processed in the same manner as are other errors in a database procedure. (For details, refer Database Procedures.)

When executing a Raise Error statement with associated *errortext*, both the *errornumber* and *errortext* are returned to the application. However, only the *errortext* is displayed. In embedded SQL and 4GL applications, this can be changed by using inquire_sql to retrieve the error number and text (dbmserror and errortext). Additionally, in embedded SQL, use the whenever statement with the sqlerror condition to inhibit the automatic display of the *errortext* and provide an alternate error handling mechanism.

The *errornumber* is considered a local DBMS server error and, by default, is returned to SQLCA variable sqlerrd(1) and to dbmserror, which is accessible using inquire_sql. The generic error number corresponding to a raise error is 41300. This number is returned, by default, to errorno, which is accessible using inquire_sql, and to sqlcode, another SQLCA variable. The number in sqlcode is negative (-41300).

If you have specified that local errors are returned to errorno and sqlcode (by issuing the set_sql(dbmserror) statement), the locations described above for the *errornumber* and its generic error number are reversed also. In such cases, it is not necessary to provide a negative number for the *errornumber*; the DBMS Server automatically negates the number when it places the number in sqlcode. For a complete discussion of local and generic error numbers, see the chapter "Working with Transactions and Handling Errors."

In interactive applications that rely on default error messages, such as QBF, the *errornumber* must be included as part of the *errortext* to display the number. For example, assume that you are working in QBF and a rule fires and, as a result, the following statement executes:

```
raise error 123445 'Invalid value inserted';
```

When the statement is executed, QBF displays a pop-up window with the message:

```
'Invalid value inserted'
```

If it is important to display the error number also, it must be included as part of the *errortext* in addition to specifying it as the *errornumber:*

```
raise error 123445
        'Error 123445: Invalid value inserted';
```

To direct the output of the raise error statement to the error log, specify with destination = (error_log). The error number and text are written to the "errlog.log" file with message identifier E_QE0300. To direct output to the session (the default behavior), specify with destination = (session). To both log an error and return it to an application, specify with destination = (session, error_log).

To direct the output of the raise error statement directly to the audit log, specify with destination=(audit_log). Any such messages are regarded as security audit events. The description indicates the source of the event (for example: message, raise error). The message text and error number are available in the detail information for the event.

Syntax

The Raise Error statement has the following format:

```
RAISE ERROR errornumber [errortext]
             [WITH DESTINATION = ([SESSION] [, ERROR_LOG] [, AUDIT_LOG])];
```

errornumber

Can be an integer constant, a local variable, or a parameter in the invoked database procedure. If it is a local variable, it must be either a non-nullable integer or smallint type.

errortext

Is an optional text string that describes the error associated with *errornumber*. It can be a string constant, a local string variable, or a parameter in the invoked database procedure. If *errortext* is not specified, interactive applications such as QBF display a default error message.

Permissions

Anyone with permission to create procedures can use this statement.

Related Statements

Execute Procedure (see page 515)

Inquire_sql (see page 562)

Message (see page 577)

Example: Raise Error

The following example enforces a relationship (or integrity constraint) between an employee and a manager. When an employee is entered into the database, a check is performed to enforce the existence of the manager of the employee. If the manager is not found, the Raise Error statement returns a message to the user and rolls back the changes made to the database by the statement that fired the rule.

```
create procedure validate_manager
       (mname varchar(30)) as
declare
       msg varchar(80) not null;
       check_val integer;
begin
       select count(*) into :check_val from manager
               where name = :mname;
       if check_val = 0 then
               msg = 'Error 99999: Manager "' + :mname +
                       '" not found.';
raise error 99999 :msg;
endif;
end;

create rule check_emp after insert into employee
execute procedure validate_manager
       (mname = new.manager);
```

Register Dbevent

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The REGISTER DBEVENT statement enables a session to specify the database events it intends to receive.

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.

A session can register for events owned by the session's effective user or for which register privilege has been granted. If an attempt is made to register for a nonexistent event, for an event for which register privilege has not been granted, or twice for the same event, the DBMS Server issues an error.

If the *schema* parameter is omitted, the DBMS Server first checks the events owned by the current user. If the specified event is not found, the DBMS Server checks the events owned by the DBA.

If the REGISTER DBEVENT statement is issued from within a transaction that is subsequently rolled back, the registration remains in effect.

Syntax

The REGISTER DBEVENT statement has the following format:

```
[EXEC SQL] REGISTER DBEVENT [schema.]event_name;
```

Embedded Usage

In an embedded REGISTER DBEVENT statement, e*vent_name* cannot be specified using a host language variable, though *event_text* can be specified using a host string variable.

Permissions

To register for an event you do not own, you must specify the *schema* parameter, and must have register privilege for the database event. To assign the register privilege, use the GRANT statement.

Related Statements

Create Dbevent (see page 350)

Get Dbevent (see page 533)

Inquire_sql (see page 562)

Raise Dbevent (see page 615)

Remove Dbevent (see page 625)

Register Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REGISTER TABLE statement maps the structure of a file to the structure of a table.

Syntax

The REGISTER TABLE statement has the following format:

```
[EXEC SQL] REGISTER TABLE [schema.]table_name
          (column_name  col_type  [IS 'external_name']
          {, column_name  col_type  [IS 'external_name']})
          AS IMPORT FROM 'security_log_file_name'
          WITH    DBMX=SXA
                        [, ROWS = integer_value];
```

Description

The REGISTER TABLE statement maps the fields in a file to columns in a virtual table. After registering the file as a table, use SQL to manipulate the contents of the file. The registered table can be referred to in database procedures. To delete a registration, use the REMOVE TABLE statement.

Note: For information on registering IMA tables, see the *System Administrator Guide*.

Note: This statement is not the same as the REGISTER...AS LINK statement, which is described in the *Ingres Star User Guide*.

The following statements can be performed against registered tables:

- CREATE VIEW
- CREATE SYNONYM
- CREATE RULE
- COMMENT
- SELECT
- INSERT, UPDATE, AND DELETE (if they are from an updateable Enterprise Access product)
- DROP
- SAVE
- REGISTER...AS LINK (as described in the *Ingres Star User Guide*)

The following statements cannot be performed against registered tables:

- MODIFY
- CREATE INDEX

Security Log Files

The security log is created and written when security logging is enabled (using the ENABLE SECURITY_AUDIT statement). The security log file has the following format:

Field Name	Data Type	Description
audittime	Date	Date and time of the audit event
User_name	char(32)	Effective user name
Real_name	char(32)	Real user name

Field Name	Data Type	Description
userprivileges	char(32)	User privileges
objprivileges	char(32)	Object privileges
database	char(32)	Database
auditstatus	char(1)	Status of event; Y for success or N for failure
auditevent	char(24)	Type of event
objecttype	char(24)	Type of object
objectname	char(32)	Name of object
description	char(80)	Text description of event
objectowner	char(32)	Owner of the object being audited
detailnum	Integer(4)	Detail number
detailinfo	varchar(256)	Detail textual information
sessionid	char(16)	Session identifier
querytext_ sequence	Integer(4)	Sequence number for query text records, where applicable

Note: When registered, a security log is read-only.

To map the columns in the virtual table to the fields in the log file, specify the IS clause in the column list and the field name from the preceding table. For example:

```
db_name char(32) is 'database'
```

maps the table column, db_name, to the security log field, database.

At least one column must be specified. If the IS clause is omitted, the column names must correspond to the field names listed in the preceding table. Columns can be specified in any order.

The security log file name must be specified as a quoted string, and must be a valid operating system file specification. To dynamically register whatever log file is in use, specify AS IMPORT FROM CURRENT. If CURRENT is specified, SQL operations on the virtual log table always see the log file in use, even if the physical log file changes.

By default, the security log shows security events for the entire Ingres installation. If the database field is omitted, the security log contains records only for the database in which the log is registered.

With Clause Options

The ROWS option in the WITH clause specifies the number of records the log is expected to contain; the default is 1000. This value is displayed by the HELP TABLE statement as Rows:, and is used by the DBMS query optimizer to produce query plans for queries that see the registered table.

The DBMS option specifies the origin of the table being registered. To register a security log, specify SXA.

Embedded Usage

The WITH clause in an embedded REGISTER TABLE statement can be specified using a host string variable (with :*hostvar*).

Permissions

The session must have the maintain_audit privilege. To query the audit log, the auditor privilege is required.

Locking

The REGISTER TABLE statement locks pages in the iiregistrations, iirelation, iiattributes, and iiaudittables catalogs.

Related Statements

Remove Table (see page 626)

Example: Register Table

The following example registers a security audit log with various attributes:

```
REGISTER TABLE aud1 (
        audittime          date           not null,
        user_name          char(32)       not null,
        real_name          char(32)       not null,
        userprivileges     char(32)       not null,
        objprivileges      char(32)       not null,
        database           char(32)       not null,
        auditstatus        char(1)        not null,
        auditevent         char(24)       not null,
        objecttype         char(24)       not null,
        objectname         char(32)       not null,
        objectowner        char(32)       not null,
        description        char(80)       not null,
        objectlabel        security_label not null,
        detailinfo         varchar(256)   not null,
        detailnum          integer4       not null,
        sessionid          char(16)       not null,
        querytext_sequence integer4       not null
) AS IMPORT FROM 'myfile'
 WITH DBMS=sxa; rows=2000
```

Remove Dbevent

Valid in: SQL, ESQL, DBProc, 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;
```

Remove Dbevent 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 is received when the application issues the GET DBEVENT statement.

If the REMOVE DBEVENT statement is issued from within a transaction that is subsequently rolled back, the REMOVE DBEVENT statement is not rolled back. If an application issues the REMOVE DBEVENT statement for a database event for which it has not registered, the DBMS Server returns an error.

Permissions

This statement is available to all users.

Related Statements

Register Dbevent (see page 620)

Create Dbevent (see page 350)

Get Dbevent (see page 533)

Inquire_sql (see page 562)

Raise Dbevent (see page 615)

Remove Table

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REMOVE TABLE statement removes the registration of a security log file.

Syntax

The REMOVE TABLE statement has the following format:

```
[EXEC SQL] REMOVE TABLE [schema.]table_name
               {, [schema.]table_name}C2 SECURITY;
```

Description

The REMOVE TABLE statement removes the mapping of a file to a virtual table. To map files to virtual tables, use the REGISTER TABLE statement. The REMOVE TABLE statement removes security log files that were registered using the REGISTER TABLE...AS IMPORT statement.

Note: This statement is not the same as the REMOVE statement, which is described in the *Ingres Star User Guide*.

Embedded Usage

No portion of an embedded REMOVE TABLE statement can be specified using host language variables.

Permissions

You must have the security privilege. However, if the target table being removed is a security audit gateway table (that is, was registered with DBMS=SXA), you must have the maintain_audit privilege.

Locking

The REMOVE TABLE statement locks the iirelation, iiattribute, iiqrytext, and iiregistrations catalogs.

Related Statements

Register Table (see page 621)

Example: Remove Table

The following example removes a security log registration:

```
remove table logfile_xyz;
```

Return

Valid in: DBProc

The RETURN statement terminates a currently executing database procedure and gives control back to the calling application, and, optionally, returns a value. The RETURN statement can only be used inside a database procedure. The statement terminates the procedure and returns control to the application. (The calling application resumes execution at the statement following execute procedure.)

The optional *return_status* returns a value to the calling application when the RETURN statement executes. *Return_status* must be a non-null integer constant, variable, or parameter whose data type is comparable with the data type of the variable to which its value is assigned. If the *return_status* is not specified or if a return statement is not executed, the procedure returns 0 to the calling application.

The INTO clause of the EXECUTE PROCEDURE statement allows the calling application to retrieve the *return_status* once the procedure has finished executing.

Syntax

The RETURN statement has the following format:
```
RETURN [return_status];
```

Permissions

This statement is available to all users.

Example: Return

The following database procedure example, emp_sales_rank, returns rows containing the employee ID, total sales, and rank of sales amongst current salesmen:

```
create procedure emp_sales_rank
    result row (int, int, money) as
declare
    sales_tot  money;
    empid        int;
    sales_rank int;
begin
    sales_rank = 0;
    for select e.empid, sum(s.sales) as sales_sum into :empid, :sales_tot
        from employee e, sales s
          where e.job = 'sales' and e.empid = s.empid
          group by e.empid order by sales_sum do
      sales_rank = sales_rank + 1;
      return row(:sales_rank, :empid, :tot_sales);
    endfor;
end"
```

Return Row

Valid in: DBProc

The RETURN ROW statement composes a row using the values computed by the result expressions and returns it to the caller of the procedure in which it is contained. It can only be used within a database procedure. A RETURN ROW statement can be executed more than once in a single procedure invocation (for example, from within a FOR or WHILE loop) and offers a mechanism for passing multiple row images back to the caller.

Procedures containing RETURN ROW statements must also contain a RESULT ROW clause and the number of expressions in each RETURN ROW statement must be equal to the number of entries in the RESULT ROW clause. The data type of the result expressions must also be compatible with the corresponding entries in the RESULT clause.

The RETURN ROW statement can only be used in a procedure called directly from a host language program. It cannot be used in a procedure that is called from another database procedure.

Syntax

The RETURN ROW statement has the following format:

```
RETURN ROW (result_expression {,result_ expression});
```

Permissions

Anyone with permission to create procedures can use this statement.

Related Statements

Create Procedure (see page 366)

For-EndFor (see page 528)

Example: Return Row

The following is a RETURN ROW example:

```
Create procedure rowproc … as
    … result row (char(8), int, float) …
  begin
    …
    for select department, count(*), avg(salary) into :a, :b, :c from personnel
        group by deptname do
    …
   return row (:a, :b, :c);
    endfor;
     …
  end;
```

Revoke

Valid in: SQL, ESQL, OpenAPI, ODBC, JDBC, .NET

The REVOKE statement revokes privileges. It removes database privileges or role access granted to the specified users, groups, roles, or public. (To confer privileges, use the GRANT statement.) You cannot revoke privileges granted by other users.

Syntax

The REVOKE statement has the following format:

```
[EXEC SQL] REVOKE [GRANT OPTION FOR]
          ALL [PRIVILEGES] | privilege {, privilege} | role {, role}
          [ON [objecttype] [schema.]objectname {, [schema.]objectname} |
                     CURRENT INSTALLATION]
          FROM PUBLIC | [auth_type] auth_id {, auth_id}
          [CASCASE | RESTRICT];
```

privilege

Specifies the privileges to revoke. To revoke all privileges, use ALL. The privileges must agree with the *objecttype* as follows:

Object Type	Valid Privileges
TABLE (omit *objecttype*)	copy_into
	copy_from
	delete
	excluding
	insert
	references
	select
	update
DATABASE (or CURRENT INSTALLATION)	[no]access
	[no]connect_time_limit
	[no]create_procedure
	[no]create_table
	[no]db_admin
	[no]idle_time_limit
	[no]lockmode
	[no]query_io_limit
	[no]query_row_limit
	[no]select_syscat
	[no]session_priority
	[no[table_statistics
	[no]update_syscat
PROCEDURE	execute

Object Type	Valid Privileges
EVENT	register
	raise
ROLE	Omit this clause

objecttype

> Specifies the type of object on which the privileges were granted. To revoke permission on a table, omit the *objecttype* parameter. Valid *objecttype*s are:
>
> - dbevent
> - procedure

objectname

> Specifies the name of the table, database procedure, database event, or role on which the privileges were granted.

auth_type

> Specifies the type of authorization identifier to which privileges were granted. *Auth_type* must be user, group, or role. The default is user. More than one *auth_type* cannot be specified.

auth_id

> Specifies the users, groups, or roles from which privileges are being revoked. The *auth_ids* must agree with the type specified by the *auth_type*.
>
> For example, if you specify group as *auth_type*, the *auth_id* list must be a list of group identifiers. If you specify PUBLIC for the *auth_id*, you must omit *auth_type*. You can revoke from users and PUBLIC in the same REVOKE statement.

Revoking a database privilege makes that privilege on the specified database undefined for the specified grantee (*auth_id*). If an attempt is made to revoke a privilege that was not granted to a specified *auth_id*, no changes are made to the privileges of that *auth_id*.

Privileges granted on specific databases are not affected by REVOKE...ON CURRENT INSTALLATION, and privileges granted on current installation are not affected by REVOKE...ON DATABASE. Revoking privileges from PUBLIC does not affect privileges granted to a specific user.

If a privilege was granted using its "no" form (for example, nocreate_table or noquery_io_limit), the same form must be used when revoking the privilege. For example, the following grant prevents a user from creating tables:

```
grant nocreate_table on database employee
      to user karenk;
```

To remove this restriction:

```
revoke nocreate_table on database employee
      from user karenk;
```

For more information about privileges, see Grant (privilege) (see page 534). For a description of group and role identifiers, and details about privilege dependencies, see the *Database Administrator Guide*.

Note: In some cases granting a database privilege imposes a restriction, and revoking the privilege removes the restriction. For example, GRANT NOCREATETABLE prevents the user from creating tables.

Revoking Grant Option

The GRANT statement GRANT OPTION enables users other than the owner of an object to grant privileges on that object. For example, the following statement enables mike to grant the select privilege (with or without grant option) to other users:

```
grant select on employee_roster to mike with grant option;
```

The GRANT OPTION can be revoked without revoking the privilege with which it was granted. For example, the following statement:

```
revoke grant option for select on employees from mike...
```

means that mike can still select data from the employees table, but cannot grant the select privilege to other users. (The grant option cannot be specified for database privileges.)

Restrict versus Cascade

The RESTRICT and CASCADE options specify how the DBMS Server handles dependent privileges. The CASCADE option directs the DBMS Server to revoke the specified privileges plus all privileges and objects that depend on the privileges being revoked. The RESTRICT option directs the DBMS Server not to revoke the specified privilege if there are any dependent privileges or objects.

The owner of an object can grant privileges on that object to any user, group, or role. 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.

If sylvester creates a view on the employees table, that view depends on the select privilege that tom granted to sylvester. For example:

```
create view njemps as select * from employees where state='New Jersey'
```

To remove his grant to tom, all grants tom can have issued, and any dependent objects, 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. The njemps view is destroyed.

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 fails, and no privileges are revoked. The njemps view is not destroyed.

Note: If privileges are revoked from specific authorization IDs (users, groups, and roles) that were also granted to public, privileges and objects that depend on the grants persist (until the privileges are revoked from PUBLIC).

The RESTRICT and CASCADE parameters have the same effect whether 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 revoking a grant option with cascade, all dependent privileges are revoked, not just the grant option portion of the dependent privileges.

RESTRICT or CASCADE must be specified when revoking privileges on tables, database procedures, or database events. When revoking database privileges, CASCADE, RESTRICT or GRANT OPTION cannot be specified (because database privileges cannot be granted with GRANT OPTION).

Embedded Usage

You cannot use host language variables in an embedded REVOKE statement.

Permissions

The REVOKE statement can be executed by a user who is either the owner of the target object or has been granted permission (using WITH GRANT OPTION) to use the statement on the specific target object by another user. To revoke database privileges, you must be working in a session that is connected to the iidbdb. If the indicated roles have security audit attributes, the session must also have maintain_audit privilege.

Locking

The REVOKE statement locks pages in the iidbpriv catalog (if revoking database privileges) or iiprotect catalog, plus pages in the system catalogs that correspond to the object type (table, view, database event, or database procedure).

Related Statements

Create Group (see page 352)

Create Role (see page 380)

Create User (see page 440)

Grant (privilege) (see page 534)

Examples: Revoke

The following are REVOKE statement examples:

1. Revoke the query_row_limit privilege defined for the role identifier,
   review_emp, on the employee database.

   ```
   revoke query_row_limit on database employee
           from role review_emp;
   ```

2. Prevent any user from granting any form of access to the payroll table
   (assuming no privileges were granted to specific users, groups, or roles).
   Delete all dependent grants.

   ```
   revoke grant option for all on payroll
           from public cascade;
   ```

3. Prevent user joeb from running the manager bonus database procedure.
   Fail if joeb has granted execute privilege to other users.

   ```
   revoke execute on procedure mgrbonus
           from joeb restrict;
   ```

4. Prevent user harry from selecting rows from the employees table
   (assuming the same privilege was not granted to public).

   ```
   revoke select on employees
           from harry cascade;
   ```

5. Prevent user roger from using role manager.

   ```
   revoke manager from roger
   ```

Rollback

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The ROLLBACK statement rolls back the current transaction.

Note: This statement has additional considerations when used in a distributed
environment. For more information, see the *Ingres Star User Guide*.

Syntax

The ROLLBACK statement has the following format:

```
[EXEC SQL] ROLLBACK [WORK] [TO savepoint_name];
```

TO savepoint_name

Rolls back only those changes made after the specified savepoint. The transaction is not terminated. Processing resumes with the statement following the ROLLBACK TO *savepointname* statement. If autocommit is enabled, the ROLLBACK statement has no effect.

If ROLLBACK is issued without the optional TO clause, the statement terminates the transaction and rolls back any changes made by the transaction.

WORK

Is an optional keyword included for compatibility with the ISO and ANSI SQL standards.

Description

The ROLLBACK statement aborts part or all of the current transaction.

Only the ROLLBACK statement without the optional TO clause can be used in database procedures, and only in procedures that are directly executed. A database procedure that is invoked by a rule cannot contain either version of the ROLLBACK statement.

If a database event registration is removed (using the REMOVE DBEVENT statement), and the transaction is subsequently rolled backed, the database event registration is not restored.

Embedded Usage

In addition to aborting all or part of the current transaction, an embedded ROLLBACK statement:

- Closes all open cursors
- Discards all statements that were prepared in the current transaction

The TO *savepoint_name* clause cannot be included if there are open cursors in the transaction. Also, when a savepoint is specified in the ROLLBACK statement, the DBMS Server discards only those statements that were prepared after the declaration of the specified savepoint.

Savepoint_name cannot be specified using a host language variable.

Usage in OpenAPI, ODBC, JDBC, .NET

In OpenAPI, ODBC, JDBC, and .NET, the rollback function is supported through the interface-specific mechanism, for example, IIapi_rollback() for OpenAPI.

Permissions

This statement is available to all users.

Locking

If the ROLLBACK statement is issued without the TO *savepoint* option, the statement terminates the transaction and releases all locks held during the transaction. If the TO *savepoint_name* option is included, no locks are released.

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.

Related Statements

Commit (see page 323)

Syntax (see page 428)

Save

Valid in: SQL, ESQL

The SAVE statement directs the DBMS Server to save the specified table until the given expiration date. By default, base tables have no expiration date. An expiration date cannot be assigned to a system table.

Syntax

The SAVE statement has the following format:

```
[EXEC SQL] SAVE [schema.]table_name [UNTIL month day year];
```

month

Must be specified as an integer from 1 through 12, or the name of the month, abbreviated or spelled out.

day

Must be a valid day of the month (1 to 31), and *year* must be a fully specified year, for example, 2001. The range of valid dates is January 1, 1970 through December 31, 2035, inclusive.

Note: If the until clause is omitted, the expiration date is set to no expiration date. To purge expired tables from the database, use the verifydb command. Expired tables are not automatically purged.

Embedded Usage

Syntax elements cannot be represented with host language variables in an embedded SAVE statement.

Permissions

You must own the table.

Locking

The SAVE statement takes an exclusive lock on the specified table.

Example: Save

The following example saves the employee table until the end of February 2001:

```
save employee until feb 27 2001;
```

Savepoint

Valid in: SQL, ESQL, OpenAPI

The SAVEPOINT statement declares a named savepoint marker within a transaction. Savepoints can be used in conjunction with the ROLLBACK statement to roll back a transaction to the specified savepoint when necessary. Using savepoints can eliminate the need to roll back an entire transaction if it is not necessary.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

Syntax

The SAVEPOINT statement has the following format:

```
[EXEC SQL] SAVEPOINT savepoint_name;
```

savepoint_name

Can be any unquoted character string conforming to rules for object names, except that the first character need not be alphabetic. This enables numeric savepoint names to be specified.

Any number of savepoints can be declared within a transaction, and the same *savepoint_name* can be used more than once. However, if the transaction is aborted to a savepoint whose name is used more than once, the transaction is backed out to the most recent use of the *savepoint_name.*

All savepoints of a transaction are rendered inactive when the transaction is terminated (with either a COMMIT, a ROLLBACK, or a system intervention upon deadlock). For more information on deadlock, see Commit (see page 323) and Rollback (see page 636) and the chapter "Working with Transactions and Handling Errors."

Embedded Usage

An embedded SAVEPOINT statement cannot be issued when a cursor is open. *Savepoint_name* cannot be specified with a host language variable.

OpenAPI Usage

In OpenAPI, SAVEPOINT functionality is achieved through IIapi_savePoint().

Permissions

This statement is available to all users.

Related Statements

Commit (see page 323)

Rollback (see page 636)

Example: Savepoint

The following example declares savepoints among other SQL statements:

```
exec sql insert into emp (name, sal, bdate)
        values ('Jones,Bill', 10000, 1945);
/*set first savepoint marker */
exec sql savepoint setone;
exec sql insert into emp (name, sal, bdate)
        values ('Smith,Stan', 20000, 1948);
/* set second savepoint marker */
exec sql savepoint 2;
exec sql insert into emp (name, sal, bdate)
        values ('Engel,Harry', 18000, 1954);
/* undo third append;first and second remain */
exec sql rollback to 2;
/* undoes second append; first remains */
exec sql rollback to setone;
exec sql commit;
/* only the first append is committed */
```

Select (interactive)

Valid in: SQL, DBProc, 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]  * | expression [AS result_column]
            {, expression [[AS] result_column]}
            [FROM from_source {, from_source}
            [WHERE search_condition] WHERE (clause)
            [GROUP BY expression{, expression}] GROUP BY (clause)
            [HAVING search_condition] HAVING (clause)
            {UNION [ALL]
            (select)
            [ORDER BY ordering-expression [ASC | DESC]
                        {, ordering-expression [ASC | DESC]}];
```

Description

The SELECT (interactive) statement returns values from one or more specified tables or views, in the form of a single result table. Using the various clauses of the select statement, the following can be specified:

- Criteria for the values to be returned in the result table

- How the values in the result table are to be sorted and grouped

This statement description presents details of the select statement in interactive SQL (ISQL). In ISQL the results of a query are displayed on your terminal. In embedded SQL (ESQL), results are returned in host language variables. For details about using the select statement in ESQL, see Select (embedded) (see page 662).

Tip: User consumption of computing resources can be restricted during queries (selects) using the grant statement. Specifically, limits can be specified for I/O and for the number of rows returned. If the DBMS query optimizer estimates that a select exceeds the specified limits, the query is not executed. For details, see Grant (privilege) (see page 534).

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:
```

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
- 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 *table.column_name* syntax. For example:

```
select managers.name, employees.name
        from managers, employees...
```

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, qualify the column name with the table to which it belongs, or with a correlation name. For details, see From (see page 647).

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 can only be used in the first of a series of UNIONed selects. However, it can be used in the CREATE TABLE...AS SELECT and INSERT INTO...SELECT statements.

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

partno	customerno	qty	unit_price
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'),
       ifnull(edept,'Unassigned')
       from employees;
```

The preceding query selects all rows from the employees table; the result table is composed of the string constant 'Name:', the name of the employee, today's date (specified using 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, 1998	Shipping
Name:	Dave Murtagh	Aug 8, 1998	Purchasing
Name:	Benny Barth	Aug 8, 1998	Unassigned
Name:	Dean Reilly	Aug 8, 1998	Lumber
Name:	Al Obidinski	Aug 8, 1998	Unassigned

The SELECT clause can be used to obtain values calculated from the contents of a table. For example, the following query calculates the weekly salary of each employee based on their annual salary:

```
select ename, annual_salary/52 from employees;
```

Aggregate functions can be used to calculate values based on the contents of column. For example, the following query returns the highest, lowest, and average salary from the employees table:

```
select max(salary), min(salary), avg(salary)
        from employees;
```

These values are based on the amounts stored in the salary column.

To specify a name for a column in the result table, use the AS *result_column* clause. In the following example, the name, weekly_salary, is assigned to the second result column:

```
select ename, annual_salary/52 as weekly_salary
        from employees;
```

If a result column name is omitted 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 *from_source* parameter can be:

- One or more tables or views, specified using the following syntax:

  `[schema.]table [[AS] corr_name]`

  where *table* is the name of a table or view, or a synonym for a table or view.

- A join between two or more tables or views, specified using the following syntax:

  `source join_type JOIN source ON search_condition`

  or

  `source join_type JOIN source USING (column {, column})`

  or

  `source CROSS JOIN source`

  For details about specifying join sources, see ANSI/ISO Join Syntax (see page 658).

- A derived table (see page 158) specified using the following syntax:

  `(select_stmt) corr_name [(column_list)]`

  where *select_stmt* is a select statement with no ORDER BY clause, *corr_name* is a mandatory correlation name, and *column_list* is an optional list of override names for the columns in the SELECT list of the *select_list*.

A maximum of 126 tables can be specified in a query, including the tables in the from list, tables in subselects, and tables and views resulting from the expansion of the definitions of any views included in the query.

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.

Group By Clause

The GROUP BY clause combines the results for identical values in a column or expression. This clause is typically used in conjunction with aggregate functions to generate a single figure for each unique value in a column or expression. 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 can be many orders for the same part number.

Nulls are used to represent unknown data, and two nulls are typically not considered to be equal in SQL comparisons. However, the group by clause treats nulls as equal and returns a single row for nulls in a grouped column or expression.

Grouping can be performed on multiple columns or expressions. 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 you specify the group by clause, all columns in the select clause must be aggregate functions, columns specified in the group by clause or expressions, all of whose column references also appear in the columns or expressions of the group by clause.

Note: Aggregate functions cannot appear anywhere in a group by clause. Derived columns may appear in a group by clause, but must be referenced by their ordinal number in the column list.

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 orders for each part for each day in the past week:

```
select odate, partno, count(*) from orders
group by odate, partno
having odate >= (date('today') - '1 week');
```

Any columns or expressions contained in the HAVING clause must follow the same limitations previously described for the SELECT clause.

Order By Clause

The ORDER BY clause allows you to specify the columns on which the results table is to be sorted. 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
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 the following 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
Loram	Editorial	Costello
Murtagh	Shipping	Myron
Obidinski	Lumber	Myron
Reilly	Finance	Costello
Smith	Shipping	Myron

To display result columns sorted in descending order (reverse numeric or alphabetic order), specify order by *column_name* 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 end of the results table.

Note: If the order by clause is omitted, the order of the rows in the results table is not guaranteed to have any relationship to the storage structure or key structure of the source tables.

In union selects, the result column names must either be the column names from the select clause of the first select statement, or the number of the result column. For example:

```
select dcolumn from dtest
union
select zcolumn from ztest
order by dcolumn
```

In addition to specifying individual column names as the ordering-expressions of the order by clause, the results table can also be sorted on the value of some expression.

For example, the query:

```
select ename, edept, emanager from employees
      order by emanager+edpt
```

produces the employee list ordered on the concatenation of the emanager and edept values.

ename	edept	emanager
Murtagh	Shipping	Myron
Obidinski	Lumber	Myron
Reilly	Finance	Costello
Barth	Lumber	Myron
Karol	Editorial	Costello
Smith	Shipping	Myron
Loram	Editorial	Costello
Delore	Finance	Costello
Kugel	food prep	Snowden

The only requirement when specifying column names or expressions in the order by clause is that all referenced columns must exist in one of the tables contained in the from clause.

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.

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.

Syntax for 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. 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 database 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 queries that join columns in the same table.

If a correlation name is assigned to a table, the table must be referred to by 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...
```

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.

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 subselects, see the chapter "Understanding the Elements of SQL Statements."

ANSI/ISO Join Syntax

In addition to performing joins using the approach described in the Joins section, new syntax introduced with the 1992 ANSI/ISO SQL standard can be used. The new syntax provides a more precise way of specifying joins that are otherwise identical to those produced from the old syntax. The new syntax also allows the specification of outer joins.

An outer join can be described as the union of two SELECT statements, where the first SELECT returns the rows of the join sources that join together according to a specified search condition; the second SELECT returns the rows from one or both sources that do not have a matching row in the other source. For rows that do not have a matching row from the other source, null values are returned in all columns of the other source.

The new syntax is specified in the FROM clause, as follows:

source join_type JOIN *source* ON *search_condition*

or

source join_type JOIN *source* USING (*column* {,*column*})

or

source CROSS JOIN *source*

where:

source

Specifies the table, view, or join where the data for the left or right side of the join originates.

join_type

Specifies the type of join as one of the following:

INNER

Specifies an inner join.

LEFT

Specifies a left outer join, which returns all values from the left source.

RIGHT

Specifies a right outer join, which returns all values from the right source.

FULL

Specifies a full outer join, which returns all values from both left and right sources.

> **Note:** RIGHT and LEFT joins are the mirror image of each other: (table1 RIGHT JOIN table2) returns the same results as (table2 LEFT JOIN table1).

> Default: INNER

ON *search_condition*

Is a valid restriction, subject to the rules for the WHERE clause. The *search_condition* must not include aggregate functions or subselects. Matching pairs of rows in the join result are those that satisfy the *search_condition*.

USING (column {,column})

Is an alternate form of the *search_condition*. Each column in the USING clause must exist unambiguously in each join source. An ON *search_condition* is effectively generated in which the *search_condition* compares the columns of the USING clause from each join source.

CROSS JOIN

Is a cross product join of all rows of the join sources.

By default, joins are evaluated left to right. To override the default order of evaluation, use parentheses. A join source can itself be a 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 are assigned null values; in the second example, the restriction determines which rows are omitted from the result table.

The following examples are identical and use an outer join in the FROM clause to display all employees along with the name of their department, if any. One uses the ON clause and the other uses an equivalent USING clause:

```
select e.ename, d.dname from
(employees e left join departments d
on e.dept = d.dept);

select e.ename, d.dname from
(employees e left join departments d
using (dept));
```

Permissions

You can select from tables in schemas owned by the effective user, group, and role of the session. To select rows from tables in schemas owned by other users, groups, and roles:

- The *schema* parameter must be specified.

- The effective user, group, or role of the session must have select permission.

Examples: Select (interactive)

The following are SELECT (interactive) statement examples:

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
```

5. Select the current user name.

```
select dbmsinfo('username');
```

6. Display the day of the week that is three days from today.

```
select dow(date('today') + date('3 days'));
```

7. Display employees whose salary is higher than the average for their department (using derived tables):

```
select e.ename from employee e,
    (select avg(e1.salary), e1.dno from employee e1 group by e1.dno)
e2(avgsal, dno)
    where e.dno = e2.dno and e.salary > e2.salary;
```

This query can alternatively be coded as:

```
select e.ename from employee e,
    (select avg(e1.salary) as avgsal, e1.dno from employee e1 group by e1.dno)
e2
    where e.dno = e2.dno and e.salary > e2.salary;
```

Select (embedded)

Valid in: ESQL

The SELECT statement returns values from tables to host language variables in an embedded SQL program. For details about the various clauses of the SELECT statement, see Select (interactive) (see page 641). The following sections discuss details of interest to the embedded SQL programmer.

Syntax

The SELECT (embedded) statement has the following format:

Non-cursor version:

```
EXEC SQL [REPEATED] SELECT [FIRST rowCount] [ALL1 | DISTINCT]
            SELECT [FIRST rowCount] [ALL | DISTINCT]
            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 ordering-expression [ASC | DESC]
                         {, ordering-expression [ASC | DESC]}]
[EXEC SQL BEGIN;
            program code;
EXEC SQL END;]
```

To retrieve long varchar or long byte columns, specify a data handler routine in the INTO clause. For details, see Retrieving Values into Host Language Variables (see page 665).

Cursor version (embedded in a DECLARE CURSOR statement):

```
SELECT [ALL|DISTINCT]
      SELECT [FIRST rowCount] [ALL | DISTINCT]
      [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]}]
      [FOR [DEFERRED | DIRECT] UPDATE OF column {, column}];
```

where *result_expression* is:

expression | result_name = expression | expression as result_name

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 sqlselect 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. If other tables must be queried 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.

During the execution of the select loop, no other statements that access the database can be issued because this causes a runtime error. For information about manipulating and updating rows and tables within the database while data is being retrieved, see the chapter "Working with Embedded SQL."

However, if your program is connected to multiple database sessions, you can issue queries 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 go to statement.

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 (see page 503) statement must be syntactically within the BEGIN-END block that delimits the select loop.

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 can 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.

The INTO clause can include a structure that substitutes 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.

If using SELECT * to retrieve into a structure, ensure that the members of the structure have a one-to-one correspondence to the columns in the result table.

Retrieving Long Varchar and Long Byte Values

To retrieve long varchar and long byte columns, specify a DATAHANDLER clause in place of the host language variable in the INTO clause. For details about data handler routines, see the chapter "Embedded SQL" and the *Embedded SQL Companion* Guide. The syntax for the DATAHANDLER clause is as follows:

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

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 account for significant performance improvements on 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

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.

Related Statements

Create Index (see page 353)

Create Table (see page 402)

Create View (see page 448)

Delete (see page 470)

Endselect (see page 503)

Insert (see page 572)

Update (see page 702)

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.

Note: This statement has additional considerations when used in a distributed environment. For more information, see the *Ingres Star User Guide*.

Syntax

The SET statement has the following format:

```
[EXEC SQL] SET AUTOCOMMIT ON|OFF
[EXEC SQL] SET CONNECTION NONE | connection_name
[EXEC SQL] SET [NO]JOURNALING|JOURNALING [ON table_name]
[EXEC SQL] SET RESULT_STRUCTURE
                HEAP|CHEAP|HEAPSORT|CHEAPSORT|HASH|CHASH|ISAM
                              |CISAM|BTREE|CBTREE
[EXEC SQL] SET LOCKMODE SESSION|ON table_name WHERE
                [LEVEL = PAGE|TABLE|SESSION|SYSTEM|ROW]
                [, READLOCK = NOLOCK|SHARED|EXCLUSIVE
                              | SESSION|SYSTEM]
                [, MAXLOCKS = n|SESSION|SYSTEM]
                [, TIMEOUT = n|SESSION|SYSTEM|NOWAIT]
[EXEC SQL] SET [NO]LOCK_TRACE
[EXEC SQL] SET [NO]LOGGING
[EXEC SQL] SET [NO]OPTIMIZEONLY
[EXEC SQL] SET [NO]PARALLEL [degree of parallelism]
[EXEC SQL] SET [NO]PRINTQRY
[EXEC SQL] SET [NO]QEP
[EXEC SQL] SET JOINOP NOTIMEOUT | TIMEOUT | TIMEOUT nnnn
[EXEC SQL] SET JOINOP [NO]GREEDY
[EXEC SQL] SET SESSION
                [ADD PRIVILEGES ( priv {,priv} )
                |DROP PRIVILEGES ( priv {,priv}) ]
                [WITH
                        ON_ERROR = ROLLBACK STATEMENT|TRANSACTION
                        ON_USER_ERROR = ROLLBACK TRANSACTION|NOROLLBACK
                        | DESCRIPTION ='session_description'
                        | NODESCRIPTION
                        | PRIORITY = INITIAL|MINIMUM|MAXIMUM|priority
                        | PRIVILEGES = ( priv {, priv})|ALL|DEFAULT
                        | NOPRIVILEGES
                         ON_LOGFUL = COMMIT|ABORT|NOTIFY]
```

```
[EXEC SQL] SET SESSION READ ONLY|WRITE
              [, ISOLATION LEVEL SERIALIZABLE | REPEATABLE READ |
                  READ COMMITTED | READ UNCOMMITTED ]
[EXEC SQL] SET UPDATE_ROWCOUNT CHANGED | QUALIFIED
[EXEC SQL] SET WORK LOCATIONS ADD|DROP|USE
              location {, location}
[EXEC SQL] SET TRANSACTION READ ONLY|WRITE
              [, ISOLATION LEVEL READ UNCOMMITTED | READ COMMITTED |
                  REPEATABLE READ | SERIALIZABLE ]
[EXEC SQL] SET ROLE NONE | rolename [WITH PASSWORD = 'pwd'];
[EXEC SQL] SET [NO]RULES
[EXEC SQL] SET [NO]PRINTRULES
[EXEC SQL] SET NOMAXCOST | MAXCOST value
[EXEC SQL] SET NOMAXCPU | MAXCPU value
[EXEC SQL] SET NOMAXIO | MAXIO value
[EXEC SQL] SET NOMAXPAGE | MAXPAGE value
[EXEC SQL] SET NOMAXQUERY | MAXQUERY value
[EXEC SQL] SET NOMAXROW | MAXROW value
[EXEC SQL] SET NOMAXIDLE | MAXIDLE value
[EXEC SQL] SET NOMAXCONNECT | MAXCONNECT value
[EXEC SQL] SET [NO]PRINTDBEVENTS
[EXEC SQL] SET [NO]LOGDBEVENTS
[EXEC SQL] SET RANDOM_SEED [value]
[EXEC SQL] SET NOUNICODE_SUBSITUTION | UNICODE_SUBSTITUTION [substitution
character]
[EXEC SQL] SET CPUFACTOR [value]
[EXEC SQL] SET DATE_FORMAT [value]
[EXEC SQL] SET DECIMAL [value]
[EXEC SQL] SET [NO]FLATTEN
[EXEC SQL] SET [NO]HASH
[EXEC SQL] SET [NO]IO_TRACE
[EXEC SQL] SET [NO]LOG_TRACE
[EXEC SQL] SET MONEY_FORMAT [value]
[EXEC SQL] SET MONEY_PREC [value]
[EXEC SQL] SET [NO]OJFLATTEN
[EXEC SQL] SET [NO]STATISTICS
[EXEC SQL] SET [NO]TRACE POINT [value]
[EXEC SQL] SET [NO]TRACE OUTPUT | TRACE OUTPUT filename
```

Embedded Usage

When using the SET LOCKMODE statement in an embedded SET statement,
host language variables can be used to specify elements to the right of the
equal sign (=) in the WHERE clause.

Usage in OpenAPI, ODBC, JDBC, and .NET

In OpenAPI, ODBC, JDBC, and .NET interfaces, the following SET commands are not supported:

- SET CONNECTION NONE | *connection_name*

- SET [NO]PRINTQRY

The following SET commands are supported through interface-specific mechanisms:

- SET AUTOCOMMIT ON|OFF uses IIapi_autocommit().

- SET SESSION [READONLY | WRITE] uses IIapi_setConnectParam(), IIapi_modifyConnect(), and/or IIapi_setEnvParam() in OpenAPI.

- SET TRANSACTION uses IIapi_setConnectParam(), IIapi_modifyConnect(), and/or IIapi_setEnvParam() in OpenAPI.

All other SET statements can be sent as SQL.

Permissions

Permissions are assigned to users by a DBA. For more information on privileges, see the *Database Administrator Guide.*

- To issue the SET [NO]PRINTQRY, SET [NO]PRINTRULES, and SET [NO]RULES statements, you must have trace permissions.

- To issue the SET [NO]RULES statement, you must be the DBA of the database to which the session is attached when the statement is issued.

- To issue the SET LOCKMODE statement, the effective user of the session must have lockmode privilege. Lockmode privilege is assigned using the GRANT statement. For more information, see Grant (privilege) (see page 534).

Autocommit

The SET AUTOCOMMIT ON statement directs the DBMS Server to treat each query as a single-query transaction. SET AUTOCOMMIT OFF, the default, means an explicit COMMIT or ROLLBACK statement or terminating the session is required to terminate a transaction.

The SET AUTOCOMMIT statement cannot be issued in a multi-statement transaction. For a description of multi-statement transaction behavior, see the chapter "Working with Transactions and Handling Errors."

Connection

The SET CONNECTION *connection_name* statement switches the current session to a connection previously established using the CONNECT statement.

To determine the current connection name, use:

```
DBMSINFO('CONNECTION_NAME')
```

The SET CONNECTION NONE statement results in no current session.

[No]Journaling

The default for journaling for a DBMS Server is established by the DEFAULT_JOURNALING option in CBF. Default journaling can be overridden by the SET [NO]JOURNALING statement, which then defines the default journaling state for tables created by the session after the statement is issued. If a table is created with the WITH [NO]JOURNALING clause that becomes the default journaling state for the table.

Important! Regardless of whether journaling is enabled for a table, journaling only occurs when journaling is enabled for the database. Journaling for the entire database is turned on or off using the ckpdb command. For details about ckpdb, see the *Command Reference Guide*.

If the current journaling status of the table is OFF, and you want to enable journaling for this table after the next checkpoint, use the SET JOURNALING ON *tablename* statement.

Note: Journaling status can be enabled only when table is first created or after a checkpoint in which case the checkpoint has a consistent version of the table against which subsequent journals can be applied.

To immediately cease journaling against a table, use the SET NOJOURNALING ON *tablename* clause. The HELP TABLE *tablename* statement shows that journaling is disabled. Journaling may be immediately stopped for a database by using the -disable_journaling option of the alterdb command. (For details, see the Command Reference Guide.)

Result_Structure

The SET RESULT_STRUCTURE statement sets the default storage structure for tables created with the AS clause of the CREATE TABLE statement. This storage structure can be any of the structures described in the MODIFY statement, that is, heap, cheap, heapsort, cheapsort, hash, chash, btree, cbtree, isam or cisam.

If a SET RESULT_STRUCTURE statement is not executed, the default storage structure for a table created by the CREATE TABLE AS statement is cheap. For example:

```
set result_structure hash;
create temp as select id ... ;
```

does the same as:

```
create temp as select id ... ;
modify temp to hash;
```

For btree, cbtree, hash, chash, isam, and cisam, the newly created table is automatically indexed on the first column. This results in the above table "temp" being created with a storage structure of hash, keyed on the first column "id".

Lockmode

The SET LOCKMODE statement sets different types and levels of locks that override the default locking strategy. Use this statement to optimize performance or enforce stricter concurrency control.

When beginning a session, the system defaults (as defined for the DBMS in CBF) are in effect. Using the SET LOCKMODE statement, these defaults can be changed.

Locking parameters can be established for all tables and indexes accessed during the session, or different locking parameters can be established for different tables or indexes.

The SET LOCKMODE command also allows you to set locking back to either session or system

The SET LOCKMODE statement cannot be issued within a transaction, except for the statement: SET LOCKMODE … WITH TIMEOUT=n|SESSION|SYSTEM|NOWAIT.

The following SET LOCKMODE parameters control locking for a session:

level

Specifies locking behavior. It must be one of the following locking levels:

row

Takes row-level locks.

If row-level locking is specified and the number of locks granted during a query exceeds the system-wide lock limit, or if the locking resources of the operating system are depleted, locking escalates to table level. This escalation occurs automatically and is independent of the user.

page

Takes page-level locks.

If page-level locking is specified and the number of locks granted during a query exceeds the system-wide lock limit, or if the locking resources of the operating system are depleted, locking escalates to table level. This escalation occurs automatically and is independent of the user. Page is the default.

table

Takes table-level locks.

session

Takes locks according to the default in effect for the session.

system

> Starts with page-level locking. If the optimizer estimates that more than maxlocks pages are referenced, uses table level locking.

readlock

Specifies the type of locking that applies when accessing a table to read the rows. It does not apply when accessing the table for INSERT, UPDATE or DELETE or the table that is the object of an INSERT ... INTO SELECT ... FROM or CREATE TABLE ... AS SELECT. Any of the following modes can be specified:

nolock

> Takes no locks when reading data.

shared

> Takes shared locks when reading data; this is the default mode of locking when reading data.

exclusive

> Takes exclusive locks when reading data; useful in "select-for-update" processing within a multi-statement transaction.

session

> Takes locks according to the current readlock default for your session.

system

> Takes locks according to the readlock default, which is shared locks.

maxlocks

Specifies the maximum number of page locks taken on a table before locking escalates to a table lock. The number of locks available is dependent upon your system configuration. The following escalation factors can be specified:

n

> Specifies the number of page locks to allow before escalating to table level locking. *n* must be an integer greater than 0.

session

> Specifies the current maxlocks default for your session.

system

> Specifies the maxlocks default, which is 50.

timeout

Specifies how long, in seconds, a lock request can remain pending. If the DBMS Server cannot grant the lock request within the specified time, the query that requested the lock aborts. Valid settings are:

n

Specifies the number of seconds to wait; *n* must be a non-negative integer. If 0 is specified, the DBMS Server waits indefinitely for the lock.

nowait

Specifies that when a lock request is made that cannot be granted without incurring a wait, control is immediately returned to the application that issued the request.

session

Specifies the current timeout default for the session.

system

Specifies the default, which is no timeout.

[No]Lock_Trace

The SET LOCK_TRACE statement enables the display of ongoing locking activity for the current session, including information about the locks used and released by your statements. Lock tracing can be started or stopped at any time during a session. For additional information regarding the usage and output of this statement, see the *Database Administrator Guide*.

Important! Use SET LOCK_TRACE as a debugging or tracing tool only. LOCK_TRACE is not a supported feature, which means that the format and layout of the output can be changed without notification.

[No]Logging

The SET NOLOGGING statement allows a session to bypass the logging and recovery system. This may speed up certain types of update operations but must be used with extreme caution.

The SET NOLOGGING statement is intended to be used for operations for which the reduction of logging overhead and log file space usage outweigh the benefits of having to recover from transaction abort. Do not use the SET NOLOGGING statement to try to improve performance during everyday use of a production database.

Caution! If journaling is set for the database within a session where NOLOGGING was set, **none** of the changes will be journaled.

When transaction logging is disabled, any error that occurs when the database is being updated (including interrupts, deadlock, lock timeout, and forced abort) or any attempt to rollback a transaction causes the DBMS Server to mark the database inconsistent.

To use the SET NOLOGGING option it is recommended that the DBA:

- Obtain exclusive access on the database to ensure that no other sessions are active against the tables being updated.

- Be prepared to recover the database. There are two cases:
  - For existing databases: Checkpoint the database prior to executing the SET NOLOGGING statement. If an error occurs, the database can be restored from the checkpoint.
  - Loading a new database: If an error occurs destroy the inconsistent database, create a new database and restart the load operation.

To enable transaction logging, issue the SET LOGGING statement

This SET statement can only be issued by the DBA of the database on which the session is operating and cannot be issued within a multi-statement transaction.

If SET NOLOGGING was in effect on a journaled database, we recommend that a checkpoint be taken immediately after the SET LOGGING statement is issued. When a session in SET NOLOGGING mode disconnects from a database, the DBMS Server executes a SET LOGGING operation to bring the database to a guaranteed consistent state before completing the disconnect.

[No]Optimizeonly

OPTIMIZEONLY specifies whether query execution should halt after the completion of query optimization.

Use SET OPTIMIZEONLY in conjunction with SET QEP when there is a requirement to view query execution plans without executing a query.

[No]Parallel

The SET PARALLEL statement controls whether the query optimizer enables the generation of parallel query execution plans.

The optional *degree of parallelism* value indicates the number of exchange nodes (or points of concurrency) built into the plan. The default value is 8 and the maximum is 32.

The SET NOPARALLEL statement (or setting the *degree of parallelism* value to 0 or 1) prevents the optimizer from creating parallel query plans.

Note: When tracing the I/O or the locks of a parallel query (using SET IO_TRACE or SET LOCK_TRACE with SET PARALLEL), the trace messages from child threads of the QEP are logged to the II_DBMS_LOG. The trace messages for the main thread are sent to the user session in the normal manner.

For a discussion of parallel query plans, see the *Database Administrator Guide*.

[No]Printqry

The SET PRINTQRY statement displays each query and its parameters as it is passed to the DBMS Server for processing.

The SET NOPRINTQRY option disables this option.

[No]Qep

The SET QEP statement displays a diagrammatic representation of the query execution plan chosen for the query by the optimizer.

The SET NOQEP option disables this option. For a discussion of query execution plans, see the *Database Administrator Guide*.

Joinop [No]Timeout

This statement changes the timeout point of the query optimizer. When the query optimizer is checking query execution plans, it stops when it believes that the best plan that it has found takes less time to execute than the amount of time already spent searching for a plan or when it has evaluated all possible query plans, whichever is reached first.

Note: This option has no effect if the GREEDY option is in effect.

If a SET JOINOP TIMEOUT nnnn is issued (where nnnn is in milliseconds) the query optimizer stops looking for query execution plans after that number of milliseconds.

If a SET JOINOP NOTIMEOUT statement is issued, the optimizer continues searching ALL possible query plans. This option is often used with the SET QEP option to ensure that the optimizer is picking the best possible query plan.

A SET JOINOP TIMEOUT statement restores the default TIMEOUT.

Joinop [No]Greedy

This statement enables or disables the complex query enumeration heuristic of the query optimizer. The greedy heuristic enables the optimizer to produce a query plan much faster than with its default technique of exhaustive searching for query execution plans when the query references a large number of tables. For a discussion of the greedy optimization heuristic, see the *Database Administrator Guide*.

Session with Add Privileges

The SET SESSION with ADD PRIVILEGES option obtains a requestable privilege while connected to Ingres. A requestable privilege is defined in the privileges list of the ALTER PROFILE, ALTER USER, CREATE PROFILE, or CREATE USER statements, but is not defined in the corresponding default privileges list.

Session with Drop Privileges

The SET SESSION with DROP PRIVILEGES option removes all privileges (including default and requestable) for the current session.

Session with On_error

The SET SESSION WITH ON_ERROR option specifies how transaction errors are handled in the current session.

To direct the DBMS Server to rollback the current transaction if an error occurs, specify ROLLBACK TRANSACTION.

To direct the DBMS Server to rollback only the current statement specify ROLLBACK statement - this is the default

To determine the current status of transaction error handling, issue the SELECT DBMSINFO('ON_ERROR_STATE') statement.

Specifying ROLLBACK TRANSACTION reduces logging overhead and can help performance. The performance gain is offset by the fact that, if an error occurs, the entire transaction is rolled back. The following errors always roll back the current transaction regardless of the current transaction error-handling setting:

- Deadlock

- Forced abort

- Lock quota exceeded

To determine if a transaction was aborted as the result of a database statement error, issue the SELECT DBMSINFO('TRANSACTION_STATE') statement. If the error aborted the transaction, this statement returns 0, indicating that the application is currently not in a transaction.

The SET SESSION WITH ON_ERROR statement cannot be issued from within a database procedure or multi-statement transaction.

Note: SQL syntax errors (including most messages beginning with E_US) do not cause a rollback. Only errors that occur during execution of the SQL statement cause a rollback.

Session with On_user_error

The SET SESSION WITH ON_USER_ERROR option enables you to specify how user errors are handled in the current session. To direct the DBMS to roll back the effects of the entire transaction if a user error occurs, specify ROLLBACK TRANSACTION. To revert back to default behavior, specify NOROLLBACK.

Session with [No]Description

The SET SESSION WITH [NO]DESCRIPTION statement is used to set the session description which is then visible in Ingres management and administration tools such as iimonitor, ipm, and ima. The maximum length of a session description is 256 characters.

If NODESCRIPTION is specified, the session description is set empty.

In ESQL, the session description can be specified using a string variable.

Session with Priority

The SESSION_PRIORITY option sets the relative importance of a session. A session can have a higher (more important) or lower (less important) priority than a session with the base or normal priority.

Only users with the SESSION_PRIORITY resource privilege can change their session priority, subject to the limit defined by the system administrator. Three fixed settings and one variable setting are available:

INITIAL

Uses the value known at session startup.

MINIMUM

Uses the lowest (least important) priority.

MAXIMUM

Uses the highest (most important) priority.

priority

Uses the session priority defined. The priority variable is specified as an integer. A positive value sets a higher priority than the base priority. A negative value sets a lower priority than the base priority. Due to operating system restrictions, the number of values above and below the base priority may be limited.

Session with [No]Privileges

When an Ingres session starts, it receives a default set of privileges. The SET SESSION WITH [NO]PRIVILEGES statement can be used to add, drop, or set the session effective privileges.

The following options are available:

ADD PRIVILEGE

Adds the listed privileges to the pre-existing effective privileges for that session, provided the user has permission to use them.

DROP PRIVILEGE

Removes the listed privileges from the effective privileges for that session.

WITH NOPRIVILEGES

Sets no effective privileges for the session.

WITH PRIVILEGES=(*priv {,priv}*)

Replaces the session privileges by the listed privileges, the user must have permission to use them.

WITH PRIVILEGES=ALL

Adds all granted privileges to the effective privileges for that session.

WITH PRIVILEGES=DEFAULT

Resets the session privileges to the default privileges as at session startup.

Note: Only one of these options can be specified. Ingres applications that require specific privileges must use this command to ensure the session has the required privileges.

Session with on_logfull

When the transaction log file fills, the oldest transaction is, by default, aborted to free up log space. The SET SESSION WITH ON_LOGFULL statement can be used to modify this behavior.

The ON_LOGFULL= option takes the following parameters:

COMMIT

Silently (without notification) commits the transaction.

ABORT

Aborts the transaction.

NOTIFY

Commits the transaction and writes notification of the commit to the DBMS log.

ON_LOGFULL can appear any number of times during a transaction, either within or without the scope of the transaction.

When specifying COMMIT or NOTIFY, if a logfull condition occurs, the multi-row update can be partially committed. For example, using the following syntax commits the rows deleted from table_name up to the moment the logfull condition is detected.

```
[EXEC SQL] SET SESSION WITH ON_LOGFULL COMMIT;

[EXEC SQL] DELETE FROM table_name;
```

If the transaction is aborted after the commit point, only the post-commit updates are rolled back.

Session

The SET SESSION statement must be issued before a transaction commences and the settings apply until that session is completed.

Ingres supports the following ANSI/ISO SQL92 standard levels of isolation:

READ UNCOMMITTED

This level is equivalent to what is commonly known as readlock=nolock. Isolation level must be specified for an entire transaction.

If session access mode is not specified and level of isolation is READ UNCOMMITTED, READ ONLY is implicit.

READ COMMITTED

This isolation level always sees committed data because data is locked before being read. The lock remains in effect until the cursor is positioned to the next page/row or closed. However, if any data on the current item of the cursor is changed, the lock is held until the transaction commits. When acquiring a lock on a page/row, Ingres releases the previous page/row if the previous page/row was not updated.

REPEATABLE READ

Data is always locked before being read. Locks are released only on nonqualifying data. All other locks are held until the end of the transaction.

SERIALIZABLE

Locks are required on all data before being read. No locks are released until the transaction ends.

SET SESSION AUTHORIZATION *username*

This allows a user with security or dbadmin privilege to immediately alter their effective user to another user within that session.

READ ONLY

When a session access mode of READ ONLY is specified, insert, update, delete, copy, and DDL operations are disallowed, and an SQLSTATE of 25000 (invalid session state) is returned. Temporary tables are the exception and are always writable.

When a READ ONLY session is begun, it registers itself with the logging system and is allowed to proceed even when a ckpdb is pending against the database for the session.

READ WRITE

If READ WRITE is specified, the level of isolation can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

If session access mode is not specified and level of isolation is READ UNCOMMITTED, READ ONLY is implicit. Otherwise, READ WRITE is implicit.

> **Note:** The access mode of a session has no effect on the locking mode of the session.

Update_Rowcount

The SET UPDATE_ROWCOUNT statement specifies the nature of the value returned by the INQUIRE_SQL(ROWCOUNT) statement.

Valid options are:

CHANGED

   Returns the number of rows changed by the last query.

QUALIFIED

   (Default) Returns the number of rows that qualified for change by the last query.

For example, consider the following table called test_table:

column1	column2	column3
Jones	000	green
Smith	000	green
Smith	000	green

and the following query:

```
UPDATE test_table SET column1 = 'Jones'
       WHERE column2 = 000 AND column3 = 'green';
```

The DBMS Server, for reasons of efficiency, does not actually update the first row because column1 already contains "Jones". However, the row does qualify for updating by the query.

If the UPDATE_ROWCOUNT option is set to CHANGED, INQUIRE_SQL(ROWCOUNT) returns 2 (the number of rows actually changed).

If the UPDATE_ROWCOUNT option is set to QUALIFIED, INQUIRE_SQL(ROWCOUNT) returns 3 (the number of rows that qualified to be changed).

To determine the setting for the UPDATE_ROWCOUNT option, issue the following statement:

```
SELECT DBMSINFO('UPDATE_ROWCNT')
```

Work Locations

The SET WORK LOCATIONS statement adds, removes, or changes the set of locations that the DBMS Server uses for sorting.

Sorting can occur when:

- Queries are executed. This may be an ORDER BY or as part of the Query Execution Plan (for example, a Full Sort Merge Join).

- Tables or indexes are modified.

To add work locations to the set of locations currently in use, issue the SET WORK LOCATIONS ADD statement and specify the locations to be added.

To remove locations from the set of locations currently in use, issue the SET WORK LOCATIONS DROP statement.

To replace the set of locations currently in use, issue the SET WORK LOCATIONS USE statement.

Work locations are defined using the CREATE LOCATION statement.

For details about configuring your installation to improve the performance of sorting, see the *Database Administrator Guide*.

Transaction

The SET TRANSACTION statement must be issued before a transaction commences and the settings last only until that transaction is completed.

Ingres supports the ANSI/ISO SQL92 standard *levels of isolation*:

READ UNCOMMITTED

This level is equivalent to what is commonly known as readlock=nolock.

READ COMMITTED

This isolation level always sees committed data because data is locked before being read. The lock remains in effect until the cursor is positioned to the next page/row or closed. However, if any data on the current item of the cursor is changed, the lock is held until the transaction commits. When acquiring a lock on a page/row, Ingres releases the previous page/row if the previous page/row was not updated.

REPEATABLE READ

With this isolation level, data is always locked before being read. Locks are released only on nonqualifying data. All other locks are held until the end of the transaction.

SERIALIZABLE

Locks are required on all data before being read. No locks are released until the transaction ends.

READ ONLY

When a transaction access mode of READ ONLY is specified, insert, update, delete, copy, and DDL operations are disallowed, and an SQLSTATE of 25000 (invalid session state) is returned. Temporary tables are the exception and are always writable.

When a READ ONLY transaction is begun, it registers itself with the logging system and is allowed to proceed even when a ckpdb is pending against the database for the session.

READ WRITE

If READ WRITE is specified, the level of isolation can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.

If *transaction access mode* is not specified and level of isolation is READ UNCOMMITTED, READ ONLY is implicit. Otherwise, READ WRITE is implicit.

Note: The access mode of a transaction has no effect on the locking mode of the transaction.

[No]Rules

The SET NORULES option disables any rules (user or system created) that would have been executed during the session. Existing rules and rules created during the session are disabled. To re-enable rules firing, issue the SET RULES statement. By default, rules are enabled.

To issue this statement, you must be the DBA of the database to which the session is connected.

Caution! After issuing the SET NORULES statement, the DBMS Server does not enforce check and referential constraints on tables nor the check option for view.

[No]Printrules

The SET PRINTRULES statement causes the DBMS Server to send a trace message to the application each time a rule is fired. This message identifies the rule and the associated database procedure that is invoked as a result of the rule's firing.

Issue the SET NOPRINTRULES to disable rule-related trace messages. By default, rule-related trace messages are not displayed.

[No]Maxcost

The SET MAXCOST statement restricts the maximum cost per query on the database in terms of disk I/O and CPU usage. The MAXCOST value must be less than or equal to the session's value for query_cost_limit. If no query_cost_limit is set, there is no limit on cost usage per query. To set query_cost_limit for a user, use the GRANT statement.

When maxcost is set, it remains in effect until another SET MAXCOST or SET NOMAXCOST statement is issued or the session terminates.

For more information, see query_cost_limit in the description of Grant (privilege) (see page 534).

[No]Maxcpu

The SET MAXCPU statement restricts the maximum CPU usage per query on the database. The MAXCPU value must be less than or equal to the session's value for query_cpu_limit. If no query_cpu_limit is set, there is no limit on cpu usage per query. To set query_cpu_limit for a user, use the GRANT statement.

When MAXCPU is set, it remains in effect until another SET MAXCPU or SET NOMAXCPU statement is issued or the session terminates.

For more information, see query_cpu_limit in the description of Grant (privilege) (see page 534).

[No]Maxio

The SET MAXIO statement restricts the estimated number of I/O operations that can be used by each subsequent query to the *value* specified. *Value* must be less than or equal to the session's value for query_io_limit. If no query_io_limit is set, there is no limit on the amount of I/O performed

When MAXIO is set, it remains in effect until another SET MAXIO or SET NOMAXIO statement is issued or the session terminates.

To set query_io_limit for a user, use the GRANT statement. For more information, see Grant (privilege) (see page 534).

[No]Maxpage

The SET MAXPAGE statement restricts the maximum number of pages per query on the database. The value must be less than or equal to the session's value for query_page_limit. If no query_page_limit is set, there is no limit on max page usage per query.

When MAXPAGE is set, it remains in effect until another SET MAXPAGE or SET NOMAXPAGE statement is issued or the session terminates.

To set query_page_limit for a user, use the GRANT statement.

[No]Maxquery

The SET MAXQUERY statement is an alias for SET MAXIO. For details, see [No]Maxio (see page 691).

[No]Maxrow

The SET MAXROW statement restricts the estimated number of rows that can be returned by subsequent queries. *Value* must be less than or equal to the session's value for query_row_limit. If no query_row_limit is set, there is no limit on the number of rows returned.

When MAXROW is set, it remains in effect until another SET MAXROW or SET NOMAXROW statement is issued, or the session terminates.

For more information, see query_row_limit in the description of Grant (privilege) (see page 534).

[No]Maxidle

The SET MAXIDLE option specifies whether a time limit is in force, and how long it is in seconds. The value entered must be less than that defined by the idle_time_limit session privilege. If no idle_time_limit is set, there is no time limit.

When MAXIDLE is set, it remains in effect until another SET MAXIDLE or SET NOMAXIDLE statement is issued, or the session terminates. For more information, see idle_time_limit in the description of Grant (privilege) (see page 534).

[No]Maxconnect

The SET MAXCONNECT statement is an alias for SET MAXIDLE. For details, see [No]Maxidle (see page 692).

[No]Printdbevents

The SET [NO]PRINTDBEVENTS option enables or disables display of event trace information for the application that raises events.

To enable the display of trace information, specify SET PRINTDBEVENTS.

To disable the display of trace information, specify SET NOPRINTDBEVENTS.

This option displays only events raised by the application issuing the SET statement. Events received by the application are not displayed.

[No]Logdbevents

The SET [NO]LOGDBEVENTS option enables or disables logging of event trace information for the application that raises events. When logging is enabled, event trace information is written to the installation log file.

To enable logging, specify SET LOGDBEVENTS.

To disable logging, specify SET NOLOGDBEVENTS.

Only events raised by the application issuing the SET statement are logged. Events received by the application are not logged.

Random_seed

This statement sets the beginning value for the random functions. 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 entire 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 enhance the "randomness" of the returned value. The seed value can be any integer.

If you omit the value, the value is a multiple of the process ID and the number of seconds since 1/1/1970.

Cpufactor

If you are working on a machine where CPU operations are more expensive than I/O operations, the default CPUFACTOR can be set to a lower value. This causes the Optimizer to look at less CPU-intensive operations.

The default is 100 CPU operations to one disk I/O.

For example: Setting CPUFACTOR to 10 indicates that one disk I/O is roughly equivalent to 10 CPU units.

Date_format

SET DATE_FORMAT specifies the format for date values. This option corresponds to the Ingres environment variable II_DATE_FORMAT and is assigned the same values. If set, DATE_FORMAT replaces the currently configured format with an alternate format. The default format setting is US. See the *System Administrator Guide* for a list of valid settings.

Decimal

SET DECIMAL specifies the character to be used as the decimal point in numeric literals. This option corresponds to the Ingres environment variable II_DECIMAL and is assigned the same values. Valid characters are the period (.) (as in 12.34) and the comma (,) (as in 12,34). The default is the period. See the *System Administrator Guide* for more information on setting this option.

[No]Flatten

SET FLATTEN specifies that query flattening be used to optimize queries, including queries involving aggregate subselects or singleton subselects.

By default, Ingres performs query flattening. (The ANSI entry SQL-92 standard specifies that query flattening is not performed.)

Use the SET NOFLATTEN statement to disable query flattening.

[No]Hash

SET HASH allows the query optimizer to consider using HASH joins.

A hash join is one in which a hash table is built with the rows of one of the join sources by hashing on the key columns of the join. The rows of the other join source are then read and hashed into the table on their key columns. The hashing of the second set of rows should quickly identify pairs of joining rows. The advantage of this technique is that it requires no index structures on the join columns (as does KEY join), nor does it require sorting on the join columns (as does merge join).

SET HASH is the default.

To stop the query optimizer from considering the use of HASH joins, use the SET NOHASH statement.

[No]Io_trace

SET IO_TRACE prints information about disk I/O during the life of each query in the session. Using this option requires the trace privilege.

Use SET NOIO_TRACE to disable this tracing option.

See I/O Tracing in the *System Administrator Guide* for operating system-specific command and output examples.

[No]Log_trace

SET LOG_TRACE starts tracing of logfile writes. To stop tracing of logfile writes, use the SET NOLOG_TRACE statement. Using this option requires the trace privilege.

When you use SET LOG_TRACE during a session, you receive a list of the log records written during execution of your query, along with other information about the log. Set log_trace output includes:

- The length of the log and the amount of space reserved for its CLR

- If the log_write is a normal log record (do/redo) or a CLR

- If the log record can be copied to the journal file

- If the log is associated with a special recovery action

For information on the use of the SET LOG_TRACE statement, see the *Database Administrator Guide*.

Money_format

SET MONEY_FORMAT specifies the format for money output. This option corresponds to the Ingres environment variable II_MONEY_FORMAT and is assigned the same values. The default is L:$. The symbol to the left of the colon indicates the location of the currency symbol. It must be "L" for a leading currency symbol or a "T" for a trailing currency symbol. The symbol to the right of the colon is the currency symbol you want displayed. Currency symbols can contain up to four physical characters. For example:

Logical Definition	Result
L:$	$100
T:DM	100DM
T:F	100F

Money_prec

SET MONEY_PREC specifies the number of decimal places to be displayed for money values. This option corresponds to the Ingres environment variable II_MONEY_FORMAT and is assigned the same values. Valid values are 0, 1, and 2. The default is 2 (for decimal currency).

[No]Ojflatten

SET NOOJFLATTEN prevents the query optimizer from using the transformation that converts a NOT EXISTS/NOT IN subselect to an outer join with the containing query. The default is SET OJFLATTEN.

[No]Statistics

SET NOSTATISTICS tells the query optimizer to ignore any statistics found in IIHITOGRAMS or IISTATISTICS and to use its default values.

The default is SET STATISTICS.

[No]Trace Point

SET TRACE POINT activates specified trace points for the current session.

Trace points are not officially supported, as Ingres reserves the right to change their effect and output without notification.

A trace point starts with a Facility Identifier followed by a number of digits. Some trace points have additional parameters. For example:

- DM420
- DM421
- QS501
- QE90

Note: Some trace points are equivalents of SET statements. For example, SET TRACE POINT QE5 is equivalent to SET NORULES.

Caution! Some trace points may cause irreversible damage to Ingres. Trace point usage is entirely at the user's own risk.

[No]Trace Output

SET TRACE OUTPUT causes trace information to be written to a specified file. For example, the following statement causes information from some trace points (such as DM420) to be sent to the *full_path_name_of_file* file.

```
SET TRACE OUTPUT full_path_name_of_file
```

SET NOTRACE OUTPUT causes trace information to be written to the default trace output file.

Related Statements

Commit (see page 323)

Create Table (see page 402)

Grant (privilege) (see page 534)

Examples: Set

The following are SET statement examples:

1. Create three tables with journal logging enabled and one without.

```
set journaling;
create table withlog1 ( ... );
create table withlog2 ( ... );
set nojournaling;
create table withlog3 ( ... ) with journaling;
create nolog1 ( ... );
```

2. Create a few tables with different structures.

```
create table a as ...;/* heap */
set result_structure hash;
create table b as select id ...;/* hash on 'id' */
set result_structure heap;
create table d as select id ...;     /* heap again */
```

3. Set lockmode parameters for the current session. Tables accessed after executing this statement are governed by these locking behavior characteristics.

```
set lockmode session where level = page,
    readlock = nolock,
    maxlocks = 50, timeout = 10;
```

4. Set the lockmode parameters explicitly for a particular table.

```
set lockmode on employee
    where level = table, readlock = exclusive,
    maxlocks = session, timeout = 0;
```

5. Reset your session default locking characteristics to the system defaults.

```
set lockmode session where level = system,
    readlock = system,
    maxlocks = system, timeout = system;
```

6. Switch sessions in a multi-session application.

```
set session connection personnel;
```

7. Set the session description to *'Payroll App: Generating invoices*'.

```
set session
    with description = 'payroll app: generating invoices';
```

8. Set the session priority to 5 below the normal base priority.

```
set session with priority = -5
```

9. Restore the initial session priority.

```
set session with priority = initial;
```

10. Changes the session role to *clerk.*

```
set role clerk with password='clerkpassword';
```

Set_sql

Valid in: ESQL

The SET_SQL statement specifies runtime options for the current session.

Use SET_SQL to switch sessions in a multiple session application, specify whether local or generic errors are returned to a session, change the default behavior when a connection error is experienced, or set trace functions. To determine the settings for the current session, use the INQUIRE_SQL statement.

SET_SQL can be used to override the II_EMBED_SET environment variable/logical. For information about II_EMBED_SET, see the *System Administrator Guide.*

Note: While SET_SQL is not supported in OpenAPI, Iiapi_catchEvent() operates similarly to embedded SQL statement SET_SQL(DBEVENTHANDLER = dbevent_handler).

Syntax

The SET_SQL statement has the following format:

```
EXEC SQL SET_SQL (object = value {, object = value})
EXEC SQL SET_SQL (SESSION = NONE)
```

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 of the error return variable dbmserror.
errorhandler	function pointer	Specifies a user-defined routine to be called when an SQL error occurs in an embedded application. The error handler must be specified as a function pointer.
Errorno	Integer	Sets the value of the error return variable errorno.
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 Server error numbers. Generic error numbers are returned by default. For information about the interaction of local and generic errors, see the chapter "Working with Transactions and Handling Errors."
Gcafile	character string	Specifies an alternate text file to which the DBMS Server writes GCA information. The default filename 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.

Object	Data Type	Description
messagehandler	function pointer	Specifies a user-defined routine to be called when a database procedure returns a message to an application. The message handler must be specified as a function pointer.
prefetchrows	Integer	Specifies the number of rows the DBMS Server buffers when fetching data for readonly cursors. Valid arguments are: 0 (default) - The DBMS calculates the optimum number of rows to prefetch. 1 - Disables prefetching. n: (positive integer) - Specifies the number of rows the DBMS prefetches. For details, see the chapter "Working with Embedded SQL."
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 iiprtgca.log file in the current directory). Specify 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 iiprtqry.log file in the current directory). Specify 1 to turn the feature on or 0 to turn the feature off.
Printtrace	Integer	Enables or disables trapping of DBMS server 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 the DBMS Server aborts on the following errors: An application issues a query, but is not connected to a database The DBMS Server fails Communications services fail Specify 1 to abort on these conditions

Object	Data Type	Description
Qryfile	character string	Specifies an alternate text file to which the DBMS Server writes query information. The default filename 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 or disables saving of the text of the last query issued. Specify 1 to enable, 0 to disable. To obtain the text of the last query, issue the inquire_sql(:*query*=querytext) statement. Use the inquire_sql(:*status*=savequery) statement to determine whether saving is enabled.
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 the DBMS Server writes tracepoint information; the default filename 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.

Issuing the SET_SQL (session = none) statement results in the state being identical to prior to the first CONNECT statement or following a DISCONNECT statement: no current session.

Permissions

This statement is available to all users.

Related Statements

Inquire_sql (see page 562)

Update

Valid in: SQL, ESQL, DBProc, OpenAPI, ODBC, JDBC, .NET

The UPDATE statement updates column values in a table.

Syntax

The UPDATE statement has the following format:

Interactive version:

```
UPDATE [schema.]table_name [corr_name]
            [FROM [schema.]table_name [corr_name]
            { , [schema.]table_name [corr_name]}]
            SET column_name = expression {, column_name = expression}
            [WHERE search_condition];
```

Embedded non-cursor version:

```
EXEC SQL [REPEATED] UPDATE [schema.]table_name [corr_name]
            [FROM [schema.]table_name [corr_name]
            { , [schema.]table_name [corr_name]}]
            SET column = expression {, column = expression}
            [WHERE search_condition];
```

Embedded cursor version:

```
EXEC SQL UPDATE [schema.]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 "Understanding the Elements of SQL Statements." If a row update violates an integrity constraint on the table, the update is not performed. For details about integrity constraints, see Create Table (see page 402).

table_name specifies the table for which the constraint is defined. A correlation name *(corr_name)* can be specified for the table for use in the *search_condition*. For a definition of correlation names and discussion of their use, see the chapter "Introducing SQL."

The expressions in the set clause can use constants or column values from the table being updated or any tables listed in the from clause.

If a column name specifies a numeric column, its associated *expression* must evaluate to a numeric value. Similarly, if a column name represents a character type, its associated *expression* must evaluate to a character type.

The result of a correlated aggregate cannot be assigned to a column. For example, the following update statement is invalid:

```
update mytable from yourtable

        set mytable.mycolumn = max(yourtable.yourcolumn);
```

To assign a null to a nullable column, use the null constant.

Note: To update long varchar or long byte columns, specify a datahandler clause in place of the host language variable in the set clause. For details about data handler routines, see the chapter "Working with Embedded SQL" and the *Embedded SQL Companion Guide*. The syntax for the datahandler clause is as follows:
```
DATAHANDLER(handler_routine ([handler_arg]))[:indicator_var]
```

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:
```

Embedded Usage

Host language variables can only be used within expressions in the SET clause and the *search_condition*. (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, as when it is used with the Ingres forms system query mode.

The non-cursor update can be formulated as a repeated query by using the keyword repeated. Doing so reduces the overhead required to run the same update repeatedly within your program. The repeated keyword directs the DBMS Server to save the query execution plan when the update is first executed.

This encoding can account for significant performance improvements on subsequent executions of the same update. The repeated keyword is available only for non-cursor updates; it is ignored if used with the cursor version. Repeated update cannot be specified as a dynamic SQL statement.

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 to change the *search_condition* after the initial execution of the statement. The saved execution plan is based on the initial value of the *search_condition* and any changes to *search_condition* are ignored. This rule does not apply to simple variables used in *search_conditions*.

Permissions

You must own the table or have update privilege. If the statement contains a WHERE clause that specifies columns of the table being updated, you must have both select and update permissions; otherwise, update permission alone is sufficient.

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 is 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 causes the same row to be updated twice if the cursor was opened in the direct update mode. If the cursor was opened in deferred update mode, more than one update cannot be issued against a row, and the update cannot be followed by a DELETE statement on the same row. Attempting to do either results in an error indicating an ambiguous update operation.

If the table was created with no duplicate rows allowed, the DBMS Server returns an error if attempt is made to insert a duplicate row.

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 see the same database table.

- The columns in the SET clause must have been declared for update at the time the cursor was declared.

- Host language variables can be used only for the cursor names or for expressions in the SET clause.

When executing a cursor update dynamically, using the PREPARE statement, the cursor must be open before the cursor UPDATE statement can be prepared. The prepared statement remains valid while the cursor is open. If the named cursor is closed and reopened, re-prepare the corresponding UPDATE statement. If an attempt is made to execute the UPDATE statement associated with the previously open cursor, the DBMS Server issues an error.

Both the COMMIT and ROLLBACK statements implicitly close all open cursors. A common programming error is to update the current row of a cursor, commit the change, and continue in a loop to repeat the process. This process fails because the first commit closes the cursor.

If the statement does not update any rows, the sqlcode of the SQLCA is set to 100. The sqlerrd(3) of the SQLCA indicates the number of rows updated by the statement.

Locking

The UPDATE statement acquires page locks for each row in the table that is evaluated against the WHERE clause.

Related Statements

Delete (see page 470)

Insert (see page 572)

Select (interactive) (see page 641)

Examples: Update

The following examples demonstrate how to replace the values of the specified columns by the values of the specified expressions for all rows of the table that satisfy the *search_condition*:

1. Give all employees who work for Smith a 10% raise.

```
update emp
    set salary = 1.1 * salary
    where dept in
        (select dno
        from dept
        where mgr in
            (select eno
            from emp
            where ename = '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 = 'Smith'));
```

3. Update the salary of all employees having names that begin with "e," using the value for a standard raise in the table dept.

```
update employee e
    from dept d
    set salary = d.std_raise * e.salary
    where e.name like 'e%' and d.dname = e.dname
```

Whenever

Valid in: ESQL

The WHENEVER statement enables your application to handle error and exception conditions arising from embedded SQL database statements. The WHENEVER statement directs the DBMS Server to perform a specified action 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. The WHENEVER statement has lexical (as opposed to logical) scope. For details, see the chapter "Working with Transactions and Handling Errors."

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 SQL database statement. The sqlcode of the SQLCA is set to a negative number.

SQLMESSAGE

Indicates that a message statement in a database procedure has executed. The sqlcode variable of the SQLCA is set to 700. If the database procedure is invoked by a rule, MESSAGE statements issued by the database procedure do not set sqlcode, and the sqlmessage condition does not occur.

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* when the condition occurs. Valid actions include:

CONTINUE

No action is taken. 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. If the program is connected to a database when the condition occurs, the program disconnects from the database without committing pending updates. In response to an error or a message statement inside a database procedure, STOP terminates the database procedure. There is no way to determine which procedure statements have been executed when the database procedure is terminated in this way. 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). When specified as the response to an error or a message statement inside a database procedure, GOTO terminates the procedure when the action is performed. You cannot determine which database procedure statements have been executed when the procedure has been terminated in this way.

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 the CALL action is taken in response to an error or a message statement inside a database procedure, another Ingres tool cannot be called. The called procedure cannot issue any database statements, because a database procedure continues to execute when a call action is specified. The called procedure can issue any forms statements that do not access the database. Do not issue form statements that access the database; for example, do not enter a display loop containing a SELECT statement, or issue the FORMINIT statement.

When the message statement is issued from a database procedure that executes as a result of a rule firing, the DBMS Server displays the message text and continues program execution, even if a WHENEVER SQLMESSAGE statement is in effect. All messages are displayed and are not returned through the SQLCA.

If your program does not include an SQLCA (and therefore no WHENEVER statements), the DBMS Server displays all errors. If your program includes an SQLCA, the DBMS Server 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 the DBMS Server to display errors and messages, set II_EMBED_SET to SQLPRINT. For information about II_EMBED_SET, see the *System Administrator Guide*.

The program's condition is automatically checked after each embedded SQL database statement or each database procedure statement. If one of the conditions has become true, the *action* specified for that condition is taken. If the *action* is GOTO, 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 SQL 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 does not allow empty IF blocks), the WHENEVER statement must not be the only statement in the block.

To avoid infinite loops, the first statement in an error handling routine must be a WHENEVER...CONTINUE that turns off error handling for the condition that caused the error. For example:

```
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 loops infinitely between the DROP statement and the label, error_label.

Embedded Usage

Host language variables cannot be used in an embedded WHENEVER statement.

Permissions

This statement is available to all users.

Locking

In general, the WHENEVER statement has no impact. However, if the specified action is stop, any locks held are dropped because this action terminates execution of the program.

Related Statements

Create Procedure (see page 366)

Examples: Whenever

The following examples describe how to enable your application to handle error and exception conditions arising from embedded SQL database statements.

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;
   ```

6. Ignore all messages originating in a database procedure. This is useful when you want to suppress informational messages when providing a production application.

   ```
   exec sql whenever sqlmessage continue;
       ...
       exec sql execute procedure proc1;
   ```

While - Endwhile

Valid in: ESQL

The WHILE - ENDWHILE statement repeats a series of statements while a specified condition is true.

Syntax

The WHILE - ENDWHILE statement has the following format:

```
[label:]        WHILE boolean_expr DO
                                statement; {statement;}
                ENDWHILE;
```

Description

TheWHILE - ENDWHILE statement defines a program loop. This statement can only be used inside a database procedure.

The Boolean expression (*boolean_expr*) must evaluate to true or false. A Boolean expression can include comparison operators (' =',' >', and so on) and these logical operators:

- AND

- OR

- NOT

The statement list can include any series of legal database procedure statements, including another WHILE statement.

As long as the condition represented by the Boolean expression remains true, the series of statements between DO and ENDWHILE is executed. The condition is tested only at the start of each loop. If values change inside the body of the loop so as to make the condition false, execution continues for the current iteration of the loop, unless an ENDLOOP statement is encountered.

The ENDLOOP statement terminates a WHILE loop. When ENDLOOP is encountered, the loop is immediately closed, and execution continues with the first statement following ENDWHILE. For example:

```
WHILE condition_1 DO
        statement_list_1
        IF condition_2 THEN
                endloop;
        ENDIF;
        statement_list_2
ENDWHILE;
```

In this case, if *condition_2* is true, *statement_list_2* is not executed in that pass through the loop, and the entire loop is closed. Execution resumes at the statement following the ENDWHILE statement.

A WHILE statement can be labeled. The label enables the ENDLOOP statement to break out of a nested series of WHILE statements to a specified level. The label precedes WHILE and is specified by a unique alphanumeric identifier followed by a colon, as in the following:
```
A: WHILE...
```

The label must be a legal object name. For details, see the chapter "Introducing SQL." The ENDLOOP statement uses the label to indicate which level of nesting to break out of. If no label is specified after ENDLOOP, only the innermost loop currently active is closed.

The following example illustrates the use of labels in nested WHILE statements:

```
label_1;            WHILE condition_1 DO
                      statement_list_1
label_2:                WHILE condition_2 DO
                          statement_list_2
                          IF condition_3 THEN
                                  ENDLOOP label_1;
                          ELSEIF condition_4 THEN
                                  ENDLOOP label_2;
                          ENDIF;
                          statement_list_3
                        ENDWHILE;
                        statement_list_4
                    ENDWHILE;
```

In this example, there are two possible breaks out of the inner loop. If *condition_3* is true, both loops are closed, and control resumes at the statement following the outer loop. If *condition_3* is false but *condition_4* is true, the inner loop is exited and control resumes at *statement_list_4*.

If an error occurs during the evaluation of a WHILE statement, the database procedure terminates and control returns to the calling application.

Permissions

Anyone with permission to create procedures can use this statement.

Example: While - Endwhile

In the following WHILE - ENDWHILE statement example, this database procedure, delete_n_rows, accepts as input a base number and a number of rows. The specified rows are deleted from the table "tab," starting from the base number. If an error occurs, the loop terminates:

```
create procedure delete_n_rows
      (base integer, n integer) as
declare
limit integer;
      err integer;
begin
      limit = base + n;
      err = 0;
      while (base < limit) do
            delete from tab where val = :base;
            if iierrornumber > 0 then
                  err = 1;
                  endloop;
            endif;
            base = base + 1;
      endwhile;
      return :err;
end
```

Appendix A: Keywords

This appendix lists Ingres keywords and indicates the contexts in which they are reserved, enabling you to avoid assigning object names that conflict with reserved words. In addition to the explicit keywords, all identifiers starting with the letters ii are reserved for Ingres system catalogs.

The keywords in these lists 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 Word Keywords

Reserved single word keywords are listed in the following table.

| Keyword | SQL | | | QUEL | | |
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
abort	*	*	*	*	*	*
activate		*			*	
add	*	*	*			
addform		*			*	

		SQL			QUEL	
Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
after			*			*
all	*	*		*	*	
alter	*		*			
and	*	*		*	*	
any	*	*	*	*	*	
append				*	*	*
array			*			
as	*	*		*	*	*
asc	*		*			
asymmetric	*	*				
at	*	*	*	*	*	*
authorization	*	*				
avg	*	*	*	*	*	
avgu		*		*	*	
before			*			*
begin	*	*	*	*		*
between	*	*	*			
breakdisplay		*			*	
by	*	*		*	*	*
byref	*		*			*
cache	*	*	*			
call		*	*		*	*
callframe			*			*
callproc	*		*			*
cascade	*	*				
case	*	*	*			
cast	*					
check	*	*	*			
clear		*	*		*	*

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
clearrow		*	*		*	*
close	*	*		*		
coalesce	*					
collate	*	*				
column	*	*	*		*	
command		*			*	
comment			*			
commit	*	*				
commited	*	*	*			
connect		*				
constraint	*	*	*			
continue	*	*				
copy	*	*	*	*	*	*
copy_from	*					
copy_into	*					
count	*	*	*	*	*	
countu		*		*	*	
create	*	*	*	*	*	*
current	*	*				
current_user	*					
currval	*	*	*			
cursor	*	*				
cycle	*	*	*			
datahandler		*				
declare	*	*	*			*
default	*	*	*			*
define	*			*		*
delete	*	*	*	*	*	*
deleterow		*	*		*	*

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
desc			*			
describe	*	*				
descriptor		*				
destroy				*	*	*
direct			*			*
disable			*			
disconnect		*				
display		*	*		*	*
distinct	*	*	*			
distribute				*		
do	*		*			*
down		*			*	
drop	*	*	*			
else	*		*			*
elseif	*		*			*
enable			*			
end	*	*	*	*	*	*
end-exec		*				
enddata		*			*	
enddisplay		*			*	
endfor	*	*	*			
endforms		*			*	
endif	*		*			*
endloop	*	*	*		*	*
endrepeat	*	*	*			
endretrieve					*	
endselect		*				
endwhile	*		*			*
escape	*	*				

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
except	*					
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			*			*

Keyword	ISQL	SQL ESQL	4GL	IQUEL	QUEL EQUEL	4GL
help_frs		*			*	
helpfile		*	*		*	*
identified		*	*			
if	*		*			*
iimessage		*			*	
iiprintf		*			*	
iiprompt		*			*	
iistatement					*	
immediate	*	*	*			*
import	*					
in	*	*	*	*	*	
include		*		*		
increment	*	*	*			
index	*	*	*	*	*	*
indicator		*				
ingres					*	
initial_user	*	*				
initialize		*	*		*	*
inittable		*	*		*	*
inner		*	*			
inquire_equel					*	
inquire_forms			*			*
inquire_frs		*			*	
inquire_ingres		*	*		*	*
inquire_sql		*	*			
insert	*	*	*			
insertrow		*	*		*	*
integrity	*	*		*		*
intersect	*					

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
into	*	*	*	*	*	*
is	*	*	*	*	*	*
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	*	*	*			

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
noorder	*	*	*			
not	*	*		*	*	
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		*			*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
putrow		*			*	
qualification			*			*
raise	*		*			
range				*	*	*
rawpct	*	*	*			
read	*					
redisplay		*	*		*	*
references	*	*	*			
referencing	*		*			
register	*	*	*	*	*	*
relocate	*	*	*	*	*	*
remove	*	*	*		*	*
rename				*		
repeat	*	*	*		*	*
repeatable	*	*	*			
repeated		*	*			
replace				*	*	*
replicate				*		
restart	*	*	*			
restrict	*	*				
result		*				
resume		*	*		*	*
retrieve				*	*	*
return	*		*			*
revoke	*	*	*			
right		*	*			
role		*	*			
rollback	*	*	*			
row	*	*	*			

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
rows	*	*				
run			*			*
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	*	*				

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
sum	*	*	*	*	*	
sumu		*		*	*	
symmetric	*	*				
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	*	*	*	*	*	*

	SQL			QUEL		
Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
while	*					*
with	*	*	*	*	*	*
work	*		*			
write	*	*	*			

Reserved Multi Word Keywords

Reserved multi-word keywords are listed in the following table.

	SQL			QUEL		
Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
add privileges			*			
after field			*			*
after 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 declare		*				
begin exclude		*				
begin transaction	*	*	*	*	*	*

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
by group			*			
by role	*					
by user	*		*			
call on			*			
call procedure			*			
class of			*			
clear array		*				
close cursor		*		*	*	
comment on	*	*	*			
connect to			*			
copy table			*			
create dbevent	*	*	*			
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			*			

	SQL			QUEL		
Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
current value	*	*	*			
define cursor				*		
declare cursor					*	
define integrity				*	*	*
define link					*	
define location				*		
define permit				*	*	*
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	*	*	*			

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
drop permit	*		*			
drop procedure			*			
drop profile	*					
drop role	*	*	*			
drop rule	*	*	*			
drop security_alarm	*	*	*			
drop sequence	*	*	*			
drop synonym	*	*	*			
drop user	*	*	*			
drop view	*		*			
each row		*				
each statement		*				
enable security_audit	*	*	*			
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	*					

| | SQL | | | QUEL | | |
Keyword	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
get data		*				
get dbevent		*	*			
get global		*				
global temporary			*			
help all		*				
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	*		*			

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
on location	*		*			
on procedure	*					
on sequence	*					
only where				*		
open cursor		*		*	*	
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			*			*
resume nextfield			*			
resume previousfield			*			
retrieve cursor		*		*	*	

	SQL			QUEL		
Keyword	**ISQL**	**ESQL**	**4GL**	**IQUEL**	**EQUEL**	**4GL**
right join	*		*			
right outer	*					
run submenu			*			*
send userevent			*			
session group			*			
session role			*			
session user			*			
set aggregate	*			*		
set attribute		*				
set autocommit	*			*		
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	*			*		

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
set maxidle	*					
set maxio	*			*		
set maxpage	*			*		
set maxquery	*			*		
set maxrow	*			*		
set money_format	*			*		
set money_prec	*			*		
set noflatten	*					
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	*			*		

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
set noprintrules	*					
set noqep	*			*		
set norules	*					
set nosql				*		
set nostatistics	*			*		
set notrace	*			*		
set nounicode_substitution	*					
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 transaction	*					
set unicode_substitution	*					
set update_rowcount	*					
set work	*					
system user			*			

Keyword	SQL			QUEL		
	ISQL	ESQL	4GL	IQUEL	EQUEL	4GL
to group	*		*			
to role	*		*			
to user	*	*				
user authorization			*			
with null				*		
with short_remark	*					

Partition Keywords

The following are words treated as keywords in a partition definition context. They are keywords only in the context of a WITH PARTITION= clause.

Keyword	ISQL	SQL ESQL 4GL	QUEL		
			IQUEL	EQUEL	4GL
automatic	*				
hash	*				
list	*				
null	*				
on	*				
partition	*				
range	*				
to	*				
values	*				
with	*				

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

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 B: Terminal Monitors

Terminal monitors interactively enter, edit, and execute individual queries or files containing queries. Terminal monitors also allow operating system level commands to be executed.

Windows: Ingres character-based utilities like Terminal Monitor will display characters correctly only if run under the supplied Ingres command prompt, which has the correct code page and font settings.

UNIX: Ingres character-based utilities like Terminal Monitor will display characters correctly only if the console window on which they are run has the correct code page set, which must match the character set value set in II_CHARSETxx for the database.

Terminal Monitor Releases

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.

How Terminal Monitors Are Accessed

To access terminal monitors, 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.

The following is a list of available flags and their descriptions:

-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 only available on Linux platforms.

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:

\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.

Windows and UNIX:

```
\e or \ed or \edit or \editor [filename]
```

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

\g or \go

Processes the current query. The contents of the buffer are transmitted to the DBMS Server and run.

\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.

UNIX:

```
\s or \sh or \shell
```

Escape to the UNIX shell (command line interpreter). Pressing Ctrl+D exits the shell and returns to the Terminal Monitor.

VMS:

```
\s or \sh or \shell
```

Escape to the command line interpreter to execute VMS commands. The VAX command line interpreter (DCL) is initiated. Subsequently, typing the logout command exits DCL and returns 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]

Write/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 in no way impedes the terminal output of your session.

\suppress and \nosupress

Tells the Terminal Monitor to suppress/not suppress the printing of the resulting data that is returned from the query

\bell and \nobell

Tells the Terminal Monitor to include (\bell) or not to include (\nobell) a bell (that is, Ctrl+G) with the continue or go prompt. The default is \nobell.

\[no]continue

Tells the Terminal Monitor to continue statement processing on error or not to continue (nocontinue). 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. For information about II_TM_ON_ERROR, see the Database 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 C: SQL Statements from Earlier Releases of Ingres

This appendix lists SQL statements from earlier releases of Ingres that are no longer necessary. While these statements are still supported, they are considered obsolete.

Substitute Statements

The following new statements can be substituted for the old statements described in this appendix:

Old Statement	New Statement
abort	rollback
begin transaction	no equivalent statement
create permit	grant
drop permit	revoke
end transaction	commit
inquire_ingres	inquire_sql
relocate	modify
set_ingres	set_sql

Abort Statement

The abort statement, when used without the optional to *savepoint_name* clause, allows the user to terminate an in-progress multi-statement transaction (MST) at any time before the transaction is committed with an explicit end transaction statement. The abort statement causes all database changes effected by the MST to be undone and terminates the MST.

You also have the option of aborting part of a transaction to a pre-declared savepoint. The abort statement with the optional to *savepoint_name* clause undoes the database effects of all statements in the MST that follow the declaration of the named savepoint. Following an abort to *savepoint_name,* the MST remains in progress, and new statements can be added to the MST in the normal fashion. Repeated aborts can be executed to the same savepoint.

This statement has the following format:

```
ABORT [TO savepoint_name]
```

Note: When executing an abort to a given savepoint, all savepoints declared after the named savepoint are nullified.

Begin Transaction (see page 759)

End Transaction (see page 763)

Examples: Abort

The following example begins a transaction, executes some SQL statements, and aborts the transaction before committing the database changes:

```
begin transaction;
insert into emp (name, sal, bdate)
    values ('Jones,Bill', 100000, 1814);
insert into emp (name, sal, bdate)
    values ('Jones,Bill', 100000, 1714);
abort; \g
/* undoes both inserts; table is unchanged */
```

The following example begins a transaction, establishes savepoints, and does a partial abort of the MST:

```
begin transaction;
insert into emp (name, sal, bdate)
    values ('Jones,Bill', 100000, 1945);
savepoint setone;
insert into emp (name, sal, bdate)
    values ('Smith,Stan', 50000, 1911);
savepoint 2; \g
/* undoes second insert; deactivates savepoint 2 */
abort to setone; \g
insert into emp (name, sal, bdate)
    values ('Smith,Stan', 50000, 1948);
abort to setone; \g
end transaction; \g
/* only the first insert is committed */
```

Begin Transaction Statement

The begin transaction SQL statement declares the beginning of a multi-statement transaction (MST). MSTs contain one or more SQL statements to be processed as a single, indivisible database action. Many SQL statements are allowed within an MST; others, however, are not. The phrase, within an MST, is strictly defined to indicate statements appearing between an initial begin transaction statement and a final end transaction statement.

After beginning an MST with begin transaction, the MST can be terminated by either committing or aborting the transaction. Use the end transaction statement to commit the MST and the abort statement to undo the MST. Ingres automatically aborts the MST in cases of deadlock.

Note: Set lockmode is not permitted within an MST. Begin transaction and end transaction are not allowed to be nested within an MST.

This statement has the following format:

```
BEGIN TRANSACTION
```

Transactions (see page 219)

Examples: Begin Transaction

The following example begins an MST, executes SQL statements, and commits the updates to the database:

```
begin transaction;
insert into emp (name, sal, bdate)
     values ('Jones,Bill', 10000, 1914);
insert into emp (name, sal, bdate)
     values ('Smith,Stan', 20000, 1948);
end transaction; \g
/* commits both inserts to table */
```

The following example begins an MST, executes SQL statement, and aborts the transaction, thus canceling the updates:

```
begin transaction;
insert into emp (name, sal, bdate)
     values ('Jones,Bill', 1000000, 1814);
insert into emp (name, sal, bdate)
     values ('Wrong,Tony', 150, 2021);
abort; \g
/* undoes both inserts; table is unchanged */
```

Create Permit Statement

The Create Permit SQL statement defines permissions for a table or view.

This statement has the following format:

```
CREATE PERMIT oplist ON | OF | TO table_name[corr_name]

[(column_name {, column_name})] TO user_name
```

optlist

Specifies a comma-separated list of operations. These operations include the following:

- SELECT

- UPDATE

- DELETE

- INSERT

- ALL

column_name

Can only be specified when the *oplist* value is UPDATE.

user_name

Specifies the login name of a user or the word ALL (meaning all users in this argument).

By default, the owner of the table has permission to perform ALL operations on the table.

Note: Permits created on a table must be grant-compatible to allow a user to access the table. Grant-compatible means that the text of the CREATE PERMIT statement can be expressed as a GRANT statement without any loss of information. The syntax of CREATE PERMIT presented above is grant-compatible. However, permits created with the CREATE PERMIT syntax documented prior to Release 6 may not be grant-compatible.

Example: Create Permit

The following example allows a user, Mildred, to select data from the employee table:

```
create permit select of employee to mildred;
```

Drop Permit Statement

Permission: You must own a table, view, database event, or procedure to drop a permission on it.

The Drop Permit statement removes permissions on tables, views, database events, and procedures.

This statement has the following format:

For tables and views:

```
[EXEC SQL] DROP PERMIT ON table_name

ALL | integer {, integer};
```

For procedures:

```
[EXEC SQL] DROP PERMIT ON PROCEDURE proc_name

ALL | integer {, integer};
```

For events:

```
[EXEC SQL] DROP PERMIT ON DBEVENT event_name

ALL | integer {, integer};
```

If the keyword ALL is used, Ingres removes all permissions defined on the specified table, view, database event, or procedure. To remove individual permissions, use the integer list. To obtain the integer values associated with specific permissions, use the HELP PERMIT statement.

Note: Permits cannot be dropped if there are dependent objects (views or database procedures) or permits. In this case, REVOKE...CASCADE must be used.

Embedded Usage: Drop Permit

In an embedded DROP PERMIT statement, no portion of the syntax can be replaced with host language variables.

Locking

The Drop Permit SQL statement takes an exclusive lock on the base table and on pages in the iipermits system catalog.

Example: Drop Permit

The following example drops all permissions on job:

```
drop permit on job all;
```

In an application, drop the second permission on procedure addemp:

```
exec sql drop permit on procedure addemp 2;
```

End Transaction Statement

The End Transaction SQL statement terminates an in-progress multi-statement transaction (MST) and commit its updates, if any, to the database. This statement causes all database updates effected by the MST to become available to other user transactions. After committing an MST with END TRANSACTION, the MST is terminated, the MST can no longer be aborted, and all its savepoints are nullified.

This statement has the following format:

```
END TRANSACTION
```

Example: End Transaction

The following example begins an MST, executes some SQL statements, and commits the updates to the database:

```
begin transaction;
insert into emp (name, sal, bdate)
    values ('Jones,Bill', 10000, 1914);
insert into emp (name, sal, bdate)
    values ('Smith,Stan', 20000, 1948);
end transaction; \g/* commits new rows to table */
```

Inquire_ingres Statement

The Inquire_ingres SQL statement returns diagnostic information about the last database statement that was executed. Inquire_ingres and inquire_sql are synonymous.

This statement has the following format:

```
EXEC SQL INQUIRE_INGRES (variable = object {, variable = object})
```

Relocate Statement

The Relocate SQL statement is used to relocate tables. This statement moves a table from its current location to the *area* corresponding to the specified *location_name*. All indexes, views, and protections for the table remain in force and valid regardless of the location of the table.

Note: The RELOCATE statement must be used when the current disk of the table becomes too full.

This statement has the following format:

RELOCATE *table_name* TO *location_name*

location_name

Refers to the area in which the new table is created. The location name must be defined on the system, and the database must have been extended to the corresponding area.

Note: *Table_name* and *location_name* must be string constants if this statement is used in an embedded program. Host language variables cannot be used to represent either.

Example: Relocate

The following example relocates the employee table to the area defined by the remote_loc *location_name*:

```
relocate employee to remote_loc;
```

Set_ingres Statement

Permission required: All users.

The Set_ingres SQL statement switches sessions in a multiple session application, specify which type of DBMS server error is returned to an application, change the default behavior when a connection error is experienced, and set trace functions.

Set_ingres can be used to override II_EMBED_SET if it is defined. For example, you can issue a set_ingres statement with the errortype object to override the error type default defined by II_EMBED_SET. Similarly, setting printqry or printgca can override the defaults defined by II_EMBED_SET. For more information about II_EMBED_SET, see the *Database Administrator Guide*.

This statement has the following format:

```
EXEC SQL SET_INGRES (object = value {, object = value})
```

Note: This statement must be terminated according to the rules of your host language.

Appendix D: SQLSTATE Values and Generic Error Codes

This appendix lists Ingres generic error codes, and maps generic error codes to SQLSTATE values.

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

SQLSTATE	Ingres Only	Description
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
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

SQLSTATE	Ingres Only	Description
22009		Invalid time zone displacement value
22011		Substring error
22012		Division by zero
22015		Interval field overflow
22018		Invalid character value for cast
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

SQLSTATE	Ingres Only	Description
2C000		Invalid character set name
2D000		Invalid transaction termination
2E000		Invalid connection name
33000		Invalid SQL descriptor name
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

SQLSTATE	Ingres Only	Description
42507	*	Reserved identifier
44000		With check option violation
50000	*	Miscellaneous Ingres-specific errors
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

Appendix E: ANSI Compliance Settings

This appendix lists the settings required to operate in compliance with ANSI/ISO Entry SQL-92 and the corresponding Ingres defaults.

To determine the setting in effect, use dbmsinfo. For details about dbmsinfo, see Status Information (see page 233).

Case sensitivity for delimited identifiers is specified when the database is created. For details about createdb, see the *Command Reference Guide* and the *System Administrator Guide*.

Query flattening and default cursor mode options are specified when the DBMS Server is configured and started.

Configuration-By-Forms Settings

If Configuration-By-Forms (cbf) is used to configure the installation, select the Preferences menu item and set the ISO_ENTRY_SQL-92 parameter to ON. This setting ensures ANSI/ISO-compliant behavior for the following areas:

- Case sensitivity of identifiers (all types)
- Default cursor mode
- Query flattening

Case Sensitivity for Identifiers

The ANSI Entry SQL-92 standard specifies how case is handled for identifiers (names of tables, for example). Identifiers can be quoted or unquoted (regular or delimited (see page 39)).

Regular Identifiers

ANSI/ISO Entry SQL-92

Regular identifiers are treated as upper case:

```
dbsminfo('db_name_case') returns UPPER.
```

Ingres Default

Regular identifiers are treated as lower case:

```
dbsminfo('db_name_case') returns LOWER.
```

Delimited Identifiers

ANSI/ISO Entry SQL-92

Delimited identifiers are case-sensitive:

`dbsminfo('db_delimited_case')` returns `MIXED`.

Ingres Default

Delimited identifiers are not case-sensitive:

`dbsminfo('db_delimited_case')` returns `LOWER`.

User Names

ANSI/ISO Entry SQL-92

User names are treated as upper case:

`dbsminfo('db_real_user_case')` returns `UPPER`.

Ingres Default

User names are treated as lower case:

`dbsminfo('db_real_user_case')` returns `LOWER`.

Default Cursor Mode

The ANSI Entry SQL-92 standard specifies the default mode for cursors. For details, see Data Manipulation with Cursors (see page 173).

ANSI/ISO Entry SQL-92

Direct mode:

`dbsminfo('db_direct_update')` returns `Y`

`dbsminfo('db_deferred_update')` returns `N`

Ingres Default

Deferred mode:

`dbsminfo('db_direct_update')` returns `N`

`dbsminfo('db_deferred_update')` returns `Y`

Query Flattening

The ANSI Entry SQL-92 standard specifies that query flattening is not performed. By default, Ingres uses query flattening to improve the performance of queries.

ANSI/ISO Entry SQL-92

No Flattening:

```
dbsminfo('flatten_singleton') returns N

dbsminfo('flatten_aggregate') returns N
```

Ingres Default

Flattening used to optimize queries, including queries involving aggregate subselects or singleton subselects:

```
dbsminfo('flatten_singleton') returns Y

dbsminfo('flatten_aggregate') returns Y
```

Connection Flags

Connection flags (also referred to as *SQL Option flags*) are specified on the command line when invoking an Ingres operating system level command or user-written application that connects to a database. For details about other connection flags, see the description of the sql command in the *Command Reference Guide* and in the *System Administrator Guide*.

The ANSI standard specifies DBMS behavior when string truncation and numeric overflow occur. To specify handling of these conditions, use the connection flags described in the following sections.

-string_truncation Connection Flag

The -string_truncation connection flag specifies how the DBMS handles attempts to write a character string to a table column that is too short to contain it (in an Insert, Update, Copy, or Create Table...as Select SQL statement). For details about this connection flag, see String Truncation Errors (see page 96).

ANSI/ISO Entry SQL-92

Specify -string_truncation=fail. If string truncation occurs, the statement that attempted to write the string fails and an error is returned.

Ingres Default

Omit flag or specify -string_truncation=ignore. By default, the string is truncated to fit into the column, the statement succeeds, and no error is iss.

-numeric_overflow Connection Flag

The -numeric_overflow connection flag specifies how the DBMS handles attempts to write numeric values that are outside the range of values supported by the data type of the receiving column (overflow and underflow).

The -numeric_overflow connection flag specifies behavior for integer, decimal, and floating point columns. For details about this connection flag, see Specifying Error Handling for Arithmetic Errors (see page 103).

ANSI/ISO Entry SQL-92 and Ingres Default

Specify -numeric_overflow=fail (or omit this flag). If numeric overflow or underflow occurs, the statement that attempted to write the value fails and an error is issued. (This is the Ingres default behavior.)

ESQL Preprocessor Flags

To specify ANSI-compliant behavior when creating embedded SQL programs, use the following flags when invoking the Ingres embedded SQL preprocessor. For details about the preprocessor, see the *Embedded SQL Companion Guide*.

-wsql ESQL Preprocessor Flag

The -wsql ESQL preprocessor flag directs the preprocessor to issue warnings when it detects SQL statements that do not comply with ANSI/ISO Entry SQL-92.wsql flag ANSI/ISO standard:handling non-compliant SQL statements flags, ESQL preprocessor:wsql

ANSI/ISO Entry SQL-92

Specify -wsql=entry_sql92.

Ingres Default

By default, the preprocessor does not issue warnings when it detects non-compliant SQL statements.

-blank_pad ESQL Preprocessor Flag

The -blank_pad ESQL preprocessor flag specifies how values are padded with blanks when selecting values from character columns into host string variables. This flag has no effect in host languages that do not support variable-length character data (for example, Fortran).

ANSI/ISO Entry SQL-92

Specify -blank_pad. The host variable is padded with blanks to its declared size. For example, if you select a value from a column defined as char(5) into a 10-byte character variable, the host variable is padded with blanks to its full length of 10 bytes.

Ingres Default

By default, the receiving variable is padded with blanks to the width of the column from which data is selected. This flag affects the results of the following SQL statements:

- execute immediate
- execute procedure
- fetch
- get data
- inquire_sql
- select

-sqlcode

The -sqlcode ESQL preprocessor flag incorporates the declaration required for the ANSI standard SQLCODE status variable. For details about SQLCODE, see SQLCODE and SQLSTATE in the chapter "Working with Transactions and Error Handling" and the ANSI standard. SQLCODE is a deprecated ANSI feature - SQLSTATE is the recommended status variable. For details, see the *Embedded SQL Companion Guide*.

ANSI/ISO Entry SQL-92

Specify -sqlcode if your program declares SQLCODE outside of an SQL declare section. This flag is optional if your source code declares SQLCODE in a declare section. To see SQLCODE, your source code must also contain an include sqlca statement. (Some host languages require the -sqlcode flag regardless of where SQLCODE is declared. For details, see the *Embedded SQL Companion Guide*.)

Ingres Default

There is no default. Ingres provides other proprietary methods for checking program status and error conditions. For details, see the chapter "Working with Transactions and Error Handling." If your source code declares a variable named SQLCODE that is not intended to be used for the ANSI status variable, specify the - nosqlcode flag to prevent Ingres from writing ANSI status information into the variable.

-check_eos (C only)

The -check_eos ESQL preprocessor flag directs the ESQL preprocessor to include code that ensures that all char strings inserted into a database are terminated with a null character (\0). By default, no checking is performed. Checking is performed only for char strings declared as arrays, and is not performed for strings declared as string pointers.

ANSI/ISO Entry SQL-92

Specify -check_eos. If your ESQL/C application attempts to insert a string that is not null-terminated, the DBMS returns an error (SQLSTATE 22024).

Ingres Default

No checking is performed.

Index

having (clause) • 155, 641
heap (storage structure) • 579, 675
heapsort (storage structure) • 579, 675
hex (function) • 108
host language • 168
hostlanguage • 34

I

i (terminal monitor command) • 751
ifnull (function) • 142
II_DATE_FORMAT • 68
in (predicate) • 153
include (statement)
    described • 166
    SQLDA • 187
include (terminal monitor command) • 751
indexes
    building parallel • 359
    compression • 358
    unique • 358
indicator variables
    character data retrieval • 172
    ESQL • 168
ingresdate (data type) • 65
    arithmetic operations • 104
    assignment • 98
    data type return code • 188
    date_part (function) • 129
    date_trunc (function) • 129
    display formats • 71
    formats (list) • 66
    functions • 129
    input formats • 65
    interval (function) • 129
    intervals • 69
    now constant • 87
initial_user constant • 87
inquire_ingres (statement) • 763
inquire_sql (statement)
    database events • 281
    described • 241
    error checking • 254, 266
insert (statement) • 207
int1 (function) • 108
int2 (function) • 108
int4 (function) • 108
integer
    data types • 57
    literals • 83

range • 57
integer (data type)
    data type return code • 188
integrity
    constraints and nulls • 90
    constraints and rules • 268
interrupts • 223
interval (data type) • 63
interval (function) • 129
isam (storage structure) • 579, 675
ISO format (dates) • 66
ISQL
    overview • 34

J

journaling • 674
    enabling/disabling • 430, 674
journals, files • 292, 363

K

keyboard Ctrl key • 223, 755
keywords
    single • 719
KnowledgeManagementExtension
    described • 37

L

labels, Embedded SQL • 162
languages, host • 168
languages,host • 34
left (function) • 118
length (function) • 118
like (predicate) • 151
limits
    ANSI identifiers • 40
    connection name length • 270
    cursor name length • 175
    data handlers • 208
    database event text length • 279
    database name length • 38
    float data type • 79
    integer data • 57
    length of connection name • 325
    length of long varchar columns • 51
    logical operators in queries • 93
    long varchar length • 209
    nested rules • 268
    number of columns in rules • 384
    number of columns in tables • 402