IBM Informix Guide to SQL

Syntax

IBM Informix Extended Parallel Server, Version 8.4
IBM Informix Dynamic Server, Version 9.4

March 2003
Part Nos. CT1SQNA (Volume 1) and CT1SRNA (Volume 2)

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

Table of Contents

Introduction

Chapter 1 Overview of SQL Syntax

Chapter 2 SQL Statements

Chapter 3 SPL Statements

Chapter 4 Segments

Introduction

In This Introduction

This introduction provides an overview of the information in this manual and describes the documentation conventions that it uses.

About This Manual

This manual describes the syntax of the structured query language (SQL) and Stored Procedure Language (SPL) statements for Version 9.4 of IBM Informix Dynamic Server and Version 8.4 of IBM Informix Extended Parallel Server.

This manual is a companion volume to the *IBM Informix Guide to SQL: Reference*, the *IBM Informix Guide to SQL: Tutorial*, and the *IBM Informix Database Design and Implementation Guide*. The *IBM Informix Guide to SQL: Reference* provides reference information about the system catalog, the built-in SQL data types, and environment variables that can affect SQL statements. The *IBM Informix Guide to SQL: Tutorial* shows how to use basic and advanced SQL and SPL routines to access and manipulate the data in your databases. The *IBM Informix Database Design and Implementation Guide* shows how to use SQL to implement and manage relational databases.

Types of Users

This manual is written for the following users:

- Database users
- Database administrators
- Database-application programmers

This manual assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming

If you have limited experience with relational databases, SQL, or your operating system, refer to the *Getting Started Guide* for your database server for a list of supplementary titles.

Software Dependencies

This manual assumes that you are using one of the following database servers:

- IBM Informix Extended Parallel Server, Version 8.4
- IBM Informix Dynamic Server, Version 9.4

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a Global Language Support (GLS) locale.

This manual assumes that you use the U.S. 8859-1 English locale as the default locale. The default is **en_us.8859-1** (ISO 8859-1) on UNIX platforms or **en_us.1252** (Microsoft 1252) for Windows environments. These locales support U.S. English format conventions for dates, times, and currency, and also support the ISO 8859-1 or Microsoft 1252 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use non-ASCII characters in your data or in SQL identifiers, or if you want to conform to localized collation rules of character data, you need to specify an appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS Programmer's Manual*.

Demonstration Databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix manuals are based on the **stores_demo** database.
- XPS The **sales_demo** database illustrates a dimensional schema for data-warehousing applications. For conceptual information about dimensional data modeling, see the *IBM Informix Database Design and Implementation Guide*. ♦
- IDS The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines. ♦

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX platforms and in the **%INFORMIXDIR%\bin** directory in Windows environments.

New Features in Dynamic Server, Version 9.4

For a comprehensive list of new database server features, see the *Getting Started Guide*. This section lists new features relevant to this manual. In addition to documenting the new features that are listed in this section, this manual corrects errata that have been identified since the previous edition.

The following list provides information about the new features for IBM Informix Dynamic Server, Version 9.4, that this manual describes.

- The following new SQL statements are documented in Chapter 2:

  ALTER SEQUENCE
  CREATE SEQUENCE
  DROP SEQUENCE
  RENAME SEQUENCE
  DESCRIBE INPUT
  SET COLLATION

- The following SQL statements support new syntax in Version 9.4:

  ALTER FUNCTION
  ALTER PROCEDURE
  CREATE FUNCTION
  CREATE PROCEDURE
  CREATE SYNONYM
  CREATE TRIGGER
  DESCRIBE
  GRANT
  REVOKE
  SELECT
  SET EXPLAIN
  SET Residency

- The SET COLLATION statement can specify a localized collating order for sorting NCHAR and NVARCHAR values that is different from what **DB_LOCALE** specifies. Database objects such as indexes and UDRs that perform sorting operations always use the collating order that was in effect when they were created.
- This release provides data definition language (DDL) statements to support *sequence objects*, from which one or more users can generate unique integers in the INT8 range. The GRANT and REVOKE statements can control privileges on a sequence, and CREATE SYNONYM can declare synonyms for sequence objects. New CURRVAL and NEXTVAL operators can read and increment sequence objects.
- The new DESCRIBE INPUT and DESCRIBE OUTPUT statements can return information about multiple input and output parameters prior to execution of a prepared statement.
- The CREATE TRIGGER statement can create INSTEAD OF Triggers on Views. You can use these to update views that in previous releases did not support UPDATE operations.
- The ORDER BY Clause of the SELECT statement can include column names and expressions that did not appear in the Projection clause.
- The FROM clause of the SELECT statement can include iterator functions that create a virtual table.
- The PDQ feature supports cursors that were declared WITH HOLD.
- The new DESCRIBE INPUT statement supports dynamic queries.

- The LOAD and UNLOAD statements for flat-file I/O can support file sizes larger than the 2 Gigabyte limit of earlier release versions.
- User-defined functions can include multiple OUT parameters.
- SPL functions can declare named return parameters.
- Functional indexes can be based on more than 16 columns. The new limit on index parts is language-dependent, but is greater than 90.
- The LVARCHAR data type can be declared with a new *size* parameter that can be larger than the former upper limit of 2048 bytes.
- The functionality of the SET Residency statement is provided automatically by the database server.

New Features in Extended Parallel Server Version 8.4

This manual describes new features in Version 8.4 of IBM Informix Extended Parallel Server. The features fall into the following areas:

- Performance enhancements
- New SQL functionality
- Version 8.3 features from Dynamic Server, Version 7.30

Performance Enhancements

This manual describes the following performance enhancements to Version 8.4 of IBM Informix Extended Parallel Server:

- Insert cursors with simple large objects
- Coarse-grain index locks
- Updates with subquery in SET clause
- Index on aggregates

New SQL Functionality

This manual describes the following new SQL functionality in Version 8.4 of IBM Informix Extended Parallel Server:

- CASE statement in SPL
- Creating a table with RANGE fragmentation

- DELETE…USING statement to delete rows based on a table join
- Globally detached indexes
- Load and unload simple large objects to external tables
- MIDDLE function
- Referential integrity for globally detached indexes
- TRUNCATE statement

Version 8.4 Features from Version 7.3

This manual describes the following features from Version 7.3 of Dynamic Server in Version 8.4 of IBM Informix Extended Parallel Server:

- Ability to retain update locks
- ALTER FRAGMENT attach with remainders
- ALTER TABLE to add or drop a foreign key constraint
- ALTER TABLE to add, drop, or modify a column
- Constraints on columns other than the fragmentation column
- COUNT function
- DBINFO provides all Version 7.3 information and adds the coserver ID and dbspace name
- Deferred constraints for all constraint types
- Deferred referential-integrity constraints
- Insert from SPL functions
- NVL and DECODE functions
- REPLACE, SUBSTR, LPAD, and RPAD string manipulation functions
- RENAME COLUMN statement
- TO_CHAR and TO_DATE functions for date conversion
- Triggers
- UPDATE SET clause subqueries
- UPPER, LOWER, and INITCAP functions for case-insensitive search
- Memory-resident tables
- Violations table

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other volumes in the documentation set.

Typographical Conventions

This manual uses the following conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All primary elements in a programming language statement (keywords) appear in uppercase letters in a serif font.
italics ***italics*** *`italics`*	Within text, new terms and emphasized words appear in italics. Within syntax and code examples, variable values that you are to specify appear in italics.
boldface ***boldface***	Names of program entities (such as classes, events, and tables), environment variables, file and pathnames, and interface elements (such as icons, menu items, and buttons) appear in boldface.
`monospace` *`monospace`*	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of product- or platform-specific information.
→	This symbol indicates a menu item. For example, "Choose **Tools→Options**" means choose the **Options** item from the **Tools** menu.

Tip: *When you are instructed to "enter" characters or to "execute" a command, immediately press* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify three types of information, as the following table describes. This information always appears in italics.

Icon	Label	Description
⚠	***Warning:***	Identifies paragraphs that contain vital instructions, cautions, or critical information
⇨	***Important:***	Identifies paragraphs that contain significant information about the feature or operation that is being described
💡	***Tip:***	Identifies paragraphs that offer additional details or shortcuts for the functionality that is being described

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-specific, product-specific, or platform-specific information.

Icon	Description
C	Identifies information that is specific to C user-defined routines (UDRs)
DB	Identifies information that is specific to DB-Access
E/C	Identifies information that is specific to IBM Informix ESQL/C

(1 of 2)

Icon	Description
Ext	Identifies information that is specific to external routines, that is, UDRs written in either C or Java language
GLS	Identifies information that relates to the IBM Informix Global Language Support (GLS) feature
IDS	Identifies information or syntax that is specific to IBM Informix Dynamic Server
Java	Identifies information that is specific to UDRs written in Java
UNIX	Identifies information that is specific to the UNIX operating system
Windows	Identifies information that applies to all Windows environments
XPS	Identifies information or syntax that is specific to IBM Informix Extended Parallel Server

(2 of 2)

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears in one or more paragraphs within a section.

Compliance Icons

Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

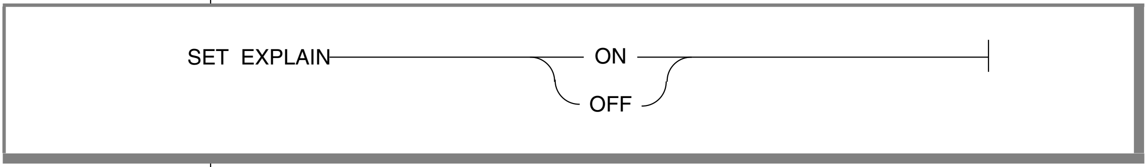
Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database
X/O	Identifies functionality that conforms to X/Open
+	Identifies an Informix extension to ANSI SQL-92 entry-level standard SQL

These icons can apply to an entire section or to one or more paragraphs within a section. If an icon appears next to a section heading, the information that applies to the indicated feature, product, or platform ends at the next heading at the same or higher level. A ♦ symbol indicates the end of feature-, product-, or platform-specific information that appears within one or more paragraphs within a section.

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement or segment, as Figure 1 shows.

Figure 1
Example of a Simple Syntax Diagram



Each syntax diagram begins at the upper-left corner and ends at the upper-right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement.

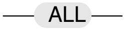
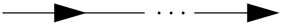
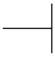
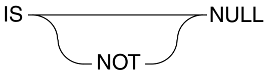
Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. The path always approaches elements from the left and continues to the right, except in the case of separators in loops. For separators in loops, the path approaches counterclockwise. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

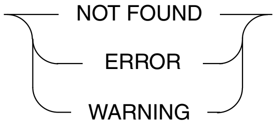
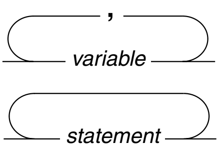
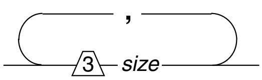
You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE letters is a keyword. You must spell the word exactly as shown; however, you can use either uppercase or lowercase letters.
(. , ; @ + * - /)	Punctuation and other nonalphanumeric characters are literal symbols that you must enter exactly as shown.
' '	[Single quotes are literal symbols that you must enter as shown.]
variable	A word in *italics* represents a value that you must supply. A table immediately following the diagram explains the value.
ADD Clause p. 3-288 ADD Clause	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page. The aspect ratio of a box is not significant.
Back to ADD Clause p. 1-14	A reference in a box in the upper-right corner of a subdiagram refers to the next higher-level diagram of which this subdiagram is a member.

(1 of 3)

Element	Description
E/C	An icon is a warning that this path is valid only for some products, or only under certain conditions. Characters on the icons indicate what products or conditions support the path. These icons might appear in path of a syntax diagram: IDS — Valid only for Dynamic Server. XPS — Valid only for Extended Parallel Server. Ext — Valid only for external routines, that is, UDRs written in C and Java. SPL — Valid only if you are using IBM Informix Stored Procedure Language (SPL) E/C — Valid only for INFORMIX-ESQL/C.
	A shaded option is the default specification. This option is in effect, unless you specify another path.
	Syntax between a pair of arrows is a subdiagram.
	The vertical line at the upper right terminates the syntax diagram.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)

(2 of 3)

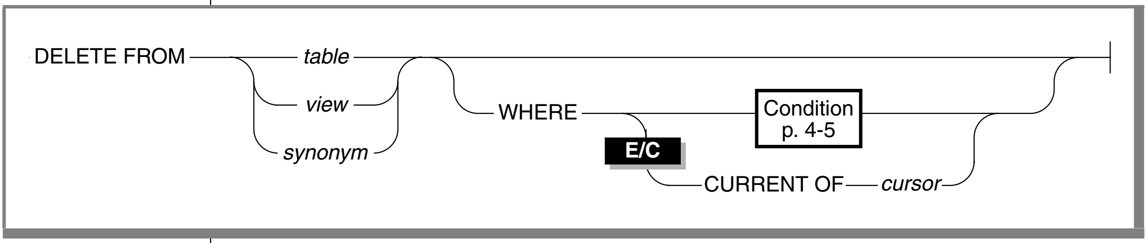
Element	Description
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items. If no symbol appears, a blank space is the separator (except in the diagrams for "Identifier" on page 4-189, "Literal Number" on page 4-216, and "Quoted String" on page 4-243, which allow no separators between characters within loops.
	A gate (/3\ ) on a path indicates that you can use that path only the indicated number of times, even if it is part of a larger loop. You can specify *size* no more than three times within this statement segment.

(3 of 3)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses most of the path elements that the previous table lists.

Figure 2
Example of a Syntax Diagram



Tip: *For purposes of illustrating how to read syntax diagrams, this diagram has been simplified, and does not reflect all of the options of the DELETE statement. See the section "DELETE" on page 2-344" for the complete syntax of DELETE.*

To use this diagram to construct a statement, start at the top left with the keyword DELETE FROM. Then follow the diagram to the right, proceeding through the options that you want.

Figure 2 illustrates the following steps:

1. Type DELETE FROM.
2. You can delete a table, view, or synonym:
   - Type the table name, view name, or synonym, as you desire.
   - You can type WHERE to limit the rows to delete.
   - If you type WHERE and you are using DB-Access or the SQL Editor, you must include the Condition clause to specify a condition to delete. To find the syntax for specifying a condition, go to the "Condition" segment on the specified page.
   - If you are using ESQL/C, you can include either the Condition clause to delete a specific condition or the CURRENT OF *cursor* clause to delete a row from the table.
3. Follow the diagram to the terminator.

   Your DELETE statement is complete.

Example-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single IBM Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using DB-Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement.

Tip: *Ellipses points in program fragments indicate that additional code (that a full application would include) has been omitted to simplify presentation of the concept under discussion. In addition, ellipses symbols never begin or end an example. In most contexts, including literal ellipses symbols in SQL code will produce an error.*

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Additional Documentation

IBM Informix Dynamic Server documentation is provided in a variety of formats:

- **Online manuals.** The IBM Informix Online Documentation site at http://www.ibm.com/software/data/informix/pubs/library/ contains manuals provided for your use. This Web site enables you to print chapters or entire books.
- **Online help.** This facility provides context-sensitive help, an error message reference, language syntax, and more.
- **Documentation notes and release notes.** Documentation notes, which contain additions and corrections to the manuals, and release notes are located in the directory where the product is installed.

  Please examine these files because they contain vital information about application and performance issues.

UNIX

On UNIX platforms in the default locale, the following online files appear in the **$INFORMIXDIR/release/en_us/0333** directory.

Online File	Purpose
ids_sqls_docnotes_9.40.html (*for Dynamic Server*) *or* **xps_sqls_docnotes_8.40.html** (*for Extended Parallel Server*)	The documentation notes file for your version of this manual describes topics that are not covered in the manual or that were modified since publication.
ids_unix_release_notes_9.40.html *and (in plain text format)* **ids_unix_release_notes_9.40.txt** (*for Dynamic Server*) *or* **xps_release_notes_9.40.html** *and (in plain text format)* **xps_release_notes_9.40.txt** (*for Extended Parallel Server*)	The release notes file describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
ids_machine_notes_9.40.txt (*for Dynamic Server*) *or* **xps_machine_notes_8.40.txt** (*for Extended Parallel Server*)	The machine notes file describes any special actions that you must take to configure and use IBM Informix products on your computer. Machine notes are named for the product described.

♦

Windows

The following items appear in the **Informix** folder. To display this folder, choose **Start→Programs→Informix→ Documentation Notes** or **Release Notes** from the task bar.

Program Group Item	Description
Documentation Notes	This item includes additions or corrections to manuals with information about features that might not be covered in the manuals or that have been modified since publication.
Release Notes	This item describes feature differences from earlier versions of IBM Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Machine notes do not apply to Windows platforms. ♦

- **Error message files**. IBM Informix software products provide ASCII files that contain error messages and their corrective actions.

  UNIX

  To read the error messages on UNIX, you can use the **finderr** command to display the error messages online. ♦

  Windows

  To read error messages and corrective actions on Windows, use the **Informix Error Messages** utility. To display this utility, choose **Start→Programs→Informix** from the task bar. ♦

Related Reading

For a list of publications that provide an introduction to database servers and operating-system platforms, refer to your *Getting Started Guide*.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

IBM Welcomes Your Comments

To help us with future versions of our manuals, let us know about any corrections or clarifications that you would find useful. Include the following information:

- The name and version of your manual
- Any comments that you have about the manual
- Your name, address, and phone number

Send electronic mail to us at the following address:

docinf@us.ibm.com

This address is reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact Customer Services.

Chapter

Overview of SQL Syntax

1

In This Chapter

This chapter provides information about how to use the SQL statements, SPL statements, and segments that are discussed in the later chapters of this book. It is organized into the following sections.

How to Enter SQL Statements

Statement descriptions are provided in this manual to help you to enter SQL statements successfully. A statement description includes this information:

- A brief introduction that explains what the statement does
- A syntax diagram that shows how to enter the statement correctly
- A table that explains each input parameter in the syntax diagram
- Rules of usage, typically with examples that illustrate these rules

If a statement can include multiple clauses, this information is provided for each clause.

Most statement descriptions conclude with references to related information in this manual and in other manuals.

The major aids for entering SQL statements include:

- The combination of the syntax diagram and syntax table
- The examples of syntax that appear in the rules of usage
- The references to related information

Using Syntax Diagrams and Syntax Tables

Before you try to use the syntax diagrams in this chapter, it is helpful to read the "Syntax Conventions" on page 12 of the Introduction. This section is the key to understanding the syntax diagrams in the statement descriptions, and explains the elements that can appear in a syntax diagram and the paths that connect the elements to each other. This section also includes an example that illustrates the elements of typical syntax diagrams. The narrative that follows the example diagram shows how to read the diagram in order to enter the statement successfully.

When a syntax diagram includes input specifications, such as identifiers, expressions, filenames. host variables, or other terms, the syntax diagram is followed by a table that describes how to enter the term without generating errors. Each syntax table includes four columns:

- The **Elements** column lists the name of each variable term that appears in the syntax diagram.
- The **Purpose** column briefly describes the term, and identifies the default value, if the term has one.
- The **Restrictions** column summarizes the restrictions on the term, such as acceptable ranges of values. (For some diagrams, restrictions that cannot be tersely summarized appear in the **Usage** notes, rather than in this column.)
- The **Syntax** column points to the SQL segment that gives the detailed syntax for the term. For a few terms, such as the names of host variables or literal characters, no page reference is provided.

The diagrams generally provide an intuitive notation for what is valid in a given SQL statement, but for some statements, dependencies or restrictions among syntax elements are identified only in the text of the Usage section.

Using Examples

To understand the main syntax diagram and subdiagrams for a statement, study the examples of syntax that appear in the rules of usage for each statement. These examples have two purposes:

- To show how to accomplish specific tasks with the statement or its clauses
- To show how to use the syntax of the statement or its clauses in a concrete way

Tip: *An efficient way to understand a syntax diagram is to find an example of the syntax and compare it with the keywords and parameters in the syntax diagram. By mapping the concrete elements of the example to the abstract elements of the syntax diagram, you can understand the syntax diagram and use it more effectively.*

For an explanation of the conventions used in the examples in this manual, see “Example-Code Conventions” on page 16 of the Introduction.

These code examples are program fragments to illustrate valid syntax, rather than complete SQL programs. In some code examples, ellipsis (. . .) symbols indicate that additional code has been omitted. To save space, however, ellipses are not shown at the beginning or end of the program fragments.

Using Related Information

For help in understanding the concepts and terminology in the SQL statement description, check the “Related Information” section at the end of each statement.

This section points to related information in this manual and other manuals that helps you to understand the statement in question. The section provides some or all of the following information:

- The names of related statements that might contain a fuller discussion of topics in this statement
- The titles of other manuals that provide extended discussions of topics in this statement

Tip: *If you do not have extensive knowledge and experience with SQL, the “IBM Informix Guide to SQL: Tutorial” gives you the basic SQL knowledge that you need to understand and use the statement descriptions in this manual.*

How to Enter SQL Comments

You can add comments to clarify the purpose or effect of particular SQL statements. You can also use comment symbols during program development to disable individual statements without deleting them from your source code.

Your comments can help you or others to understand the role of the statement within a program, SPL routine, or command file. The code examples in this manual sometimes include comments that clarify the role of an SQL statement within the code, but your own SQL programs will be easier to read and to maintain if you document them with frequent comments.

The following table shows the SQL comment indicators that you can enter in your code. Here a *Y* in a column signifies that you can use the symbol with the product or with the database type identified in the column heading. An *N* in a column signifies that you cannot use the symbol with the indicated product or database type. (For additional information about a special use of comments, see the section "Optimizer Directives" on page 4-222.)

Comment Symbol	ESQL/C	SPL Routine	DB-Access	ANSI-Compliant Databases	Databases Not ANSI Compliant	Description
double hyphen (--)	Y	Y	Y	Y	Y	The double hyphen precedes a comment within a single line. To comment more than one line, you must put the double hyphen symbols at the beginning of each comment line.
braces ({ })	N	Y	Y	Y	Y	Braces enclose the comment. The { precedes the comment, and the } follows it. Braces can delimit single-line or multiple-line comments, but comments cannot be nested.

Characters within the comment are ignored by the database server.

IDS

The section "Optimizer Directives" on page 4-222 describes a context where information that you specify within comments can influence query plans of the database server, and where (besides comments in these two formats), comments in the style of the C language are also valid. ♦

ANSI

If the product that you are using supports both comment symbols, your choice of a comment symbol depends on requirements for ANSI compliance:

- Double hyphen (--) complies with the ANSI/ISO standard for SQL.
- Braces ({ }) are an Informix extension to the ANSI/ISO standard.

If ANSI compliance is not an issue, your choice of comment symbols is a matter of personal preference. ♦

DB

In DB-Access, you can use either comment symbol when you enter SQL statements with the SQL editor and when you create SQL command files with the SQL editor or a system editor. An SQL command file is an operating-system file that contains one or more SQL statements. Command files are also known as command scripts. For more information about command files, see the discussion of command scripts in the *IBM Informix Guide to SQL: Tutorial*. For information on how to create and modify command files with the SQL editor or a system editor in DB-Access, see the *IBM Informix DB-Access User's Guide*. ♦

SPL

You can use either comment symbol in any line of an SPL routine. See the discussion of how to comment and document an SPL routine in the *IBM Informix Guide to SQL: Tutorial*. ♦

E/C

In ESQL/C, the double hyphen (--) can begin a comment that extends to the end of the same line. For information on language-specific comment symbols in ESQL/C programs, see the *IBM Informix ESQL/C Programmer's Manual*. ♦

Examples of SQL Comment Symbols

These examples illustrate different ways to use the SQL comment indicators.

Examples of the Double-Hyphen Symbol

The next example uses the double hyphen (--) to include a comment after an SQL statement. The comment appears on the same line as the statement.

```
SELECT * FROM customer -- Selects all columns and rows
```

The following example uses the same SQL statement and the same comment as in the preceding example, but places the comment on a line by itself:

```
SELECT * FROM customer
   -- Selects all columns and rows
```

In the following example, the user enters the same SQL statement as in the preceding example but now enters a multiple-line comment:

```
SELECT * FROM customer
   -- Selects all columns and rows
   -- from the customer table
```

DB

SPL

Examples of the Braces Symbols

This example uses braces ({ }) to delimit a comment after an SQL statement. In this example, the comment appears on the same line as the statement.

```
SELECT * FROM customer {Selects all columns and rows}
```

The next example uses the same SQL statement and the same comment as in the preceding example, but the comment appears on a line by itself:

```
SELECT * FROM customer
   {Selects all columns and rows}
```

In the following example, the same SQL statement as in the preceding example is followed by a multiple-line comment:

```
SELECT * FROM customer
   {Selects all columns and rows
    from the customer table}
```

GLS

Non-ASCII Characters in SQL Comments

You can enter non-ASCII characters (including multibyte characters) in SQL comments if the database locale supports the non-ASCII characters. For further information on the GLS aspects of SQL comments, see the *IBM Informix GLS User's Guide*.

Categories of SQL Statements

SQL statements are traditionally divided into these twelve logical categories:

- **Data definition statements**. These *data definition language* (DDL) statements can declare, rename, modify, or destroy database objects.
- **Data manipulation statements**. These *data manipulation language* (DML) statements can retrieve, insert, delete, or modify data values.
- **Cursor manipulation statements**. These statements can declare, open, and close *cursors*, which are data structures for operations on multiple rows of data.
- **Cursor optimization statements**. These statements can be used to improve the performance of cursors.
- **Dynamic management statements**. These statements support memory management and allow users to specify at runtime the details of DML operations.
- **Data access statements**. These statements specify access privileges and support concurrent access to the database by multiple users.
- **Data integrity statement**s. These implement transaction logging and support the referential integrity of the database.
- **Optimization statements**. These can be used to improve the performance of operations on the database.
- **Routine definition statements**. These can declare, define, modify, execute, or destroy user-defined routines that the database stores.
- **Client/server connection statements**. These can open or close a connection between a database and a client application.
- **Auxiliary statements**. These can provide information about the database. (This is also a residual category for statements that are not closely related to the other statement categories.)
- **Optical subsystem statements**. These support storage and retrieval of database objects in the optical subsystem, whose statements are separately documented in *IBM Informix Optical Subsystem Guide*.)

The SQL statements of each category are listed in the pages that follow. As their descriptions in Chapter 3 indicate, some statements (and options of some statements, as designated with special icons in the syntax diagrams) are specific to Dynamic Server or to Extended Parallel Server.

Data Definition Statements

ALTER ACCESS_METHOD
ALTER FRAGMENT
ALTER FUNCTION
ALTER INDEX
ALTER PROCEDURE
ALTER ROUTINE
ALTER SEQUENCE
ALTER TABLE
CLOSE DATABASE
CREATE ACCESS_METHOD
CREATE AGGREGATE
CREATE CAST
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE DUPLICATE
CREATE EXTERNAL TABLE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE INDEX
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROLE
CREATE ROUTINE FROM
CREATE ROW TYPE
CREATE SCHEMA
CREATE SEQUENCE
CREATE SYNONYM
CREATE TABLE
CREATE Temporary TABLE
CREATE TRIGGER
CREATE VIEW
DROP ACCESS_METHOD
DROP AGGREGATE
DROP CAST
DROP DATABASE
DROP DUPLICATE
DROP FUNCTION
DROP INDEX
DROP OPCLASS
DROP PROCEDURE
DROP ROLE
DROP ROUTINE
DROP ROW TYPE
DROP SEQUENCE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP VIEW
RENAME COLUMN
RENAME DATABASE
RENAME INDEX
RENAME SEQUENCE
RENAME TABLE

Data Manipulation Statements

DELETE
INSERT
SELECT
UPDATE
LOAD
TRUNCATE
UNLOAD

Cursor Manipulation Statements

CLOSE
DECLARE
FETCH
FLUSH
FREE
OPEN
PUT
SET AUTOFREE

Cursor Optimization Statements

SET AUTOFREE
SET DEFERRED_PREPARE

Dynamic Management Statements

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DESCRIBE
DESCRIBE INPUT
EXECUTE
EXECUTE IMMEDIATE
FREE
GET DESCRIPTOR
INFO
PREPARE
SET DEFERRED_PREPARE
SET DESCRIPTOR

Data Access Statements

GRANT
GRANT FRAGMENT
LOCK TABLE
REVOKE
REVOKE FRAGMENT
SET ISOLATION
SET LOCK MODE
SET ROLE
SET SESSION AUTHORIZATION
SET TRANSACTION
SET Transaction Mode
UNLOCK TABLE

Data Integrity Statements

BEGIN WORK
COMMIT WORK
ROLLBACK WORK
SET Database Object Mode
SET LOG
SET PLOAD FILE
SET Transaction Mode
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE

Optimization Statements

SET Default Table Space
SET Default Table Type
SET ENVIRONMENT
SET EXPLAIN
SET OPTIMIZATION
SET PDQPRIORITY
SET Residency
SET SCHEDULE LEVEL
SET STATEMENT CACHE
UPDATE STATISTICS

Routine Definition Statements

ALTER FUNCTION
ALTER PROCEDURE
ALTER ROUTINE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROUTINE FROM
DROP FUNCTION
DROP PROCEDURE
DROP ROUTINE
EXECUTE FUNCTION
EXECUTE PROCEDURE
SET DEBUG FILE TO

Auxiliary Statements

INFO
OUTPUT
GET DIAGNOSTICS
SET COLLATION
SET DATASKIP
WHENEVER

Client/Server Connection Statements

CONNECT
DATABASE
DISCONNECT
SET CONNECTION

IDS

Optical Subsystem Statements

ALTER OPTICAL CLUSTER
CREATE OPTICAL CLUSTER
DROP OPTICAL CLUSTER
RELEASE
RESERVE
SET MOUNTING TIMEOUT

Important: *Optical Subsystem statements are described in the "IBM Informix Optical Subsystem Guide."*

ANSI

ANSI Compliance and Extensions

The following lists show statements that are compliant with the ANSI SQL-92 standard at the entry level, statements that are ANSI compliant but include Informix extensions, and statements that are Informix extensions to the ANSI standard.

ANSI-Compliant Statements

CLOSE
COMMIT WORK
EXECUTE IMMEDIATE
ROLLBACK WORK
SET SESSION AUTHORIZATION
SET TRANSACTION

ANSI-Compliant Statements with Informix Extensions

CONNECT
CREATE SCHEMA AUTHORIZATION
CREATE TABLE
CREATE Temporary TABLE
CREATE VIEW
DECLARE
DELETE
DISCONNECT
EXECUTE
FETCH
GRANT
INSERT
OPEN
SELECT
SET CONNECTION
SET Transaction Mode
UPDATE
WHENEVER

Statements That Are Extensions to the ANSI Standard

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
ALTER ACCESS_METHOD
ALTER FRAGMENT
ALTER FUNCTION
ALTER INDEX
ALTER OPTICAL CLUSTER
ALTER PROCEDURE
ALTER ROUTINE
ALTER SEQUENCE
ALTER TABLE

BEGIN WORK

CLOSE DATABASE
CREATE ACCESS_METHOD
CREATE AGGREGATE
CREATE CAST
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE DUPLICATE
CREATE EXTERNAL TABLE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE INDEX
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE OPTICAL CLUSTER
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROLE
CREATE ROUTINE FROM
CREATE ROW TYPE
CREATE SEQUENCE
CREATE SYNONYM
CREATE TRIGGER

DATABASE
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DESCRIBE
DESCRIBE INPUT
DROP ACCESS_METHOD
DROP AGGREGATE
DROP CAST
DROP DATABASE
DROP DUPLICATE
DROP FUNCTION
DROP INDEX
DROP OPCLASS
DROP OPTICAL CLUSTER
DROP PROCEDURE
DROP ROLE
DROP ROUTINE
DROP ROW TYPE
DROP SEQUENCE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP VIEW

EXECUTE FUNCTION
EXECUTE PROCEDURE
FLUSH
FREE

GET DESCRIPTOR
GET DIAGNOSTICS
GRANT FRAGMENT
INFO

LOAD
LOCK TABLE

OUTPUT

PREPARE
PUT

RELEASE
RENAME COLUMN
RENAME DATABASE
RENAME INDEX
RENAME SEQUENCE
RENAME TABLE
RESERVE
REVOKE
REVOKE FRAGMENT

SET AUTOFREE
SET COLLATION
SET Database Object Mode
SET DATASKIP
SET DEBUG FILE TO
SET Default Table Type
SET Default Table Space
SET DEFERRED_PREPARE
SET DESCRIPTOR
SET ENVIRONMENT
SET EXPLAIN
SET ISOLATION
SET LOCK MODE
SET LOG
SET MOUNTING TIMEOUT
SET OPTIMIZATION
SET PDQPRIORITY
SET PLOAD FILE
SET RESIDENCY
SET ROLE
SET SCHEDULE LEVEL
SET STATEMENT CACHE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE

TRUNCATE

UNLOAD
UPDATE STATISTICS
UNLOCK TABLE

Chapter

2

SQL Statements

In This Chapter

This chapter describes the syntax and semantics of SQL statements that are recognized by Dynamic Server or Extended Parallel Server. Statements (and statement segments, and notes describing usage) that are not marked by the icon for one of these database servers are valid for both.

The statement descriptions appear in alphabetical order. For some statements, important details of the semantics appear in other volumes of this documentation set, as indicated by cross-references.

For many statements, the syntax diagram, or the table of terms immediately following the diagram, or both, can includes references to syntax segments in Chapter 4, "Segments."

When the name of an SQL statement includes lowercase characters, such as "CREATE Temporary TABLE," it means that two or more different keywords can follow the preceding uppercase keyword.

For an explanation of the structure of statement descriptions, see Chapter 1, "Overview of SQL Syntax."

+

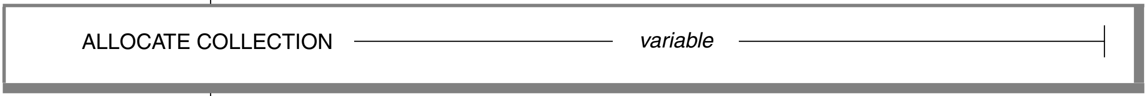
IDS

E/C

ALLOCATE COLLECTION

Use the ALLOCATE COLLECTION statement to allocate memory for a variable of a collection data type (such as LIST, MULTISET, or SET) or an untyped collection variable. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
variable	Name of typed or untyped collection variable to allocate	Must be an unallocated ESQL/C collection-type host variable	Must conform to language-specific rules for names

Usage

The ALLOCATE COLLECTION statement allocates memory for an ESQL/C variable that can store the value of a collection data type.

To create a collection variable for an ESQL/C program

1. Declare the collection variable as a client collection variable in an ESQL/C program.

   The collection variable can be a typed or untyped collection variable.

2. Allocate memory for the collection variable with the ALLOCATE COLLECTION statement.

The ALLOCATE COLLECTION statement sets **SQLCODE** (**sqlca.sqlcode**) to zero (0) if the memory allocation was successful and to a negative error code if the allocation failed.

You must explicitly release memory with the DEALLOCATE COLLECTION statement. After you free the collection variable with the DEALLOCATE COLLECTION statement, you can reuse the collection variable.

Tip: *The ALLOCATE COLLECTION statement allocates memory for an ESQL/C collection variable only. To allocate memory for an ESQL/C row variable, use the ALLOCATE ROW statement.*

Examples

The following example shows how to allocate resources with the ALLOCATE COLLECTION statement for the untyped collection variable, **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
. . .
```

The following example uses ALLOCATE COLLECTION to allocate resources for a typed collection variable, **a_typed_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_typed_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_typed_set;
. . .
```

Related Information

Related examples: Refer to the collection-variable example in PUT.

Related statements: ALLOCATE ROW and DEALLOCATE COLLECTION

For a discussion of collection data types in ESQL/C programs, see the *IBM Informix ESQL/C Programmer's Manual*.

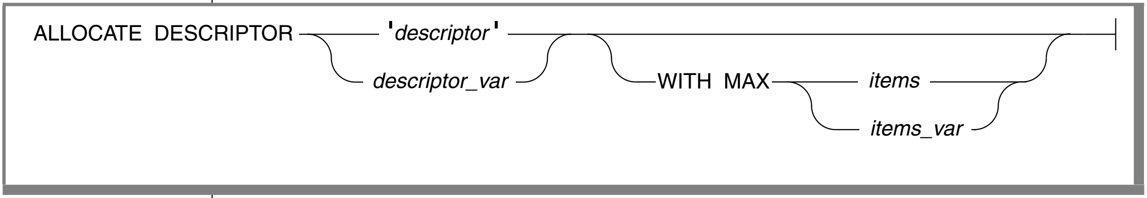
+

E/C

ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate memory for a system-descriptor area. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	Use single (') quotes. Must be the unique name of an unallocated system-descriptor area	Quoted String, p. 4-243
descriptor_var	Host variable that contains the name of a system-descriptor area	Must contain name of an unallocated system-descriptor area	Language specific
items	Number of item descriptors in *descriptor.* Default value is `100`.	Must be an unsigned INTEGER greater than zero	Literal Number, p. 4-216
items_var	Host variable that contains the number of items	Data type must be INTEGER or SMALLINT	Language specific

Usage

The ALLOCATE DESCRIPTOR statement creates a *system-descriptor area*, which is a location in memory to hold information that a DESCRIBE statement obtains, or to hold information about the WHERE clause of a statement.

A system-descriptor area contains one or more fields called *item descriptors*. Each item descriptor holds a data value that the database server can receive or send. The item descriptors also contain information about the data, such as data type, length, scale, precision, and nullability.

A system-descriptor area holds information that a DESCRIBE...USING SQL DESCRIPTOR statement obtains or it holds information about the WHERE clause of a dynamically executed statement.

If the name that you assign to a system-descriptor area matches the name of an existing system-descriptor area, the database server returns an error. If you free the descriptor with the DEALLOCATE DESCRIPTOR statement, you can reuse the descriptor.

WITH MAX Clause

You can use the WITH MAX clause to indicate the maximum number of item descriptors you need. When you use this clause, the COUNT field is set to the number of *items that* you specify. If you do not specify the WITH MAX clause, the default value of the COUNT field is `100`. You can change the value of the COUNT field with the SET DESCRIPTOR statement.

The following examples show valid ALLOCATE DESCRIPTOR statements. Each example includes the WITH MAX clause. The first line uses embedded variable names to identify the system-descriptor area and to specify the desired number of item descriptors. The second line uses a quoted string to identify the system-descriptor area and an unsigned integer to specify the desired number of item descriptors.

```
EXEC SQL allocate descriptor :descname with max :occ;

EXEC SQL allocate descriptor 'desc1' with max 3;
```

Related Information

Related statements: DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR

For more information on system-descriptor areas, refer to the *IBM Informix ESQL/C Programmer's Manual*.

+

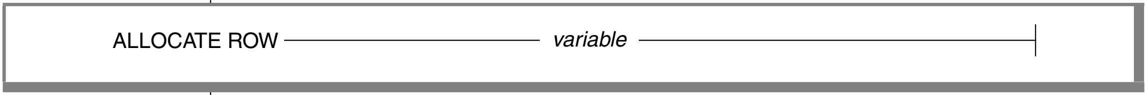
IDS

E/C

ALLOCATE ROW

Use the ALLOCATE ROW statement to allocate memory for a ROW variable. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
variable	Name of a typed or untyped ROW variable to allocate	Must be an unallocated ESQL/C ROW-type host variable.	Must conform to language-specific rules for names.

Usage

The ALLOCATE ROW statement allocates memory for a variable that stores ROW-type data.

To create a row variable using your ESQL/C program

1. Declare the ROW variable.

   The ROW variable can be a typed or untyped ROW variable.
2. Allocate memory for the ROW variable with the ALLOCATE ROW statement.

The ALLOCATE ROW statement sets **SQLCODE** (**sqlca.sqlcode**) to zero (0) if the memory allocation was successful and to a negative error code if the allocation failed.

You must explicitly release memory with the DEALLOCATE ROW statement. Once you free the ROW variable with the DEALLOCATE ROW statement, you can reuse the ROW variable.

Tip: *The ALLOCATE ROW statement allocates memory for an ESQL/C row variable only. To allocate memory for an ESQL/C collection variable, use the ALLOCATE COLLECTION statement.*

When you use the same ROW variable in multiple calls without deallocating it, a memory leak on the client computer results. Because there is no way to determine if a pointer is valid when it is passed, ESQL/C assumes that it is not valid and assigns it to a new memory location.

Example

The following example shows how to allocate resources with the ALLOCATE ROW statement for the typed ROW variable, **a_row**:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (a int, b int) a_row;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate row :a_row;
```

Related Information

Related statements: ALLOCATE COLLECTION and DEALLOCATE ROW

For a discussion of complex data types in ESQL/C programs, see the *IBM Informix ESQL/C Programmer's Manual*.

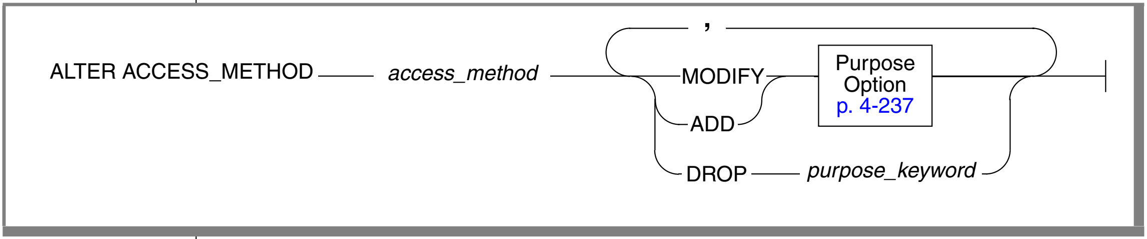
+

IDS

ALTER ACCESS_METHOD

The ALTER ACCESS_METHOD statement changes the attributes of a user-defined access method in the **sysams** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
access_method	Name of the access method to alter	The access method must be registered in the **sysams** system catalog table with a previous CREATE ACCESS_METHOD statement	Database Object Name, p. 4-46
purpose_keyword	A keyword that indicates which feature to change	The keyword must be associated with the access method by a previous statement	Purpose Functions, Methods, Flags, and Values, p. 4-239

Usage

Use ALTER ACCESS_METHOD to modify the definition of a user-defined access method. You must be the owner of the access method or have DBA privileges to alter an access method.

When you alter an access method, you change the purpose-option specifications (purpose functions, purpose methods, purpose flags, or purpose values) that define the access method. For example, you might alter an access method to assign a new user-defined function or method name or to provide a multiplier for the scan cost on a table.

If a transaction is in progress, the database server waits to alter the access method until the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

Example

The following statement alters the **remote** user-defined access method:

```
ALTER ACCESS_METHOD remote
   ADD am_scancost = FS_scancost,
   ADD am_rowids,
   DROP am_getbyid,
   MODIFY am_costfactor = 0.9;
```

The preceding example makes the following changes to the access method:

- Adds a user-defined function or method named **FS_scancost()**, which is associated in the **sysams** table with the **am_scancost** keyword
- Sets (adds) the **am_rowids** flag
- Drops the user-defined function or method associated with the **am_getbyid** keyword
- Modifies the **am_costfactor** value

Related Information

Related statements: CREATE ACCESS_METHOD and DROP ACCESS_METHOD

For detailed information about how to set purpose-option specifications, see "Purpose Options" on page 4-237.

For more information on primary-access methods, see the *IBM Informix Virtual-Table Interface Programmer's Guide*.

For more information on secondary-access methods, see the *IBM Informix Virtual-Index Interface Programmer's Guide*.

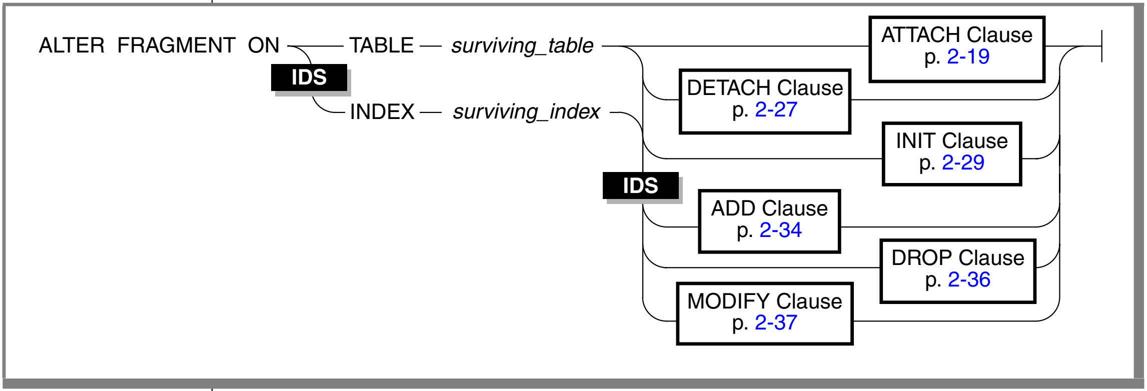
For a discussion of privileges, see the GRANT statement or the *IBM Informix Database Design and Implementation Guide*.

+

ALTER FRAGMENT

Use the ALTER FRAGMENT statement to alter the distribution strategy or storage location of an existing table or index.

Syntax



Element	Purpose	Restrictions	Syntax
surviving_index	Index on which to modify the distribution or storage	Must exist when the statement executes	Database Object Name, p. 4-46
surviving_table	Table on which to modify the distribution or storage	Must exist. See "Restrictions on the ALTER FRAGMENT Statement" on page 2-17	Database Object Name, p. 4-46

Usage

The ALTER FRAGMENT statement applies only to table fragments or index fragments that are located at the current site (or cluster, for Extended Parallel Server). No remote information is accessed or updated.

You must have the Alter or the DBA privilege to change the fragmentation strategy of a table. You must have the Index or the DBA privilege to alter the fragmentation strategy of an index.

Warning: *This statement can cause indexes to be dropped and rebuilt. Before undertaking alter operations, check corresponding sections in your "Performance Guide" to review effects and strategies.*

Clauses of the ALTER FRAGMENT statement support the following tasks.

Clause	**Purpose**
ATTACH	Combines tables that contain identical table structures into a single fragmented table
DETACH	Detaches a table fragment or slice from a fragmentation strategy and places it in a new table
INIT	Provides the following options: ■ Defines and initializes a fragmentation strategy on a table ■ Creates a fragmentation strategy for tables ■ Changes the order of evaluation of fragment expressions ■ Alters the fragmentation strategy of an existing table or index ■ Changes the storage location of an existing table
ADD	Adds an additional fragment to an existing fragmentation list
DROP	Drops an existing fragment from a fragmentation list
MODIFY	Changes an existing fragmentation expression

Use the CREATE TABLE statement or the INIT clause of the ALTER FRAGMENT statement to create fragmented tables.

Restrictions on the ALTER FRAGMENT Statement

You cannot use the ALTER FRAGMENT statement on a temporary table, an external table, or on a view. If your table or index is not already fragmented, the only clauses available to you are INIT and ATTACH.

XPS

You cannot use ALTER FRAGMENT on a generalized-key (GK) index. If the *surviving_table* has hash fragmentation, the only clauses available are ATTACH and INIT. You cannot use the ALTER FRAGMENT statement on any table that has a dependent GK index defined on it. In addition, you cannot use this statement on a table that has range fragmentation. ♦

IDS

You cannot use ALTER FRAGMENT on a typed table that is part of a table hierarchy. ♦

ALTER FRAGMENT and Transaction Logging

If your database uses logging, ALTER FRAGMENT is executed within a single transaction. When the fragmentation strategy uses large numbers of records, you might run out of log space or disk space. (The database server requires extra disk space for the operation; it later frees the disk space).

When you run out of log space or disk space, try one of the following procedures to make more space available:

- Turn off logging and turn it back on again at the end of the operation. This procedure indirectly requires a backup of the root dbspace.
- Split the operations into multiple ALTER FRAGMENT statements, moving a smaller portion of records each time.

For information about log-space requirements and disk-space requirements, see your *Administrator's Guide*. That guide also contains detailed instructions about how to turn off logging. For information about backups, refer to your *IBM Informix Backup and Restore Guide*.

Determining the Number of Rows in the Fragment

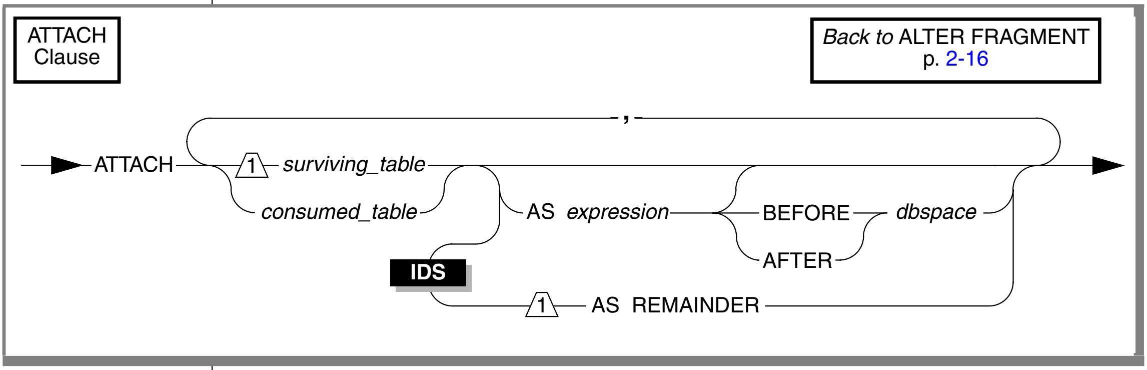
You can place as many rows into a fragment as the available space in the dbspace allows.

To find out how many rows are in a fragment

1. Run the UPDATE STATISTICS statement on the table. This step fills the **sysfragments** system catalog table with the current table information.
2. Query the **sysfragments** system catalog table to examine the **npused** and **nrows** fields. The **npused** field gives you the number of data pages used in the fragment, and the **nrows** field gives you the number of rows in the fragment.

ATTACH Clause

Use the ATTACH clause to combine tables that contain identical table structures into a fragmentation strategy.



Element	Purpose	Restrictions	Syntax
consumed_ table	Table that loses its identity to be merged with *surviving_table*	Must exist. Cannot include serial columns nor unique, referential, or primary key constraints. See also "General Restrictions for the ATTACH Clause" on page 2-20.	Database Object Name, p. 4-46
dbspace	Dbspace(s) that specifies where the consumed table expression exists in the fragmentation list	Must exist. See also "Altering Hybrid-Fragmented Tables" on page 2-22.	Identifier, p. 4-189
expression	Expression that defines which rows are stored in a fragment	Can include only columns from the current table, and only data values from a single row. See also "General Restrictions for the ATTACH Clause" on page 2-20.	Condition, p. 4-24; Expression, p. 4-67
surviving_table	Table on which to modify the distribution or storage	Must exist. Cannot have any constraints. See also "Restrictions on the ALTER FRAGMENT Statement" on page 2-17.	Database Object Name, p. 4-46

To use this clause, you must have the DBA privilege, or else be the owner of the specified tables. The ATTACH clause supports the following tasks:

- Creates a single fragmented table by combining two or more identically-structured, nonfragmented tables

  (See "Combining Nonfragmented Tables to Create a Fragmented Table" on page 2-21.)
- Attaches one or more tables to a fragmented table

  (See "Attaching a Table to a Fragmented Table" on page 2-21.)

General Restrictions for the ATTACH Clause

Any tables that you attach must have been created previously in separate dbspaces. You cannot attach the same table more than once.

All consumed tables listed in the ATTACH clause must have the same structure as the surviving table. The number, names, data types, and relative position of the columns must be identical.

The *expression* cannot include aggregates, subqueries, nor variant functions.

IDS

User-defined routines and references to fields of a row-type column are not valid. You cannot attach a fragmented table to another fragmented table. ♦

XPS

Additional Restrictions on the ATTACH Clause Specific to XPS

In addition to the general restrictions, every consumed table must be of the same usage type as the surviving table. For information about how to specify the usage type of a table, refer to "Logging Options" on page 2-215.

The ATTACH clause is not valid under either of the following conditions:

- If the consumed tables contain data that belongs in some existing fragment of the surviving table
- If existing data in the surviving table would belong in a new fragment

Thus, you cannot use the ATTACH clause for data movement among fragments. To perform this task, see the "INIT Clause" on page 2-29.

Using the BEFORE, AFTER, and REMAINDER Options

The BEFORE and AFTER options allow you to place a new fragment either before or after an existing dbspace. You cannot use the BEFORE and AFTER options when the distribution scheme is round-robin.

When you attach a new fragment without an explicit BEFORE or AFTER option, the database server places the added fragment at the end of the fragmentation list, unless a remainder fragment exists. If a remainder fragment exists, the new fragment is placed just before the remainder fragment. You cannot attach a new fragment after the remainder fragment.

XPS

When you create or append to a hybrid-fragmented table, the positioning specification (BEFORE, AFTER, or REMAINDER) applies to an entire dbslice. You can use any dbspace in a dbslice to identify the dbslice for the BEFORE or AFTER position. ♦

Combining Nonfragmented Tables to Create a Fragmented Table

When you transform tables with identical table structures into fragments in a single table, you allow the database server to manage the fragmentation instead of allowing the application to manage the fragmentation. The distribution scheme can be round-robin or expression-based.

To make a single, fragmented table from two or more identically-structured, nonfragmented tables, the ATTACH clause must contain the surviving table in the *attach list*. The attach list is the list of tables in the ATTACH clause.

IDS

To include a **rowid** column in the newly-created single, fragmented table, attach all tables first and then add the **rowid** with the ALTER TABLE statement. ♦

Attaching a Table to a Fragmented Table

To attach a nonfragmented table to an already fragmented table, the nonfragmented table must have been created in a separate dbspace and must have the same table structure as the fragmented table. In the following example, a round-robin distribution scheme fragments the table **cur_acct**, and the table **old_acct** is a nonfragmented table that resides in a separate dbspace. The example shows how to attach **old_acct** to **cur_acct**:

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH old_acct
```

IDS

When you attach one or more tables to a fragmented table, a consumed_table must be nonfragmented. ♦

XPS

When you attach one or more tables to a fragmented table, a consumed_table can be nonfragmented or have hash fragmentation.

If you specify a consumed_table that has hash fragmentation, the hash column specification must match that of the surviving_table and any other consumed_table involved in the attach operation. ♦

XPS

Altering Hybrid-Fragmented Tables

When you alter a hybrid-fragmented table with either an ATTACH or DETACH clause, you need specify only one dbspace to identify the entire set of dbspaces that are associated with a given expression in the base fragmentation strategy of the table. The set of dbspaces associated with an expression in the base fragmentation strategy of the table might have been defined as one or more dbslices or a dbspaces. For more information, see "Fragmenting by HYBRID" on page 2-243.

If you know the name of the dbslice but not the names any of the dbspaces that it is made up of, you can name the first dbspace in the dbslice by adding `.1` to the name of the dbslice. For example, if the dbslice were named **dbsl1**, you could specify `dbsl1.1`.

Effect of the ATTACH Clause

After an ATTACH operation, all consumed tables no longer exist. Any constraints (CHECK or NOT NULL) that were on the consumed tables also no longer exist. You must reference the records that were in the consumed tables through the surviving table.

What Happens to Indexes?

In a logging database, an ATTACH operation extends any attached index on the surviving table according to the new fragmentation strategy of the surviving table. All rows in the consumed table are subject to these automatically adjusted indexes. For information on whether the database server completely rebuilds the index on the surviving table or reuses an index that was on the consumed table, see your *Performance Guide*.

IDS

In a nonlogging database, an ATTACH operation does not extend indexes on the surviving table according to the new fragmentation strategy of the surviving table. To extend the fragmentation strategy of an attached index according to the new fragmentation strategy of the surviving table, you must drop the index and re-create it on the surviving table. ♦

A detached index on the surviving table retains its same fragmentation strategy. That is, a detached index does not automatically adjust to accommodate the new fragmentation of the surviving table. For more information on what happens to indexes, see the discussion about altering table fragments in your *Performance Guide*.

What Happens to BYTE and TEXT Columns?

IDS

Each BYTE and TEXT column in every table that is named in the ATTACH clause must have the same storage type, either blobspace or tblspace. If the BYTE or TEXT column is stored in a blobspace, the same column in all tables must be in the same blobspace. If the BYTE or TEXT column is stored in a tblspace, the same column must be stored in a tblspace in all tables. ♦

XPS

In Extended Parallel Server, BYTE and TEXT columns are stored in separate fragments created for that purpose. If a table includes a BYTE or TEXT column, the database server creates a separate, additional fragment in the same dbspace as each regular table fragment. BYTE or TEXT columns are stored in the separate fragment that is associated with the regular table fragment where a given row resides.

When an ATTACH occurs, BYTE and TEXT fragments of the consumed table become part of the surviving table and continue to be associated with the same rows and data fragments as they were before the ATTACH. ♦

What Happens to Triggers and Views?

When you attach tables, triggers on the surviving table survive the ATTACH, but triggers on the consumed table are automatically dropped. No triggers are activated by the ATTACH clause, but subsequent data-manipulation operations on the new rows can activate triggers.

Views on the surviving table survive the ATTACH operation, but views on the consumed table are automatically dropped.

What Happens with the Distribution Scheme?

You can attach a nonfragmented table to a table with any type of supported distribution scheme. In general, the resulting table has the same fragmentation strategy as the prior fragmentation strategy of the surviving_table.

When you attach two or more nonfragmented tables, however, the distribution scheme can either be based on expression or round-robin.

IDS

The following table shows the distribution schemes that can result from different distribution schemes of the tables mentioned in the ATTACH clause.

Prior Distribution Scheme of Surviving Table	Prior Distribution Scheme of Consumed Table	Resulting Distribution Scheme
None	None	Round-robin or expression
Round-robin	None	Round-robin
Expression	None	Expression

♦

XPS

The following table shows the distribution schemes that can result from different distribution schemes of the tables mentioned in the ATTACH clause.

Prior Distribution Scheme of Surviving Table	Prior Distribution Scheme of Consumed Table	Resulting Distribution Scheme
None	None	Round-robin or expression
None	Hash	Hybrid
Round-robin	None	Round-robin
Expression	None	Expression
Hash	None	Hybrid
Hash	Hash	Hybrid
Hybrid	None	Hybrid
Hybrid	Hash	Hybrid

When you attach a nonfragmented table to a table that has hash fragmentation, the resulting table has hybrid fragmentation.

You can attach a table with a hash distribution scheme to a table that currently has no fragmentation, hash fragmentation, or hybrid fragmentation. In any of these situations, the resulting table has a hybrid distribution scheme. ♦

The following examples illustrate the use of the ATTACH clause to create fragmented tables with different distribution schemes.

Round-Robin Distribution Scheme

The following example combines nonfragmented tables **pen_types** and **pen_makers** into a single, fragmented table, **pen_types**. Table **pen_types** resides in dbspace **dbsp1,** and table **pen_makers** resides in dbspace **dbsp2**. Table structures are identical in each table.

```
ALTER FRAGMENT ON TABLE pen_types ATTACH pen_types, pen_makers
```

After you execute the ATTACH clause, the database server fragments the table **pen_types** round-robin into two dbspaces: the dbspace that contained **pen_types** and the dbspace that contained **pen_makers**. Table **pen_makers** is consumed, and no longer exists; all rows that were in table **pen_makers** are now in table **pen_types**.

Expression Distribution Scheme

Consider the following example that combines tables **cur_acct** and **new_acct** and uses an expression-based distribution scheme. Table **cur_acct** was originally created as a fragmented table and has fragments in dbspaces **dbsp1** and **dbsp2**. The first statement of the example shows that table **cur_acct** was created with an expression-based distribution scheme. The second statement of the example creates table **new_acct** in **dbsp3** without a fragmentation strategy. The third statement combines the tables **cur_acct** and **new_acct**. Table structures (columns) are identical in each table.

```
CREATE TABLE cur_acct (a int) FRAGMENT BY EXPRESSION
   a < 5 in dbsp1,
   a >= 5 and a < 10 in dbsp2;

CREATE TABLE new_acct (a int) IN dbsp3;

ALTER FRAGMENT ON TABLE cur_acct ATTACH new_acct AS a>=10;
```

When you examine the **sysfragments** system catalog table after you alter the fragment, you see that table **cur_acct** is fragmented by expression into three dbspaces. For additional information about the **sysfragments** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

In addition to simple range rules, you can use the ATTACH clause to fragment by expression with hash or arbitrary rules. For a discussion of all types of expressions in an expression-based distribution scheme, see "FRAGMENT BY Clause for Tables" on page 2-31.

Warning: *When you specify a date value as the default value for a parameter, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the date value. When you specify a 2-digit year,* ***DBCENTURY*** *can affect how the database server interprets the date value, so the UDR might not use the default value that you intended. For more information, see the "IBM Informix Guide to SQL: Reference."*

XPS

Hybrid Fragmentation Distribution Scheme

Consider a case where monthly sales data is added to the **sales_info** table defined below. Due to the large amount of data, the table is distributed evenly across multiple coservers with a system-defined hash function. To manage monthly additions of data to the table, it is also fragmented by a date expression. The combined hybrid fragmentation is declared in the following CREATE TABLE statement:

```
CREATE TABLE sales_info (order_num INT, sale_date DATE, ...)
   FRAGMENT BY HYBRID (order_num) EXPRESSION
   sale_date >= '01/01/1996' AND sale_date < '02/01/1996'
   IN sales_slice_9601,
   sale_date >= '02/01/1996' AND sale_date < '03/01/1996'
   IN sales_slice_9602,
. . .
   sale_date >= '12/01/1996' AND sale_date < '01/01/1997'
   IN sales_slice_9612
```

The data values for a new month are originally loaded from an external source. The data values are distributed evenly across the name coservers on which the **sales_info** table is defined, using a system-defined hash function on the same column:

```
CREATE TABLE jan_97 (order_num INT, sale_date DATE, ...)
   FRAGMENT BY HASH (order_num) IN sales_slice_9701
INSERT INTO jan_97 SELECT (...) FROM ...
```

After data values are loaded, you can attach the new table to **sales_info.** You can issue the following ALTER FRAGMENT statement to attach the new table:

```
ALTER FRAGMENT ON TABLE sales_info ATTACH jan_97
   AS sale_date >= '01/01/1997' AND sale_date < '02/01/1997'
```

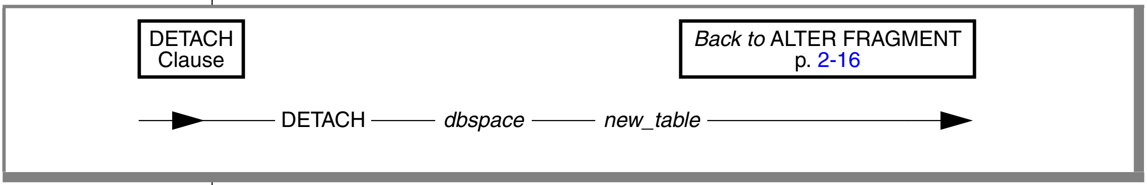
DETACH Clause

Use the DETACH clause to detach a table fragment from a distribution scheme and place the contents into a new nonfragmented table.

XPS

In Extended Parallel Server, the new table can also be a table with hash fragmentation. ♦

For an explanation of distribution schemes, see "FRAGMENT BY Clause for Tables" on page 2-31.



Element	Purpose	Restrictions	Syntax
dbspace	Dbspace that contains the fragment to be detached. With a hybrid-fragmented table, *dbspace* identifies a set of dbspaces (XPS only). See "Altering Hybrid-Fragmented Tables" on page 2-22.	Must exist at the time of execution.	Identifier, p. 4-189
new_table	Nonfragmented table that results after you execute the ALTER FRAGMENT statement. (In XPS, this can also be a hash-fragmented table.)	Must not exist before the time of execution.	Database Object Name, p. 4-46

The new table that results from executing the DETACH clause does not inherit any indexes or constraints from the original table. Only the data remains.

Similarly, the new table does not inherit any privileges from the original table. Instead, the new table has the default privileges for any new table. For further information on default table-level privileges, see the GRANT statement on "Table-Level Privileges" on page 2-463.

The DETACH clause cannot be applied to a table if that table is the parent of a referential constraint or if a **rowid** column is defined on the table.

XPS

In Extended Parallel Server, you cannot use the DETACH clause if the table has a dependent GK index defined on it. ♦

Detach That Results in a Nonfragmented Table

The following example shows the table **cur_acct** fragmented into two dbspaces, **dbsp1** and **dbsp2**:

```
ALTER FRAGMENT ON TABLE cur_acct DETACH dbsp2 accounts
```

This example detaches **dbsp2** from the distribution scheme for **cur_acct** and places the rows in a new table, **accounts**. Table **accounts** now has the same structure (column names, number of columns, data types, and so on) as table **cur_acct**, but the table **accounts** does not contain any indexes or constraints from the table **cur_acct**. Both tables are now nonfragmented. The following example shows a table that contains three fragments:

```
ALTER FRAGMENT ON TABLE bus_acct DETACH dbsp3 cli_acct
```

This statement detaches **dbsp3** from the distribution scheme for **bus_acct** and places the rows in a new table, **cli_acct**. Table **cli_acct** now has the same structure (column names, number of columns, data types, and so on) as **bus_acct,** but the table **cli_acct** does not contain any indexes or constraints from the table **bus_acct**. Table **cli_acct** is a nonfragmented table, and table **bus_acct** remains a fragmented table.

XPS

Detach That Results in a Table with Hash Fragmentation

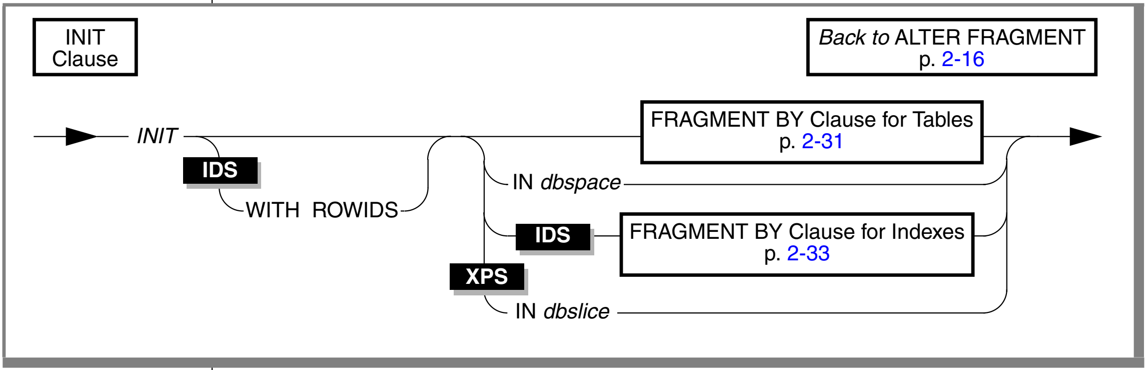
The new table is a hash-fragmented table if the *surviving_table* had hybrid fragmentation and the detached dbslice has more than one fragment. In a hybrid-fragmented table, the dbslice is detached if you specify any dbspace in that slice. For example, see the **sales_info** table discussed in the "Hybrid Fragmentation Distribution Scheme" on page 2-26. Once the January 1997 data is available in **sales_info**, you might archive year-old **sales_info** data.

```
ALTER FRAGMENT ON TABLE sales_info
   DETACH sales_slice_9601.1 jan_96
```

In this example, data from January 1996 is detached from the **sales_info** table and placed in a new table called **jan_96**.

INIT Clause

The INIT clause of ALTER FRAGMENT has the following syntax.


INIT Clause
Back to ALTER FRAGMENT p. 2-16
INIT
IDS
WITH ROWIDS
FRAGMENT BY Clause for Tables p. 2-31
IN dbspace
IDS
FRAGMENT BY Clause for Indexes p. 2-33
XPS
IN dbslice

Element	Purpose	Restrictions	Syntax
dbslice	Dbslice storing fragmented information	Must exist at time of execution.	Identifier, p. 4-189
dbspace	Dbspace storing fragmented information	Must exist at time of execution.	Identifier, p. 4-189

The INIT clause can accomplish tasks like the following:

- Move a nonfragmented table from one dbspace to another dbspace.
- Convert a fragmented table to a nonfragmented table.
- Fragment an existing not fragmented table without redefining it.
- Convert a fragmentation strategy to another fragmentation strategy.
- IDS Fragment an existing index that is not fragmented without redefining the index.
- Convert a fragmented index to a nonfragmented index. ♦

XPS You cannot use the INIT clause to change the fragmentation strategy of a table that has a GK index. ♦

When you use the INIT clause to modify a table, the **tabid** value in the system catalog tables changes for the affected table. The **constrid** of all unique and referential constraints on the table also change.

For more information about the storage spaces in which you can store a table, see "Using the IN Clause" on page 2-237.

Warning: *When you execute the ALTER FRAGMENT statement with this clause, it results in data movement if the table contains any data. If data moves, the potential exists for significant logging, for the transaction being aborted as a long transaction, and for a relatively long exclusive lock being held on the affected tables. Use this statement when it does not interfere with day-to-day operations.*

IDS

WITH ROWIDS Option

Nonfragmented tables contain a hidden column called **rowid**. By default, fragmented tables do not contain this column unless it is explicitly created. You can use the WITH ROWIDS option to add a new **rowid** column. The database server assigns a unique **rowid** number to each row and automatically creates an index that it can use to find the physical location of the row. The **rowid** value of a row cannot be updated, but remains stable during the existence of the row. Each row requires an additional 4 bytes to store the **rowid** column after you specify the WITH ROWIDS option.

Important: *The **rowid** column is a deprecated feature. You should use primary keys, rather than the **rowid** column, as an access method.*

Converting a Fragmented Table to a Nonfragmented Table

You might decide that you no longer want a table to be fragmented. You can use the INIT clause to convert a fragmented table to a nonfragmented table. The following example shows the original fragmentation definition as well as how to use the ALTER FRAGMENT statement to convert the table:

```
CREATE TABLE checks (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2, dbsp3;

ALTER FRAGMENT ON TABLE checks INIT IN dbsp1;
```

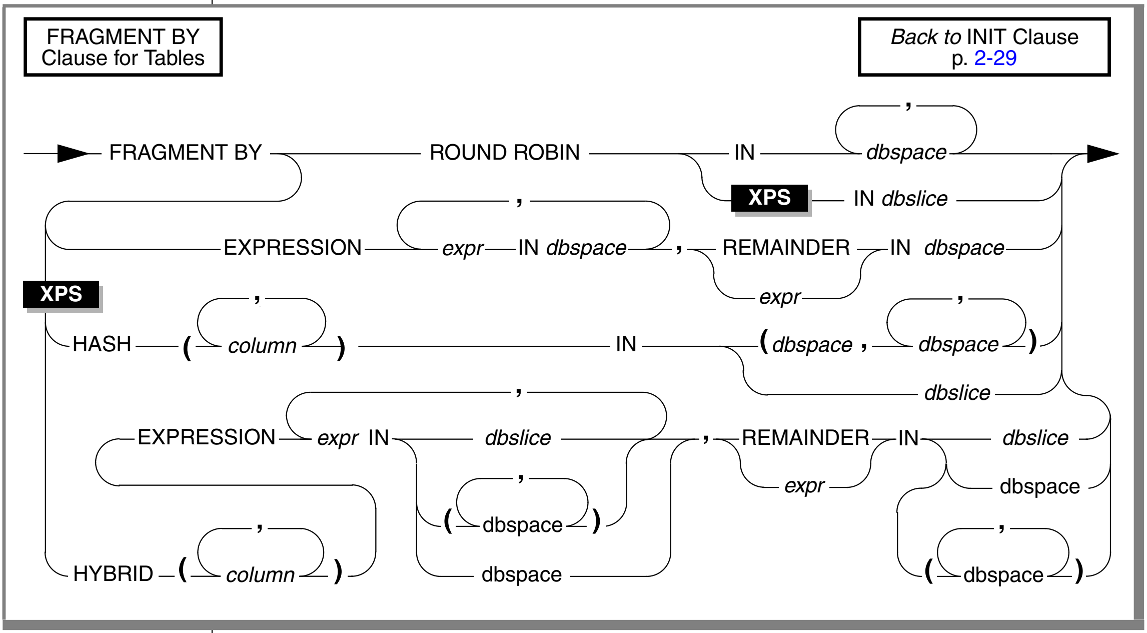
You must use the IN *dbspace* clause to place the table in a dbspace explicitly.

When you use the INIT clause to change a fragmented table to a nonfragmented table, all attached indexes become nonfragmented indexes. In addition, constraints that do not use existing user indexes (detached indexes) become nonfragmented indexes. All newly nonfragmented indexes exist in the same dbspace as the new nonfragmented table.

Using the INIT clause to change a fragmented table to a nonfragmented table has no effect on the fragmentation strategy of detached indexes, nor of constraints that use detached indexes.

FRAGMENT BY Clause for Tables

Use the FRAGMENT BY portion of the INIT clause to fragment an existing non-fragmented table, or to convert one fragmentation strategy to another.



Element	Purpose	Restrictions	Syntax
column	Column to which fragmentation strategy applies	Must exist in the specified table.	Identifier, p. 4-189
dbslice	Dbslice that contains the table fragment	Must be defined.	Identifier, p. 4-189
dbspace	Dbspace that contains the table fragment	Must specify at least two but no more than 2,048 dbspaces.	Identifier, p. 4-189
expr	Expression that defines a table fragment by a range, hash, or arbitrary rule	Must evaluate to a Boolean value (`true` or `false`).	Expression, p. 4-67

In the HYBRID clause, *column* identifies the column or columns on which you want to apply the hash portion of the hybrid table fragmentation strategy. The *expression* can contain only columns from the current table and only data values from a single row. No subqueries or aggregates are allowed. In addition, the built-in CURRENT, DATE, and TIME functions are not valid.

For more information on the available fragmentation strategies, see the "FRAGMENT BY Clause" on page 2-238.

Changing an Existing Fragmentation Strategy on a Table

You can redefine a fragmentation strategy on a table if you decide that your initial strategy does not fulfill your needs. When you alter a fragmentation strategy, the database server discards the existing fragmentation strategy and moves records to fragments as defined in the new fragmentation strategy.

The following example shows the statement that originally defined the fragmentation strategy on the table **account** and then shows an ALTER FRAGMENT statement that redefines the fragmentation strategy:

```
CREATE TABLE account (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2;
ALTER FRAGMENT ON TABLE account
   INIT FRAGMENT BY EXPRESSION
   col1 < 0 IN dbsp1,
   col2 >= 0 IN dbsp2;
```

If an existing dbspace is full when you redefine a fragmentation strategy, you must not use it in the new fragmentation strategy.

Defining a Fragmentation Strategy on a Nonfragmented Table

The INIT clause can define a fragmentation strategy on a nonfragmented table, regardless of whether the table was created with a storage option:

```
CREATE TABLE balances (col1 INT, col2 INT) IN dbsp1;
ALTER FRAGMENT ON TABLE balances INIT
   FRAGMENT BY EXPRESSION
   col1 <= 500 IN dbsp1,
   col1 > 500 AND col1 <=1000 IN dbsp2,
   REMAINDER IN dbsp3;
```

IDS

When you use the INIT clause to fragment an existing nonfragmented table, all indexes on the table become fragmented in the same way as the table. ♦

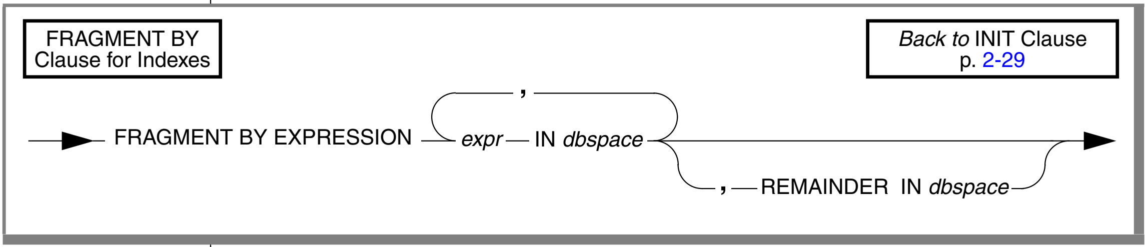
XPS

When you use the INIT clause to fragment an existing nonfragmented table, indexes retain their existing fragmentation strategy. ♦

IDS

FRAGMENT BY Clause for Indexes

The INIT FRAGMENT BY clause for indexes allows you to fragment an existing index that is not fragmented without redefining the index. Use this clause to define an expression-based distribution scheme for indexes.



Element	Purpose	Restrictions	Syntax
dbspace	Dbspace that contains the fragmented information	You must specify at least two but no more than 2,048 dbspaces.	Identifier, p. 4-189
expr	Expression defining an index fragment by a range, hash, or arbitrary rule	Must return a Boolean value.	Condition, p. 4-24; Expression, p. 4-67

You can convert an existing fragmentation strategy to another fragmentation strategy. Any existing fragmentation strategy is discarded and records are moved to fragments as defined in the new fragmentation strategy. You can also convert a fragmented index to a nonfragmented index.

The expression can contain only columns from the current table and data values from only a single row. No subqueries nor aggregates are allowed. The built-in CURRENT, DATE, and TIME functions are not valid here.

Fragmenting Unique and System Indexes

You can fragment unique indexes only if the table uses an expression-based distribution scheme. Any columns referenced in the fragment expression must be indexed columns. If your ALTER FRAGMENT INIT statement fails to meet either of these restrictions, the INIT fails, and work is rolled back.

You might have an attached unique index on a table fragmented by **Column A**. If you use INIT to change the table fragmentation to **Column B**, the INIT fails because the unique index is defined on **Column A**. To resolve this issue, use the INIT clause on the index to detach it from the table fragmentation strategy and fragment it separately.

System indexes (such as those used in referential constraints and unique constraints) use user indexes if the indexes exist. If no user indexes can be used, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented index on the constraint columns and then use the ALTER TABLE statement to add the constraint.

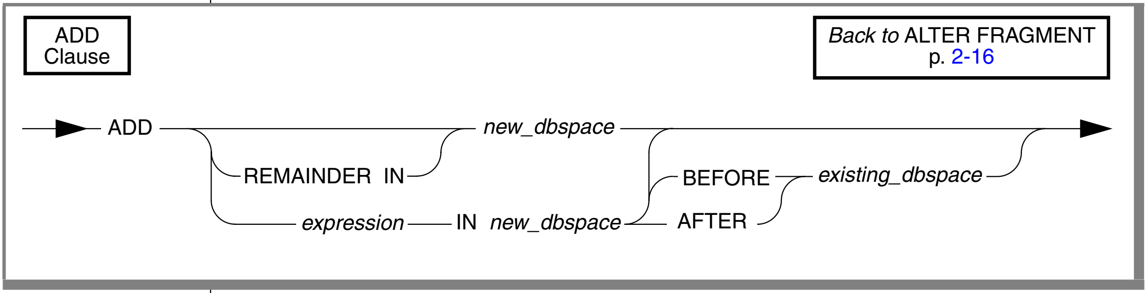
Detaching an Index from a Table-Fragmentation Strategy

You can detach an index from a table-fragmentation strategy with the INIT clause, which causes an attached index to become a detached index. This breaks any dependency of the index on the table fragmentation strategy.

IDS

ADD Clause

Use the ADD clause to add another fragment to an existing fragmentation list.



Element	Purpose	Restrictions	Syntax
existing_dbspace	Name of a dbspace in an existing fragmentation list	Must exist at the time when you execute the statement	Identifier, p. 4-189
expression	Expression that defines the new fragment that is to be added	Must return a Boolean value (`true` or `false`)	Condition, p. 4-24; Expression, p. 4-67
new_dbspace	Name of dbspace to be added to the fragmentation scheme	Must exist at the time when you execute the statement	Identifier, p. 4-189

The *expression* can contain *column* names only from the current table and data values only from a single row. No subqueries or aggregates are allowed. In addition, the built-in CURRENT, DATE, and TIME functions are not valid here.

Adding a New Dbspace to a Round-Robin Distribution Scheme

You can add more dbspaces to a round-robin distribution scheme. The following example shows the original round-robin definition:

```
CREATE TABLE book (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

To add another dbspace, use the ADD clause, as in this example:

```
ALTER FRAGMENT ON TABLE book ADD dbsp3;
```

Adding Fragment Expressions

Adding a fragment expression to the fragmentation list in an expression-based distribution scheme can relocate records from existing fragments into the new fragment. When you add a new fragment into the middle of the fragmentation list, all the data existing in fragments after the new one must be re-evaluated. The next example shows the original expression definition:

```
FRAGMENT BY EXPRESSION
   c1 < 100 IN dbsp1,
   c1 >= 100 AND c1 < 200 IN dbsp2,
   REMAINDER IN dbsp3;
```

To add another fragment to hold rows between `200` and `300`, use the following ALTER FRAGMENT statement:

```
ALTER FRAGMENT ON TABLE news ADD
   c1 >= 200 AND c1 < 300 IN dbsp4;
```

Any rows that were formerly in the remainder fragment and that fit the criteria `c1 >= 200` and `c1 < 300` are moved to the new dbspace.

Using the BEFORE and AFTER Options

The BEFORE and AFTER options allow you to place a new fragment either before or after an existing dbspace. You cannot use the BEFORE and AFTER options when the distribution scheme is round-robin.

When you attach a new fragment without an explicit BEFORE or AFTER option, the database server places the added fragment at the end of the fragmentation list, unless a remainder fragment exists. If a remainder fragment exists, the new fragment is placed just before the remainder fragment. You cannot attach a new fragment after the remainder fragment.

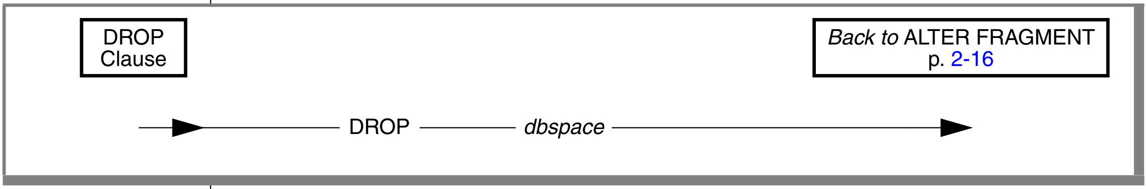
Using the REMAINDER Option

You cannot add a remainder fragment when one already exists. When you add a new fragment to the fragmentation list, and a remainder fragment exists, the records in the remainder fragment are retrieved and re-evaluated. Some of these records might move to the new fragment. The remainder fragment always remains the last item in the fragment list.

IDS

DROP Clause

Use the DROP clause to drop an existing fragment from a fragmentation list.



Element	Purpose	Restrictions	Syntax
dbspace	Name of dbspace that contains the dropped fragment	Must exist at the time when you execute the statement.	Identifier, p. 4-189

You cannot drop one of the fragments when the table contains only two fragments. You cannot drop a fragment in a table that is fragmented with an expression-based distribution scheme if the fragment contains data that cannot be moved to another fragment. If the distribution scheme contains a REMAINDER option, or if the expressions were constructed in an overlapping manner, you can drop a fragment that contains data.

When you want to make a fragmented table nonfragmented, use either the INIT or DETACH clause.

When you drop a fragment from a dbspace, the underlying dbspace is not affected. Only the fragment data values within that dbspace are affected.

When you drop a fragment, the database server attempts to move all the records in the dropped fragment to another fragment. In this case, the destination fragment might not have enough space for the additional records. When this happens, follow one of the procedures that are listed in "ALTER FRAGMENT and Transaction Logging" on page 2-18 to increase your space, and retry the procedure.

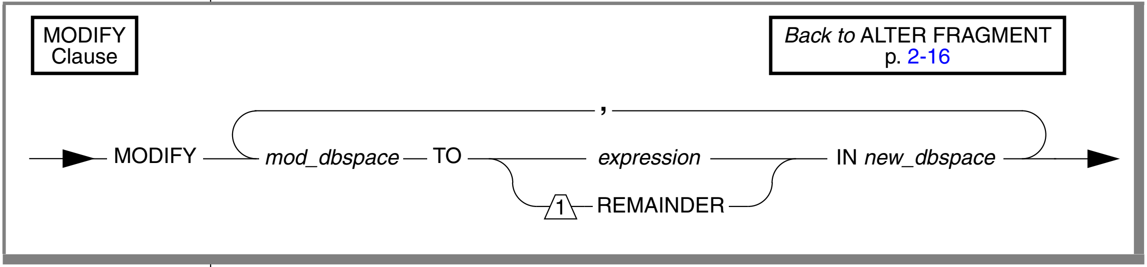
The following examples show how to drop a fragment from a fragmentation list. The first line shows how to drop an index fragment, and the second line shows how to drop a table fragment.

```
ALTER FRAGMENT ON INDEX cust_indx DROP dbsp2;

ALTER FRAGMENT ON TABLE customer DROP dbsp1;
```

IDS

MODIFY Clause

Use the MODIFY clause to change an existing fragment expression on an existing dbspace. You can also use the MODIFY clause to move a fragment expression from one dbspace to a different dbspace.



Element	Purpose	Restrictions	Syntax
expression	Modified range, hash, or arbitrary expression	Can specify columns in current table only and data from only a single row.	Condition, p. 4-24; Expression, p. 4-67
mod_ dbspace	Modified dbspace	Must exist when you execute the statement.	Identifier, p. 4-189
new_dbspace	Dbspace that contains the modified information	Must exist when you execute the statement.	Identifier, p. 4-189

The expression must evaluate to a Boolean value (true or false).

No subqueries nor aggregates are allowed in the *expression*. In addition, the built-in CURRENT, DATE, and TIME functions are not allowed.

When you use the MODIFY clause, the underlying dbspaces are not affected. Only the fragment data values within the dbspaces are affected.

You cannot change a REMAINDER fragment into a nonremainder fragment if records within the REMAINDER fragment do not satisfy the new *expression*.

When you use the MODIFY clause to change an expression without changing the dbspace storage for the expression, you must use the same name for the *mod_dbspace* and the *new_dbspace*, as in the following example:

```
ALTER FRAGMENT ON TABLE cust_acct
   MODIFY dbsp1 TO acct_num < 65 IN dbsp1
```

When you use the MODIFY clause to move an expression from one dbspace to another, *mod_dbspace* is the name of the dbspace where the expression was previously located, and *new_dbspace* is the new location for the expression:

```
ALTER FRAGMENT ON TABLE cust_acct
   MODIFY dbsp1 TO acct_num < 35 IN dbsp2
```

Here the distribution scheme for the **cust_acct** table is modified so that all row items in column **acct_num** that are less than 35 are now contained in the dbspace **dbsp2**. These items were formerly contained in the dbspace **dbsp1**.

When you use the MODIFY clause to change the expression and to move it to a new dbspace, you must change both the *expression* and the *dbspace* name.

If your indexes are attached indexes, and you modify the table, the index fragmentation strategy is also modified.

Related Information

Related statements: CREATE TABLE, CREATE INDEX, and ALTER TABLE

For a discussion of fragmentation strategy, refer to the *IBM Informix Database Design and Implementation Guide*.

For information on how to maximize performance when you make fragment modifications, see your *Performance Guide*.

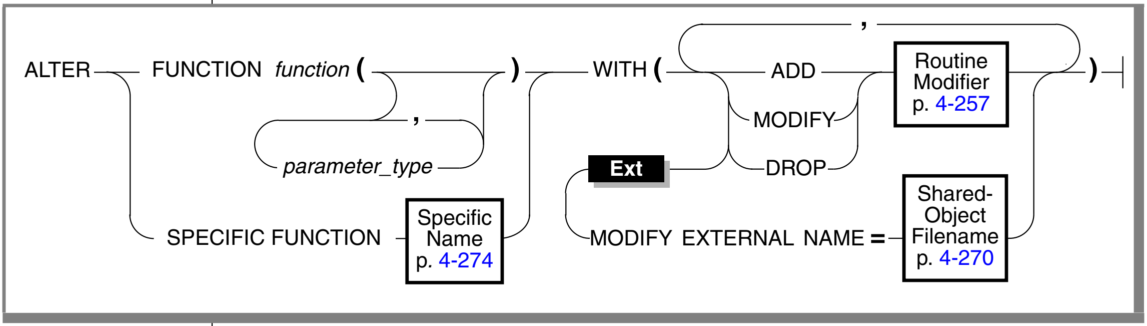
+

IDS

ALTER FUNCTION

Use the ALTER FUNCTION statement to change the routine modifiers or pathname of a user-defined function.

Syntax



Element	Purpose	Restrictions	Syntax
function	User-defined function to be modified	Must be registered in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type.*	Database Object Name, p. 4-46
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *function*.	Identifier, p. 4-189

Usage

The ALTER FUNCTION statement allows you to modify a user-defined function to tune its performance. With this statement you can modify characteristics that control how the function executes. You can also add or replace related used-defined routines (UDRs) that provide alternatives for the optimizer, which can improve performance.

All modifications take effect on the next invocation of the function.

To use the ALTER FUNCTION statement, you must be the owner of the UDR or have the DBA privilege.

Keywords That Introduce Modifications

Use the following keywords to introduce what you modify in the UDR.

Keyword	Purpose
ADD	Introduces a routine modifier that you want to add to the user-defined function
MODIFY	Introduces a routine modifier for which you want to modify a value
DROP	Introduces a routine modifier that you want to remove from the user-defined function
MODIFY EXTERNAL NAME (for external functions only)	Introduces a new location for the executable file
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be T (equivalent of using the keyword ADD to add the routine modifier). For example, both of the following statements alter the **func1** function so that it can be executed in parallel in the context of a parallelizable data query:

```
ALTER FUNCTION func1 WITH (MODIFY PARALLELIZABLE)
ALTER FUNCTION func1 WITH (ADD PARALLELIZABLE)
```

See also "Altering Routine Modifiers Example" on page 2-48.

Related Information

Related Statements: ALTER PROCEDURE, ALTER ROUTINE, CREATE FUNCTION, and CREATE PROCEDURE

For a discussion on how to create and use SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion on how to create and use external routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

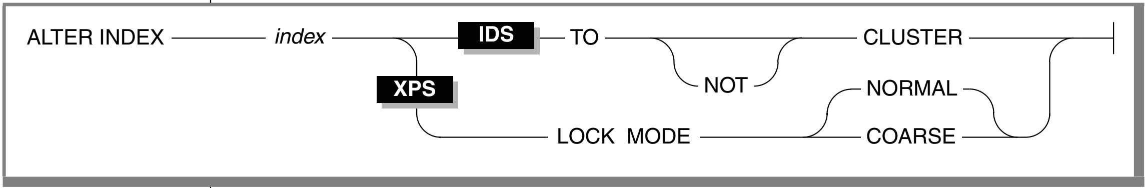
For information about how to create C UDRs, see the *IBM Informix DataBlade API Programmer's Guide*.

+

ALTER INDEX

Use the ALTER INDEX statement to change the clustering attribute or the locking mode of an existing index.

Syntax



Element	Purpose	Restrictions	Syntax
index	Name of the index to be altered	Must exist	Database Object Name, p. 4-46

Usage

ALTER INDEX is valid only on indexes created explicitly with the CREATE INDEX statement. It cannot modify indexes that were created implicitly to support constraints, and it cannot modify an index on a temporary table.

IDS

You cannot change the collating order of an existing index. If you use ALTER INDEX to modify an index after SET COLLATION has specified a non-default collating order, the SET COLLATION statement has no effect on the index. ♦

TO CLUSTER Option

The TO CLUSTER option causes the database server to reorder the rows of the physical table according to the indexed order.

The next example shows how you can use the ALTER INDEX TO CLUSTER statement to order the rows in the **orders** table physically. The CREATE INDEX statement creates an index on the **customer_num** column of the table. Then the ALTER INDEX statement causes the physical ordering of the rows.

```
CREATE INDEX ix_cust ON orders (customer_num);
ALTER INDEX ix_cust TO CLUSTER;
```

For an ascending index, TO CLUSTER puts rows in lowest-to-highest order. For a descending index, the rows are reordered in highest-to-lowest order.

When you reorder, the entire file is rewritten. This process can take a long time, and it requires sufficient disk space to maintain two copies of the table.

While a table is clustering, it is locked IN EXCLUSIVE MODE. When another process is using the table to which the index name belongs, the database server cannot execute the ALTER INDEX statement with the TO CLUSTER option; it returns an error unless lock mode is set to WAIT. (When lock mode is set to WAIT, the database server retries the ALTER INDEX statement.)

Over time, if you modify the table, you can expect the benefit of an earlier cluster to disappear because rows are added in space-available order, not sequentially. You can recluster the table to regain performance by issuing another ALTER INDEX TO CLUSTER statement on the clustered index. You do not need to drop a clustered index before you issue another ALTER INDEX TO CLUSTER statement on a currently clustered index.

XPS

If you are using Extended Parallel Server, you cannot use the CLUSTER option on STANDARD tables. ♦

TO NOT CLUSTER Option

The TO NOT CLUSTER option drops the cluster attribute on the index name without affecting the physical table. Because only one clustered index per table can exist, you must use the TO NOT CLUSTER option to release the cluster attribute from one index before you assign it to another. The following statements illustrate how to remove clustering from one index and how a second index physically reclusters the table:

```
CREATE UNIQUE INDEX ix_ord
   ON orders (order_num);

CREATE CLUSTER INDEX ix_cust
   ON orders (customer_num);
. . .
ALTER INDEX ix_cust TO NOT CLUSTER;

ALTER INDEX ix_ord TO CLUSTER;
```

The first two statements create indexes for the **orders** table and cluster the physical table in ascending order on the **customer_num** column. The last two statements recluster the physical table in ascending order on the **order_num** column.

XPS

LOCK MODE Options

Use the LOCK MODE clause to specify the locking granularity of the index.

When you use the COARSE mode, index-level locks are acquired on the index instead of item-level or page-level locks. This mode reduces the number of lock calls on an index.

The COARSE mode offers performance advantages when you know the index is not going to change; for example, when read-only operations are performed on the index.

Use the NORMAL mode to have the database server place item-level or page-level locks on the index as necessary. Use this mode when the index gets updated frequently.

When the database server executes the LOCK MODE COARSE option, it acquires an exclusive lock on the table for the duration of the ALTER INDEX statement. Any transactions currently using a lock of finer granularity must complete before the database server switches to the COARSE lock mode.

Related Information

Related statements: CREATE INDEX and CREATE TABLE

For a discussion of the performance implications of clustered indexes, see your *Performance Guide*.

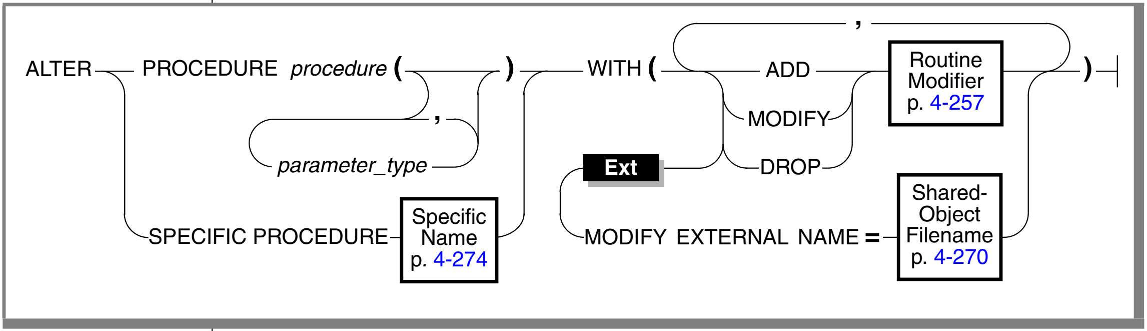
+

IDS

ALTER PROCEDURE

Use the ALTER PROCEDURE statement to change the routine modifiers or pathname of a previously defined external procedure.

Syntax



Element	Purpose	Restrictions	Syntax
procedure	User-defined procedure to modify	Must be registered in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type.*	Database Object Name, p. 4-46
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *procedure*.	Identifier, p. 4-189

Usage

The ALTER PROCEDURE statement allows you to modify an external procedure to tune its performance by modifying characteristics that control how it executes. You can also add or replace related UDRs that provide alternatives for the optimizer, which can improve performance.

To use the ALTER PROCEDURE statement, you must be the owner of the UDR or have the DBA privilege.

If the name is not unique among routines registered in the database, you must enter one or more appropriate values for *parameter_type*.

All modifications take effect on the next invocation of the procedure.

Use the following keywords to introduce the items in the external procedure that you want to modify.

Keyword	Purpose
ADD	Introduces a routine modifier that you want to add to the external procedure
MODIFY	Introduces a routine modifier for which you want to modify a value
DROP	Introduces a routine modifier that you want to remove from the external procedure
MODIFY EXTERNAL NAME (for external routines only)	Introduces a new location for the executable file, specifying a different pathname from the original
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be `T` (equivalent to using the keyword ADD to add the routine modifier). For example, both of the following statements alter the **proc1** procedure so that it can be executed in parallel in the context of a parallelizable data query:

```
ALTER PROCEDURE proc1 WITH (MODIFY PARALLELIZABLE)
ALTER PROCEDURE proc1 WITH (ADD PARALLELIZABLE)
```

See also "Altering Routine Modifiers Example" on page 2-48.

Related Information

Related Statements: ALTER FUNCTION, ALTER ROUTINE, CREATE FUNCTION, CREATE PROCEDURE, DROP PROCEDURE, and DROP ROUTINE

For a discussion on how to create and use SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion on how to create and use external routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For information about how to create C UDRs, see the *IBM Informix DataBlade API Programmer's Guide*.

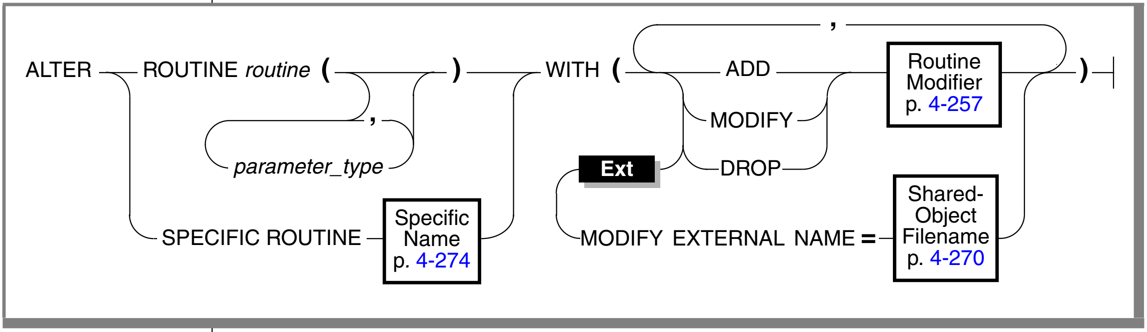
+

IDS

ALTER ROUTINE

Use the ALTER ROUTINE statement to change the routine modifiers or pathname of a previously defined user-defined routine (UDR).

Syntax



Element	Purpose	Restrictions	Syntax
routine	User-defined routine to modify	Must be registered in the database. If the name does not uniquely identify a routine, you must enter one or more appropriate values for *parameter_type.*	Database Object Name, p. 4-46
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *routine*.	Identifier, p. 4-189

Usage

The ALTER ROUTINE statement allows you to modify a previously defined UDR to tune its performance by modifying characteristics that control how the UDR executes. You can also add or replace related UDRs that provide alternatives for the optimizer, which can improve performance.

This statement is useful when you do not know whether a UDR is a user-defined function or a user-defined procedure. When you use this statement, the database server alters the appropriate user-defined procedure or user-defined function.

All modifications take effect on the next invocation of the UDR.

To use the ALTER ROUTINE statement, you must be the owner of the UDR or have the DBA privilege.

Restrictions

If the name does not uniquely identify a UDR, you must enter one or more appropriate values for *parameter_type*.

When you use this statement, the type of UDR cannot be ambiguous. The UDR that you specify must refer to either a user-defined function or a user-defined procedure. If either of the following conditions exist, the database server returns an error:

- The name (and parameters) that you specify applies to both a user-defined procedure and a user-defined function.
- The specific name that you specify applies to both a user-defined function and a user-defined procedure.

Keywords That Introduce Modifications

Use the following keywords to introduce the items in the UDR that you want to modify.

Keyword	**Purpose**
ADD	Introduces a routine modifier that you want to add to the UDR
MODIFY	Introduces a routine modifier for which you want to modify a value
DROP	Introduces a routine modifier that you want to remove from the UDR
MODIFY EXTERNAL NAME (for external UDRs only)	Introduces a new location for the executable file
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be `T` (equivalent to using the keyword ADD to add the routine modifier).

For example, both of the following statements alter the **func1** UDR so that it can be executed in parallel in the context of a parallelizable data query statement:

```
ALTER ROUTINE func1 WITH (MODIFY PARALLELIZABLE)
ALTER ROUTINE func1 WITH (ADD PARALLELIZABLE)
```

Altering Routine Modifiers Example

Suppose you have an external function **func1** that is set to handle NULL values and has a cost per invocation set to `40`. The following ALTER ROUTINE statement adjusts the settings of the function by dropping the ability to handle NULL values, tunes the **func1** by changing the cost per invocation to `20`, and indicates that the function can execute in parallel:

```
ALTER ROUTINE func1(CHAR, INT, BOOLEAN)
   WITH (
      DROP HANDLESNULLS,
      MODIFY PERCALL_COST = 20,
      ADD PARALLELIZABLE
      )
```

Because the name **func1** is not unique to the database, the data type parameters are specified so that the routine signature would be unique. If this function had a Specific Name, for example, **raise_sal**, specified when it was created, you could identify the function with the following first line:

```
ALTER SPECIFIC ROUTINE raise_sal
```

Related Information

Related Statements: ALTER FUNCTION, ALTER PROCEDURE, CREATE FUNCTION, CREATE PROCEDURE, DROP FUNCTION, DROP PROCEDURE, and DROP ROUTINE

For a discussion on how to create and use SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion on how to create and use external routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For information about how to create C UDRs, see the *IBM Informix DataBlade API Programmer's Guide*.

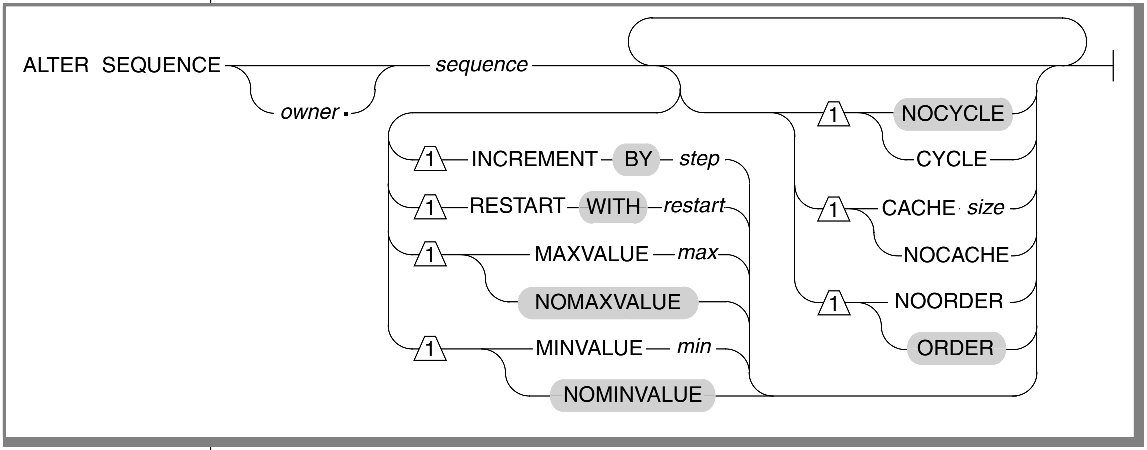
+

IDS

ALTER SEQUENCE

Use the ALTER SEQUENCE statement to modify the definition of a sequence.

Syntax



Element	Purpose	Restrictions	Syntax
max	New upper limit on values	Must be integer > CURRVAL and *restart*	Literal Number, p. 4-216
min	New lower limit on values	Must be integer < CURRVAL and *restart*	Literal Number, p. 4-216
owner	Owner of *sequence*	Cannot be changed by this statement	Owner Name, p. 4-234
restart	New first value in sequence	Must be integer in the INT8 range	Literal Number, p. 4-216
sequence	Name of an existing sequence	Must exist. Cannot be a synonym.	Identifier, p. 4-189
size	New number of values to preallocate in memory	Integer > 2 but < cardinality of values in one cycle (= |(*max* - *min*)/*step*|)	Literal Number, p. 4-216
step	New interval between values	Must be a nonzero integer	Literal Number, p. 4-216

Usage

ALTER SEQUENCE redefines an existing sequence object. It only affects subsequently generated values (and any unused values in the sequence cache).

You cannot use the ALTER SEQUENCE statement to rename a sequence nor to change the owner of a sequence.

You must be the owner, or the DBA, or else have the ALTER privilege on the sequence to modify its definition. Only elements of the sequence definition that you specify explicitly in the ALTER SEQUENCE statement are modified. An error occurs if you make contradictory changes, such as specifying both MAXVALUE and NOMAXVALUE, or both the CYCLE and NOCYCLE options.

INCREMENT BY Option

Use the INCREMENT BY option to specify a new interval between successive numbers in a sequence. The interval, or *step* value, can be a positive whole number (for ascending sequences) or a negative whole number (for descending sequences) in the INT8 range. The BY keyword is optional.

START WITH Option

Use the START WITH option to specify a new first number of the sequence. The *restart* value must be an integer within the INT8 range that is greater than or equal to the *min* value (for an ascending sequence) or that is less than or equal to the *max* value (for a descending sequence), if *min* or *max* is specified in the ALTER SEQUENCE statement. The WITH keyword is optional.

MAXVALUE or NOMAXVALUE Option

Use the MAXVALUE option to specify a new upper limit of values in the sequence. The maximum value, or *max*, must be an integer in the INT8 range that is greater than *sequence*.CURRVAL and *restart (*or greater than the *origin* in the original CREATE SEQUENCE statement, if *restart* is not specified).

Use the NOMAXVALUE option to replace the current limit with a new default maximum of `2e64` for ascending sequences or `-1` for descending sequences.

MINVALUE or NOMINVALUE Option

Use the MINVALUE option to specify a new lower limit of values in the sequence. The minimum value, or *min*, must be an integer the INT8 range that is less than *sequence*.CURRVAL and *restart (*or less than the *origin* in the original CREATE SEQUENCE statement, if *restart* is not specified).

Use the NOMINVALUE option to replace the current lower limit with a default of `1` for ascending sequences or `-(2e64)` for descending sequences.

CYCLE or NOCYCLE Option

Use the CYCLE option to continue generating sequence values after the sequence reaches the maximum (ascending) or minimum (descending) limit, to replace the NOCYCLE attribute. After an ascending sequence reaches *max*, it generates the *min* value for the next value. After a descending sequence reaches *min*, it generates the *max* value for the next sequence value.

Use the NOCYCLE option to prevent the sequence from generating more values after reaching the declared limit. Once the sequence reaches the limit, the next reference to *sequence*.NEXTVAL returns an error message.

CACHE or NOCACHE Option

Use the CACHE option to specify a new number of sequence values that are preallocated in memory for rapid access. The cache size must be a whole number in the INT range that is less than the number of values in a cycle (or less than |(*max* - *min*)/*step*|). The minimum size is 2 preallocated values.

Use NOCACHE to have no values preallocated in memory. (See also the description of SEQ_CACHE_SIZE in "CREATE SEQUENCE" on page 2-206.)

ORDER or NOORDER Option

These keywords have no effect on the behavior of the sequence. The sequence always issues values to users in the order of their requests, as if the ORDER keyword were always specified. The ORDER and NOORDER keywords are accepted by the ALTER SEQUENCE statement, however, for compatibility with implementations of sequence objects in other dialects of SQL.

Related Information

Related statements: CREATE SEQUENCE, DROP SEQUENCE, RENAME SEQUENCE, CREATE SYNONYM, DROP SYNONYM, GRANT, REVOKE, INSERT, UPDATE, and SELECT

For information about the **syssequences** system catalog table in which sequence objects are registered, see the *IBM Informix Guide to SQL: Reference*.

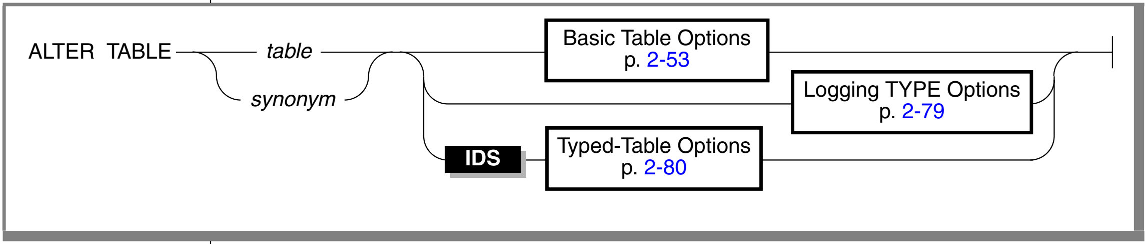
For information about initializing, generating, or reading values from a sequence, see "NEXTVAL and CURRVAL Operators" on page 4-102.

+

ALTER TABLE

Use the ALTER TABLE statement to modify the definition of a table.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Synonym for the table to be altered	Synonym and its table must exist; **USETABLENAME** is not set	Database Object Name, p. 4-46
table	Name of table to be altered	Must exist in current database	Database Object Name, p. 4-46

Usage

Altering a table on which a view depends might invalidate the view.

Warning: *The clauses available with this statement have varying performance implications. Before you undertake alter operations, check corresponding sections in your "Performance Guide" to review effects and strategies.*

You cannot alter a temporary table. You also cannot alter a violations or diagnostics table. In addition, you cannot add, drop, or modify a column if the table that contains the column has a violation table or a diagnostics table associated with it. If the **USETABLENAME** environment variable is set, you cannot specify a *synonym* for the table in the ALTER TABLE statement.

XPS

If a table has range fragmentation, only the Logging TYPE options and LOCK MODE clause are valid. All other ALTER TABLE options return an error. ♦

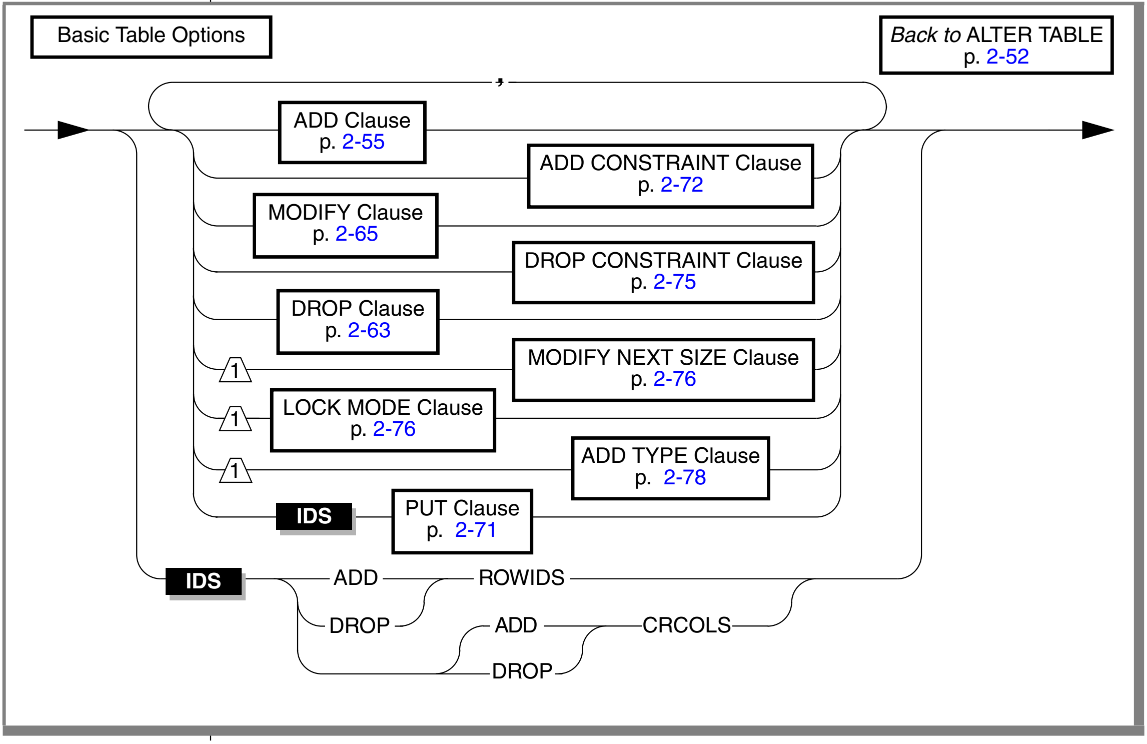
If you have a static or raw table, the only information that you can alter is the logging type of the table. That is, the Logging TYPE options are the only part of the ALTER TABLE statement that you can use.

To use ALTER TABLE, you must meet one of the following conditions:

- You must have DBA privilege on the database containing the table.
- You must own the table.
- You must have the Alter privilege on the specified table and the Resource privilege on the database where the table resides.
- To add a referential constraint, you must have the DBA or References privilege on either the referenced columns or the referenced table.
- To drop a constraint, you must have the DBA privilege or be the owner of the constraint. If you are the owner of the constraint but not the owner of the table, you must have Alter privilege on the specified table. You do not need the References privilege to drop a constraint.

Basic Table Options

The Basic Table Options segment of ALTER TABLE has the following syntax.



You can use the Basic Table Options segment to modify the schema of a table by adding, modifying, or dropping columns and constraints, or changing the extent size or locking granularity of a table. The database server performs alterations in the order that you specify. If any of the actions fails, the entire operation is cancelled.

IDS

You can also associate a table with a named ROW type or specify a new storage space to store large-object data. You can also add or drop **rowid** columns and shadow columns for Enterprise Replication. You cannot, however, specify these options in conjunction with any other alterations. ♦

IDS

Using the ADD ROWIDS Keywords

Use the ADD ROWIDS keywords to add a new column called **rowid** to a fragmented table. (Fragmented tables do not contain the hidden **rowid** column by default.) When you add a **rowid** column, the database server assigns a unique number to each row that remains stable for the life of the row. The database server creates an index that it uses to find the physical location of the row. After you add the **rowid** column, each row of the table contains an additional 4 bytes to store the **rowid** value.

Tip: *Use the ADD ROWIDS clause only on fragmented tables. In nonfragmented tables, the* ***rowid*** *column remains unchanged. It is recommended that you use primary keys as an access method rather than exploiting the* ***rowid*** *column.*

For additional information about the **rowid** column, refer to your *Administrator's Reference*.

IDS

Using the DROP ROWIDS Keywords

The DROP ROWIDS keywords can drop a **rowid** column that you added (with either the CREATE TABLE or ALTER FRAGMENT statement) to a fragmented table. You cannot drop the **rowid** column of a nonfragmented table.

IDS

Using the ADD CRCOLS Keywords

The ADD CRCOLS keywords create shadow columns, **cdrserver** and **cdrtime**, that Enterprise Replication uses for conflict resolution. These columns enable the database server to use the time-stamp or SPL conflict-resolution rule. For more information, refer to "Using the WITH CRCOLS Option" on page 2-235 and to the *IBM Informix Dynamic Server Enterprise Replication Guide*.

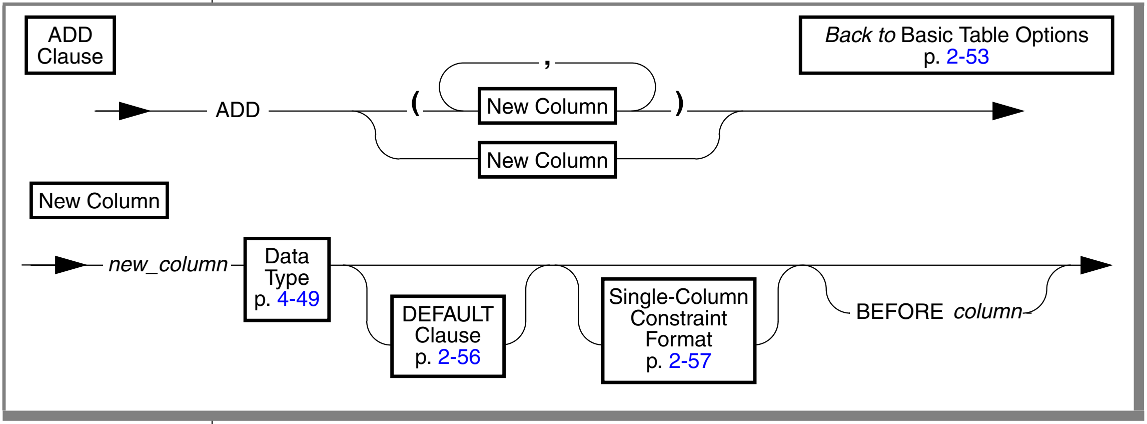
IDS

Using the DROP CRCOLS Keywords

Use the DROP CRCOLS keywords to drop the **cdrserver** and **cdrtime** shadow columns. You cannot drop these columns if Enterprise Replication is in use.

ADD Clause

Use the ADD clause to add a column to a table.



Element	Purpose	Restrictions	Syntax
column	Name of column before which *new_column* is to be placed	Must already exist in the table.	Identifier, p. 4-189
new_column	Name of column that you are adding	You cannot add a serial column if the table contains data.	Identifier, p. 4-189

The following restrictions apply to the ADD clause:

- You cannot add a serial column to a table if the table contains data

XPS

- In Extended Parallel Server, you cannot add a column to a table that has a bit-mapped index. ♦

Using the BEFORE Option

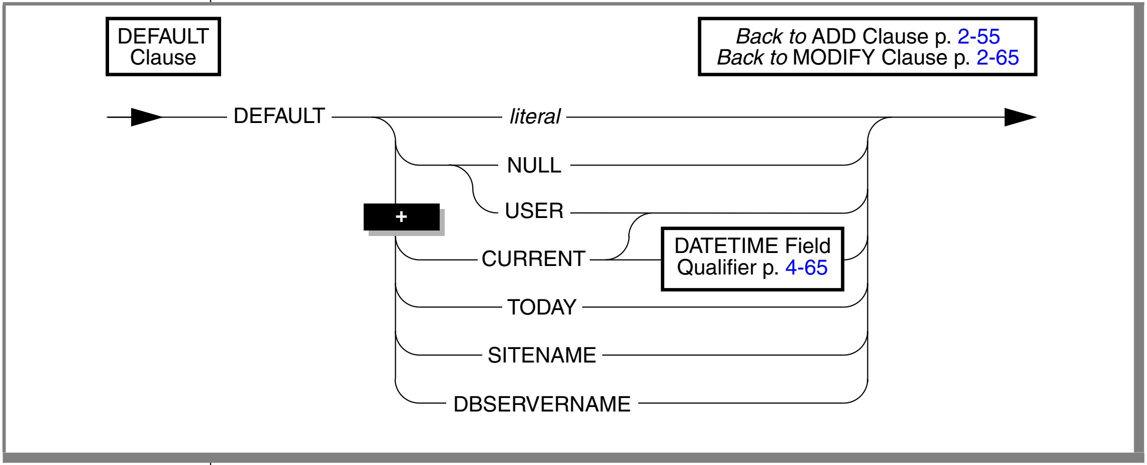
The BEFORE option specifies the column before which to add the new column(s). In the following example, the BEFORE option directs the database server to add the **item_weight** column before the **total_price** column:

```
ALTER TABLE items
   ADD (item_weight DECIMAL(6,2) NOT NULL
      BEFORE total_price)
```

If you do not include the BEFORE option, the database server adds the new column or list of columns to the end of the table definition by default.

DEFAULT Clause

Use the DEFAULT clause to specify at value that the database server should insert in a column when an explicit value for the column is not specified.


DEFAULT Clause
Back to ADD Clause p. 2-55
Back to MODIFY Clause p. 2-65
DEFAULT
literal
NULL
USER
+
CURRENT
DATETIME Field Qualifier p. 4-65
TODAY
SITENAME
DBSERVERNAME

Element	Purpose	Restrictions	Syntax
literal	Literal default value for the column	Must be appropriate for the data type of the column. See "Using a Literal as a Default Value" on page 2-218.	Expression, p. 4-67

You cannot specify a default value for serial columns. If the table that you are altering already has rows in it when you add a column that contains a default value, the database server inserts the default value for all pre-existing rows.

The following example adds a column to the **items** table. In **items**, the new column **item_weight** has a literal default value:

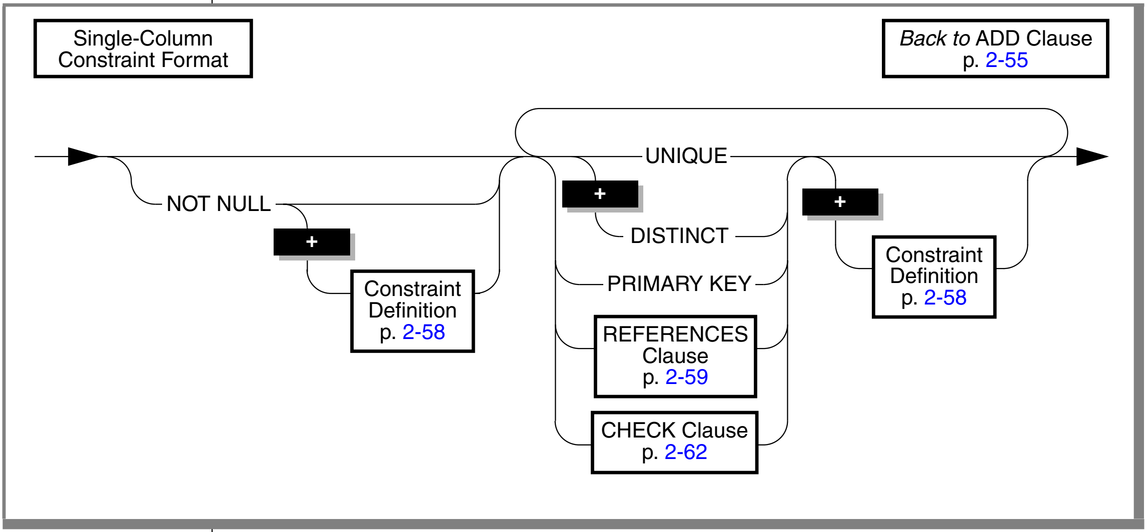
```
ALTER TABLE items
   ADD item_weight DECIMAL (6, 2) DEFAULT 2.00
   BEFORE total_price
```

In this example, each existing row in the **items** table has a default value of `2.00` for the **item_weight** column.

For more information about the options of the DEFAULT clause, refer to "DEFAULT Clause" on page 2-217.

Single-Column Constraint Format

Use the Single-Column Constraint Format to associate constraints with a specified column.


Single-Column Constraint Format
Back to ADD Clause p. 2-55
NOT NULL
+
Constraint Definition p. 2-58
UNIQUE
+
DISTINCT
PRIMARY KEY
REFERENCES Clause p. 2-59
CHECK Clause p. 2-62
+
Constraint Definition p. 2-58

You cannot specify a primary-key or unique constraint on a new column if the table contains data. In the case of a unique constraint, however, the table can contain a *single* row of data. When you want to add a column with a primary-key constraint, the table must be empty when you issue the ALTER TABLE statement.

The following rules apply when you place primary-key or unique constraints on existing columns:

- When you place a primary-key or unique constraint on a column or set of columns, the database server creates an internal B-tree index on the constrained column or set of columns unless a user-created index was defined on the column or set of columns.
- When you place a primary-key or unique constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares the index. If the existing index allows duplicates, however, the database server returns an error. You must then drop the existing index before you add the constraint.
- When you place a primary-key or unique constraint on a column or set of columns, and a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible), and the index is shared.

You cannot place a unique constraint on a BYTE or TEXT column, nor can you place referential constraints on columns of these types. A check constraint on a BYTE or TEXT column can check only for IS NULL, IS NOT NULL, or LENGTH.

When you place a referential constraint on a column or set of columns, and an index already exists on that column or set of columns, the index is upgraded to unique (if possible) and the index is shared.

Using Not-Null Constraints with ADD

If a table contains data, when you add a column with a not-null constraint you must also include a DEFAULT clause. If the table is empty, however, you can add a column and apply only the not-null constraint. The following statement is valid whether or not the table contains data:

```
ALTER TABLE items
   ADD (item_weight DECIMAL(6,2)
   DEFAULT 2.0 NOT NULL
      BEFORE total_price)
```

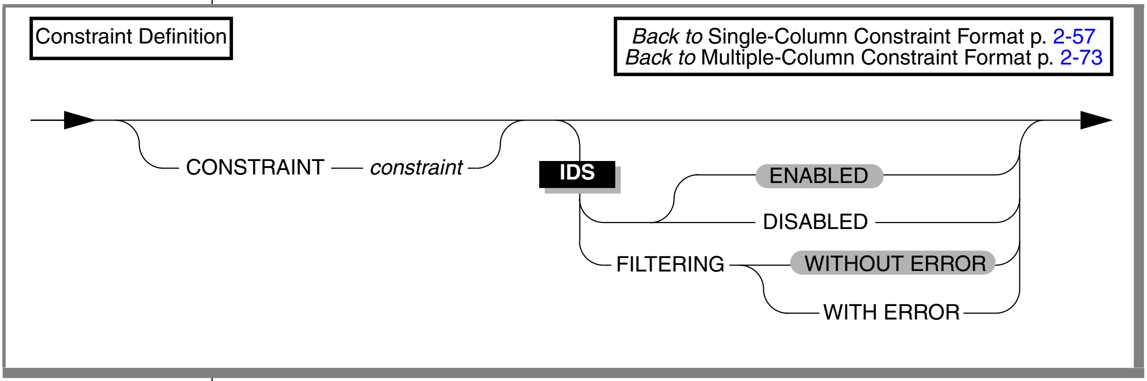
Constraint Definition

IDS

In Dynamic Server, use the Constraint Definition portion of the ALTER TABLE statement to declare the name of a constraint and to set the mode of the constraint to disabled, enabled, or filtering. ♦

XPS

In Extended Parallel Server, use the Constraint Definition portion of the ALTER TABLE statement to declare the name of a constraint. ♦

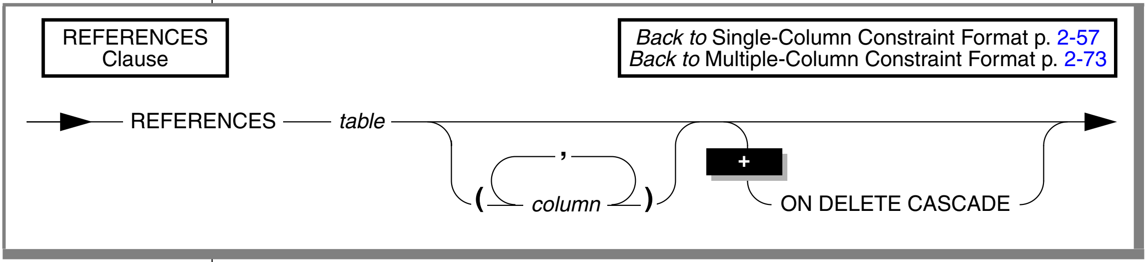

Constraint Definition
Back to Single-Column Constraint Format p. 2-57
Back to Multiple-Column Constraint Format p. 2-73
CONSTRAINT
constraint
IDS
ENABLED
DISABLED
FILTERING
WITHOUT ERROR
WITH ERROR

Element	Purpose	Restrictions	Syntax
constraint	Name declared here to the constraint	Must be unique.	Identifier, p. 4-189

For more information about constraint-mode options, see "Choosing a Constraint-Mode Option" on page 2-230.

REFERENCES Clause

The REFERENCES clause has the following syntax.


REFERENCES Clause
Back to Single-Column Constraint Format p. 2-57
Back to Multiple-Column Constraint Format p. 2-73
REFERENCES
table
(
,
column
)
+
ON DELETE CASCADE

Element	Purpose	Restrictions	Syntax
column	Referenced column in the referenced table	See "Restrictions on Referential Constraints" on page 2-60.	Identifier, p. 4-189
table	The referenced table	The referenced and the referencing tables must reside in the same database.	Database Object Name, p. 4-46

The REFERENCES clause allows you to place a foreign-key reference on a column. The referenced column can be in the same table as the referencing column, or in a different table in the same database.

If the referenced table is different from the referencing table, the default is the primary-key column. If the referenced table is the same as the referencing table, there is no default.

Restrictions on Referential Constraints

You must have the REFERENCES privilege to create a referential constraint.

The following restrictions apply to the *column* that is specified (the referenced column) in the REFERENCES clause:

- The referenced and referencing tables must be in the same database.
- The referenced column (or set of columns) must have a unique or primary-key constraint.
- The referencing and referenced columns must be the same data type.

  (The only exception is that a referencing column must be an integer data type if the referenced column is a serial data type.)
- You cannot place a referential constraint on a BYTE or TEXT column.
- IDS Constraints uses the collation in effect at their time of creation. ♦
- A column-level REFERENCES clause can include only a single column name.
- The maximum number of columns in a table-level REFERENCES clause is 16.
- IDS The total length of the columns in a table-level REFERENCES clause cannot exceed 390 bytes. ♦
- XPS The total length of the columns in a table-level REFERENCES clause cannot exceed 255 bytes. ♦

Default Values for the Referenced Column

If the referenced table is different from the referencing table, you do not need to specify the referenced column; the default column is the primary-key column (or columns) of the referenced table. If the referenced table is the same as the referencing table, you must specify the referenced column.

The following example creates a new column in the **cust_calls** table, **ref_order**. The **ref_order** column is a foreign key that references the **order_num** column in the **orders** table.

```
ALTER TABLE cust_calls
   ADD ref_order INTEGER
   REFERENCES orders (order_num)
   BEFORE user_id
```

When you place a referential constraint on a column or set of columns, and a duplicate or unique index already exists on that column or set of columns, the index is shared.

Using the ON DELETE CASCADE Option

Use the ON DELETE CASCADE option if you want rows deleted in the child table when corresponding rows are deleted in the parent table. If you do not specify cascading deletes, the default behavior of the database server prevents you from deleting data in a table if other tables reference it.

If you specify this option, later when you delete a row in the parent table, the database server also deletes any rows associated with that row (foreign keys) in a child table. The advantage of the ON DELETE CASCADE option is that it allows you to reduce the quantity of SQL statements needed to perform delete actions.

For example, in the **stores_demo** database, the **stock** table contains the **stock_num** column as a primary key. The **catalog** table refers to the **stock_num** column as a foreign key. The following ALTER TABLE statements drop an existing foreign-key constraint (without cascading delete) and add a new constraint that specifies cascading deletes:

```
ALTER TABLE catalog DROP CONSTRAINT aa

ALTER TABLE catalog ADD CONSTRAINT
   (FOREIGN KEY (stock_num, manu_code) REFERENCES stock
   ON DELETE CASCADE CONSTRAINT ab)
```

With cascading deletes specified on the child table, in addition to deleting a stock item from the **stock** table, the delete cascades to the catalog table that is associated with the stock_num foreign key. This cascading delete works only if the **stock_num** that you are deleting was not ordered; otherwise, the constraint from the **items** table would disallow the cascading delete. For more information, see "Restrictions on DELETE When Tables Have Cascading Deletes" on page 2-346.

If a table has a trigger with a DELETE trigger event, you cannot define a cascading-delete referential constraint on that table. You receive an error when you attempt to add a referential constraint that specifies ON DELETE CASCADE to a table that has a delete trigger.

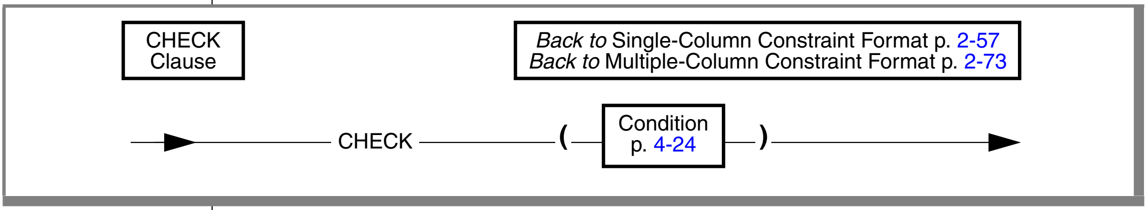
For information about syntax restrictions and locking implications when you delete rows from tables that have cascading deletes, see "Considerations When Tables Have Cascading Deletes" on page 2-346.

Locks Held During Creation of a Referential Constraint

When you create a referential constraint, the database server places an exclusive lock on the referenced table. The lock is released after you finish with the ALTER TABLE statement or at the end of a transaction (if you are altering the table in a database that uses transaction logging).

CHECK Clause

A check constraint designates a condition that must be met *before* data can be inserted into a column.


CHECK Clause
Back to Single-Column Constraint Format p. 2-57
Back to Multiple-Column Constraint Format p. 2-73
CHECK
(
Condition p. 4-24
)

During an insert or update, if a row returns *false* for any check constraint defined on a table, the database server returns an error. No error is returned, however, if a row returns NULL for a check constraint. In some cases, you might want to use both a check constraint and a NOT NULL constraint.

Check constraints are defined using *search conditions*. The search condition cannot contain user-defined routines, subqueries, aggregates, host variables, or rowids. In addition, the condition cannot contain the variant built-in functions CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY.

The check constraint cannot include columns in different tables. When you are using the ADD or MODIFY clause, the check constraint cannot depend upon values in other columns of the same table.

The next example adds a new **unit_price** column to the **items** table and includes a check constraint to ensure that the entered value is greater than 0:

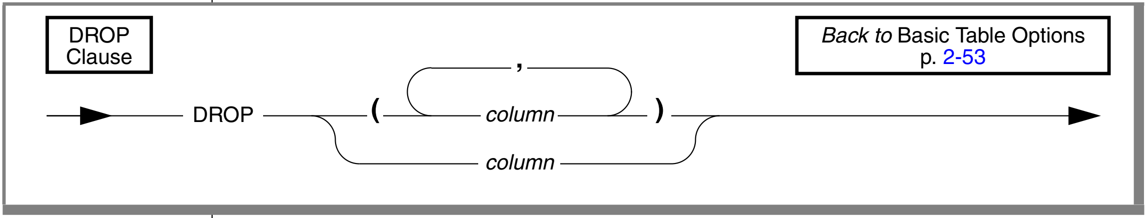
```
ALTER TABLE items
   ADD (unit_price MONEY (6,2) CHECK (unit_price > 0) )
```

To create a constraint that checks values in more than one column, use the ADD CONSTRAINT clause. The following example builds a constraint on the column that was added in the previous example. The check constraint now spans two columns in the table.

```
ALTER TABLE items ADD CONSTRAINT CHECK (unit_price < total_price)
```

DROP Clause

Use the DROP clause to drop one or more columns from a table.



Element	Purpose	Restrictions	Syntax
column	Name of a column to be dropped	Must exist in the table. If any fragment expression references it, or if it is the last column in the table, *column* cannot be dropped.	Identifier, p. 4-189

You cannot issue an ALTER TABLE DROP statement that would drop every column from the table. At least one column must remain in the table.

You cannot drop a column that is part of a fragmentation strategy.

XPS

In Extended Parallel Server, you cannot use the DROP clause if the table has a dependent GK index. ♦

How Dropping a Column Affects Constraints

When you drop a column, all constraints on that column are also dropped:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.

- All check constraints that reference the column are dropped.
- If the column is part of a multiple-column primary-key or unique constraint, the constraints placed on the multiple columns are also dropped. This action, in turn, triggers the dropping of all referential constraints that reference the multiple columns.

Because any constraints that are associated with a column are dropped when the column is dropped, the structure of other tables might also be altered when you use this clause. For example, if the dropped column is a unique or primary key that is referenced in other tables, those referential constraints also are dropped. Therefore the structure of those other tables is also altered.

How Dropping a Column Affects Triggers

In general, when you drop a column from a table, the triggers based on that table remain unchanged. If the column that you drop appears in the action clause of a trigger, however, dropping the column can invalidate the trigger. The following statements illustrate the possible effects on triggers:

```
CREATE TABLE tab1 (i1 int, i2 int, i3 int);
CREATE TABLE tab2 (i4 int, i5 int);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
   BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 DROP i4;
```

After the ALTER TABLE statement, **tab2** has only one column. The **col1trig** trigger is invalidated because the action clause as it is currently defined with values for two columns cannot occur.

If you drop a column that occurs in the triggering column list of an UPDATE trigger, the database server drops the column from the triggering column list. If the column is the only member of the triggering column list, the database server drops the trigger from the table. For more information on triggering columns in an UPDATE trigger, see "CREATE TRIGGER" on page 2-269.

If a trigger is invalidated when you alter the underlying table, drop and then re-create the trigger.

How Dropping a Column Affects Views

When you drop a column from a table, the views based on that table remain unchanged. That is, the database server does not automatically drop the corresponding columns from associated views.

The view is not automatically dropped because ALTER TABLE can change the order of columns in a table by dropping a column and then adding a new column with the same name. In this case, views based on the altered table continue to work, but retain their original sequence of columns.

If a view is invalidated when you alter the underlying table, you must rebuild the view.

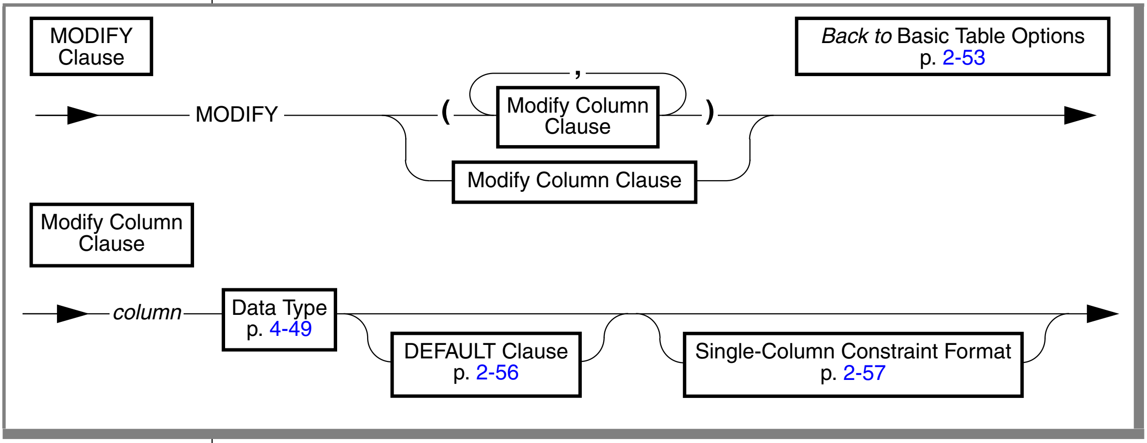
XPS

How Dropping a Column Affects a Generalized-Key Index

In Extended Parallel Server, if you drop a column from a table that has a dependent GK index, all GK indexes on the table that refer to the dropped column are dropped. Any GK indexes on other tables that refer to the dropped column are also dropped.

MODIFY Clause

Use the MODIFY clause to change the data type, length, or default value of a column, or to allow or disallow NULL values in a column.



Element	Purpose	Restrictions	Syntax
column	Column to modify	Must exist in table. Cannot be a collection data type.	Identifier, p. 4-189

XPS

In Extended Parallel Server, you cannot use the MODIFY clause if the table has a dependent GK index. ♦

IDS

You cannot change the data type of a column to a collection or a row type. ♦

When you modify a column, *all* attributes previously associated with that column (that is, default value, single-column check constraint, or referential constraint) are dropped. When you want certain attributes of the column to remain, such as PRIMARY KEY, you must re-specify those attributes.

For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, but you want to keep the default value (in this case, `1`) and the NOT NULL column attribute, you can issue this statement:

```
ALTER TABLE items MODIFY (quantity SMALLINT DEFAULT 1 NOT NULL)
```

Tip: *Both attributes are specified again in the MODIFY clause.*

When you change the data type of a column, the database server does not perform the modification in-place. The next example (for Dynamic Server only) changes a VARCHAR(15) column to an LVARCHAR(3072) column:

```
ALTER TABLE stock MODIFY (description LVARCHAR(3072))
```

When you modify a column that has column constraints associated with it, the following constraints are dropped:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.
- If the modified column is part of a multiple-column primary-key or unique constraint, all referential constraints that reference the multiple columns also are dropped.

For example, if you modify a column that has a unique constraint, the unique constraint is dropped. If this column was referenced by columns in other tables, those referential constraints are also dropped. In addition, if the column is part of a multiple-column primary-key or unique constraint, the multiple-column constraints are not dropped, but any referential constraints placed on the column by other tables *are* dropped.

For another example, suppose that a column is part of a multiple-column primary-key constraint. This primary key is referenced by foreign keys in two other tables. When this column is modified, the multiple-column primary-key constraint is not dropped, but the referential constraints placed on it by the two other tables *are* dropped.

Using the MODIFY Clause in Different Situations

The characteristics of the object you are attempting to modify can affect how you handle your modifications.

Altering BYTE and TEXT Columns

You can use the MODIFY clause to change a BYTE column to a TEXT column, and vice versa. You cannot use the MODIFY clause, however, to change a BYTE or TEXT column to any other type of column, and vice versa.

IDS

You can also use the MODIFY clause to change a BYTE column to a BLOB column and a TEXT column to a CLOB column. ♦

Altering the Next Serial Number

You can use the MODIFY clause to reset the next value of a serial column. You cannot set the next value below the current maximum value in the column because that action can cause the database server to generate duplicate numbers. You can set the next value, however, to any value higher than the current maximum, which creates gaps in the sequence.

The following example sets the next serial number to 1000:

```
ALTER TABLE my_table MODIFY (serial_num serial (1000))
```

As an alternative, you can use the INSERT statement to create a gap in the sequence of a serial column. For more information, see "Inserting Values into Serial Columns" on page 2-495.

IDS

Altering the Next Serial Number of a Typed Table

You can set the initial serial number or modify the next serial number for a row-type field with the MODIFY clause of the ALTER TABLE statement. (You cannot set the start number for a serial field when you create a row type.)

Suppose you have row types **parent**, **child1**, **child2**, and **child3**.

```
CREATE ROW TYPE parent (a int);
CREATE ROW TYPE child1 (s serial) UNDER parent;
CREATE ROW TYPE child2 (b float, s8 serial8) UNDER child1;
CREATE ROW TYPE child3 (d int) UNDER child2;
```

You then create corresponding typed tables:

```
CREATE TABLE OF TYPE parent;
CREATE TABLE OF TYPE child1 UNDER parent;
CREATE TABLE OF TYPE child2 UNDER child1;
CREATE TABLE OF TYPE child3 UNDER child2;
```

To change the next SERIAL and SERIAL8 numbers to 75, you can enter the following command:

```
ALTER TABLE child3tab MODIFY (s serial(75), s8 serial8(75))
```

When the ALTER TABLE statement executes, the database server updates corresponding serial columns in the **child1**, **child2**, and **child3** tables.

Altering the Structure of Tables

When you use the MODIFY clause, you can also alter the structure of other tables. If the modified column is referenced by other tables, those referential constraints are dropped. You must add those constraints to the referencing tables again, using the ALTER TABLE statement.

When you change the data type of an existing column, all data is converted to the new data type, including numbers to characters and characters to numbers (if the characters represent numbers). The following statement changes the data type of the **quantity** column:

```
ALTER TABLE items MODIFY (quantity CHAR(6))
```

When a primary-key or unique constraint exists, however, conversion takes place only if it does not violate the constraint. If a data type conversion would result in duplicate values (by changing FLOAT to SMALLFLOAT, for example, or by truncating CHAR values), the ALTER TABLE statement fails.

Modifying Tables for NULL Values

You can modify an existing column that formerly permitted NULLs to disallow NULLs, provided that the column contains no NULL values. To do this, specify MODIFY with the same column name and data type and the NOT NULL keywords. Those keywords create a not-null constraint on the column.

You can modify an existing column that did not permit NULLs to permit-NULLs. To do this, specify MODIFY with the column name and the existing data type, and omit the NOT NULL keywords. The omission of the NOT NULL keywords drops the not-null constraint on the column. If a unique index exists on the column, you can remove it using the DROP INDEX statement.

An alternative method of permitting NULLs in an existing column that did not permit NULLs is to use the DROP CONSTRAINT clause to drop the not-null constraint on the column.

IDS

Adding a Constraint When Existing Rows Violate the Constraint

If you use the MODIFY clause to add a constraint in the enabled mode and receive an error message because existing rows would violate the constraint, take the following steps to add the constraint successfully:

1. Add the constraint in the disabled mode.

   Issue the ALTER TABLE statement again, but this time specify the DISABLED keyword in the MODIFY clause.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
3. Issue the SET CONSTRAINTS statement to switch the database object mode of the constraint to the enabled mode.

   When you issue this statement, existing rows in the target table that violate the constraint are duplicated in the violations table; however, you receive an integrity-violation error message, and the constraint remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table.

   You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the constraint.
6. After you fix all the nonconforming rows in the target table, issue the SET statement again to enable the constraint that were disabled.

   Now the constraint is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint.

XPS

How Modifying a Column Affects a Generalized-Key Index

In Extended Parallel Server, when you modify a column, all GK indexes that reference the column are dropped if the column is used in the GK index in a way that is incompatible with the new data type of the column.

For example, if a numeric column is changed to a character column, any GK indexes involving that column are dropped if they involve arithmetic expressions.

How Modifying a Column Affects Triggers

If you modify a column that appears in the triggering column list of an UPDATE trigger, the trigger is unchanged.

When you modify a column in a table, the triggers based on that table remain unchanged, but the column modification might invalidate the trigger.

The following statements illustrate the possible affects on triggers:

```
CREATE TABLE tab1 (i1 int, i2 int, i3 int);
CREATE TABLE tab2 (i4 int, i5 int);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
   BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 MODIFY i4 char;
```

After the ALTER TABLE statement, column **i4** accepts only character values. Because character columns accept only values enclosed in quotation marks, the action clause of the **col1trig** trigger is invalidated.

If a trigger is invalidated when you modify the underlying table, drop and then re-create the trigger.

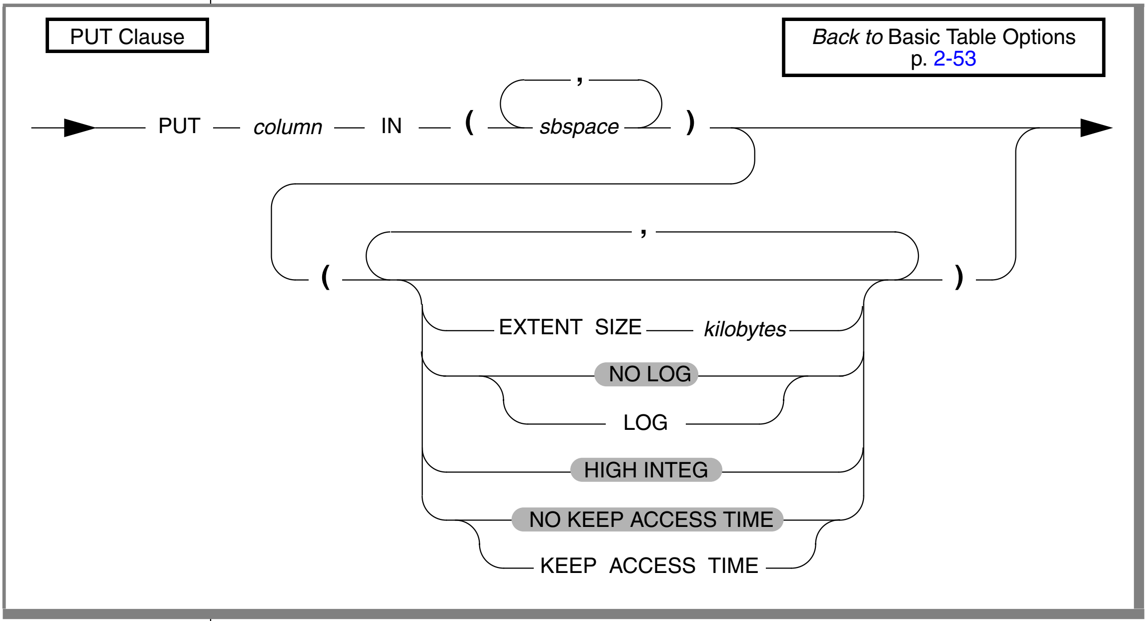
How Modifying a Column Affects Views

When you modify a column in a table, the views based on that table remain unchanged. If a view is invalidated when you alter the underlying table, you must rebuild the view.

IDS

PUT Clause

Use the PUT clause to specify the storage space (an sbspace) for a column that contains smart large objects. This clause can specify storage characteristics for a new column or replace the storage characteristics of an existing column.



Element	Purpose	Restrictions	Syntax
column	Column to store in the specified sbspace	Must contain a user-defined, or complex, or BLOB, or CLOB data type.	Identifier, p. 4-189
kilobytes	Number of kilobytes to allocate for the extent size	Must be an integer value.	Literal Number, p. 4-216
sbspace	Name of an area of storage for smart large objects	The sbspace must exist.	Identifier, p. 4-189

When you modify the storage characteristics of a column, *all* attributes previously associated with the storage space for that column are dropped. When you want certain attributes to remain, you must respecify those attributes. For example, to retain logging, you must respecify the LOG keyword.

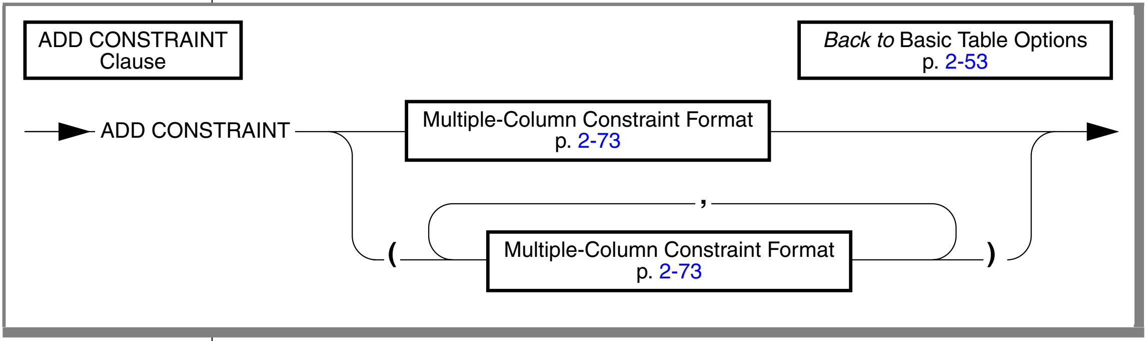
The format *column.field* is not valid here. That is, the smart large object that you are storing cannot be one field of a ROW type.

When you modify the storage characteristics of a column that holds smart large objects, the database server does not alter smart large objects that already exist, but applies the new storage characteristics to only those smart large objects that are inserted after the ALTER TABLE statement takes effect.

For more information on the available storage characteristics, refer to the counterpart of this section in the CREATE TABLE statement, "PUT Clause" on page 2-249. For a discussion of large-object characteristics, refer to "Large-Object Data Types" on page 4-57.

ADD CONSTRAINT Clause

Use the ADD CONSTRAINT clause to specify a constraint on a new or existing column or on a set of columns.


ADD CONSTRAINT Clause
Back to Basic Table Options p. 2-53
ADD CONSTRAINT
Multiple-Column Constraint Format p. 2-73
,
(
Multiple-Column Constraint Format p. 2-73
)

For example, to add a unique constraint to the **fname** and **lname** columns of the customer table, use the following statement:

```
ALTER TABLE customer ADD CONSTRAINT UNIQUE (lname, fname)
```

To declare a name for the constraint, change the preceding statement:

```
ALTER TABLE customer
   ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust
```

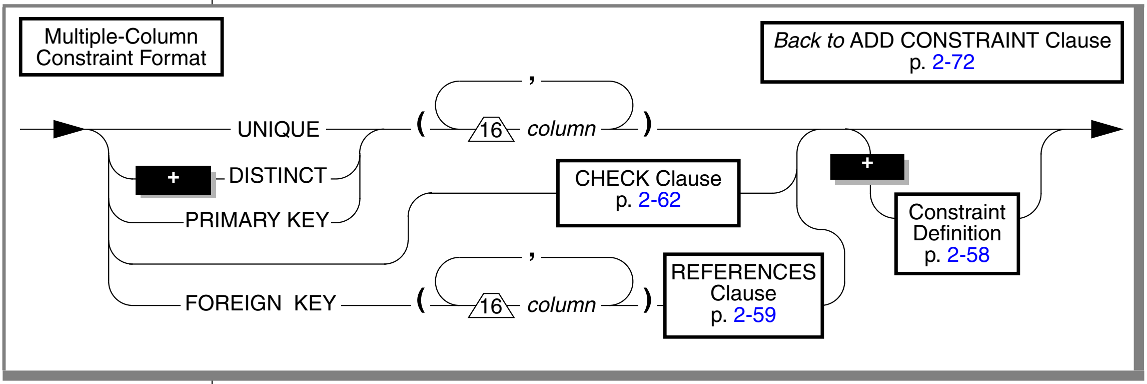
When you do not specify a name for a new constraint, the database server provides one. You can find the name of the constraint in the **sysconstraints** system catalog table. For more information about the **sysconstraints** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

IDS

When you add a constraint, the collating order must be the same as when the table was created. ♦

Multiple-Column Constraint Format

Use the Multiple-Column Constraint Format option to assign a constraint to one column or a set of columns.


Multiple-Column Constraint Format
Back to ADD CONSTRAINT Clause p. 2-72
UNIQUE
+
DISTINCT
PRIMARY KEY
FOREIGN KEY
(
,
16
column
)
CHECK Clause p. 2-62
REFERENCES Clause p. 2-59
+
Constraint Definition p. 2-58

Element	Purpose	Restrictions	Syntax
column	A column on which the constraint is placed	No more than 16 columns.	Identifier, p. 4-216

A multiple-column constraint has these restrictions:

- It can include no more than 16 column names.
- IDS The total length of the list of columns cannot exceed 390 bytes. ♦
- XPS The total length of the list of columns cannot exceed 255 bytes. ♦

You can declare a name for the constraint and set its mode by means of "Constraint Definition" on page 2-58.

Adding a Primary-Key or Unique Constraint

When you place a primary-key or unique constraint on a column or set of columns, those columns must contain unique values. The database server checks for existing constraints and indexes:

- If a user-created unique index already exists on that column or set of columns, the constraint shares the index.
- If a user-created index that allows duplicates already exists on that column or set of columns, the database server returns an error.

  In this case, you must drop the existing index before adding the primary-key or unique constraint.
- If a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.
- If no referential constraint or user-created index exists on that column or set of columns, the database server creates an internal B-tree index on the specified columns.

When you place a referential constraint on a column or set of columns, and an index already exists on that column or set of columns, the index is shared.

If you own the table or have the Alter privilege on the table, you can create a check, primary-key, or unique constraint on the table and specify yourself as the owner of the constraint. To add a referential constraint, you must have the References privilege on either the referenced columns or the referenced table. When you have the DBA privilege, you can create constraints for other users.

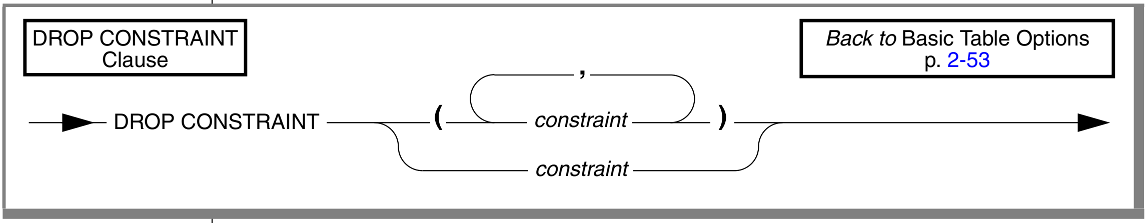
IDS

Recovery from Constraint Violations

If you use the ADD CONSTRAINT clause to add a constraint in the enabled mode, you receive an error message because existing rows would violate the constraint. For a procedure to add the constraint successfully, see "Adding a Constraint When Existing Rows Violate the Constraint" on page 2-69.

DROP CONSTRAINT Clause

Use the DROP CONSTRAINT clause to drop a named constraint.



Element	Purpose	Restrictions	Syntax
constraint	Constraint to be dropped	Must exist.	Database Object Name, p. 4-46

To drop an existing constraint, specify the DROP CONSTRAINT keywords and the name of the constraint. Here is an example of dropping a constraint:

```
ALTER TABLE manufact DROP CONSTRAINT con_name
```

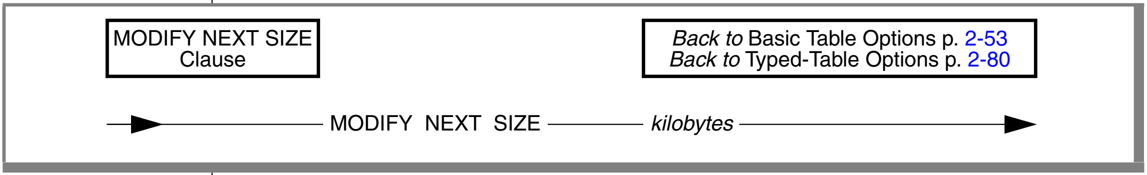
If no name is specified when the constraint is created, the database server generates the name. You can query the **sysconstraints** system catalog table for the name and owner of a constraint. For example, to find the name of the constraint placed on the **items** table, you can issue the following statement:

```
SELECT constrname FROM  sysconstraints
   WHERE tabid = (SELECT tabid FROM systables
      WHERE tabname = 'items')
```

When you drop a primary-key or unique constraint that has a corresponding foreign key, the referential constraints are dropped. For example, if you drop the primary-key constraint on the **order_num** column in the **orders** table and **order_num** exists in the **items** table as a foreign key, that referential relationship is also dropped.

MODIFY NEXT SIZE Clause

Use the MODIFY NEXT SIZE clause to change the size of new extents.


MODIFY NEXT SIZE Clause
Back to Basic Table Options p. 2-53
Back to Typed-Table Options p. 2-80
MODIFY NEXT SIZE
kilobytes

Element	Purpose	Restrictions	Syntax
kilobytes	Length (in kilobytes) assigned here for the next extent for this table	Minimum length is four times the disk-page size on your system.	Expression, p. 4-67

For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes. The maximum length is equal to the chunk size. The following example specifies an extent size of 32 kilobytes:

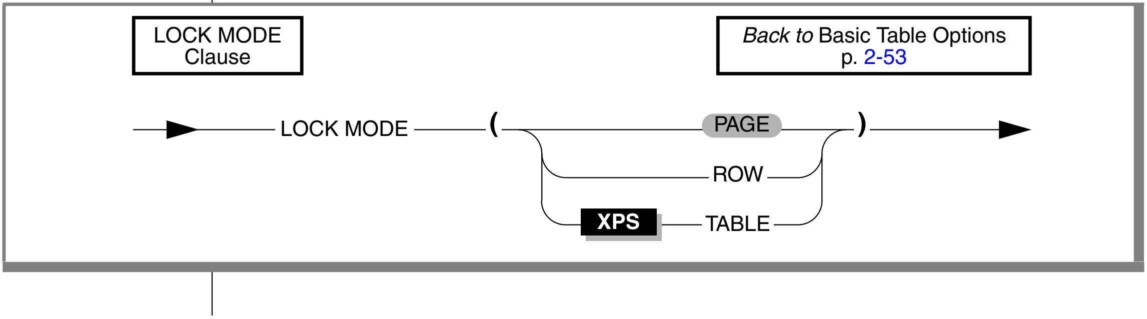
```
ALTER TABLE customer MODIFY NEXT SIZE 32
```

When you use this clause, the size of existing extents does not change. You cannot change the size of existing extents without unloading all of the data.

To change the size of existing extents, you must unload all the data, modify the extent and next-extent sizes in the CREATE TABLE statement of the database schema, re-create the database, and reload the data. For information about how to optimize extents, see your *Administrator's Guide*.

LOCK MODE Clause

Use the LOCK MODE keywords to change the locking granularity of a table.



The following table describes the locking-granularity options available.

Granularity	Purpose
PAGE	Obtains and releases one lock on a whole page of rows This is the default locking granularity. Page-level locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all the rows. For example, if you are processing the contents of a table in the same order as its cluster index, page locking is especially appropriate.
ROW	Obtains and releases one lock per row Row-level locking provides the highest level of concurrency. If you are using many rows at one time, the lock-management overhead can become significant. You can also exceed the maximum number of locks available, depending on the configuration of your database server.
TABLE (XPS only)	Places a lock on the entire table This type of lock reduces update concurrency in comparison to row and page locks. A table lock reduces the lock-management overhead for a table. Multiple read-only transactions can still access the table.

IDS

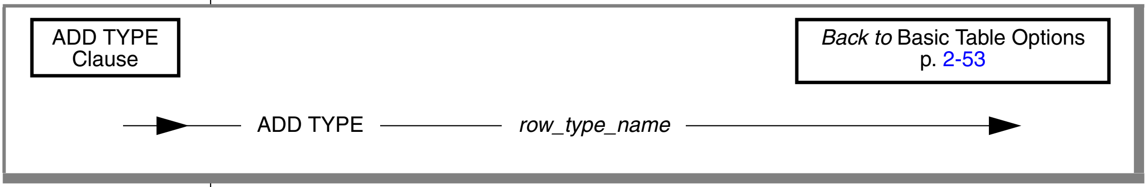
Precedence and Default Behavior

The LOCK MODE setting in an ALTER TABLE statement takes precedence over the settings of the **IFX_DEF_TABLE_LOCKMODE** environment variable and the DEF_TABLE_LOCKMODE configuration parameter. For information about the **IFX_DEF_TABLE_LOCKMODE** environment variable, refer to the *IBM Informix Guide to SQL: Reference*. For information about the DEF_TABLES_LOCKMODE configuration parameter, refer to the *IBM Informix Dynamic Server Administrator's Reference*.

IDS

ADD TYPE Clause

Use the ADD TYPE clause to convert a table that is not based on a named ROW data type into a typed table.



Element	Purpose	Restrictions	Syntax
row_type_name	Name of the row type being added to the table	The field types of this ROW type must match the column types of the table.	Data Type, p. 4-49

To add a named ROW type to a table, all of the following must be true:

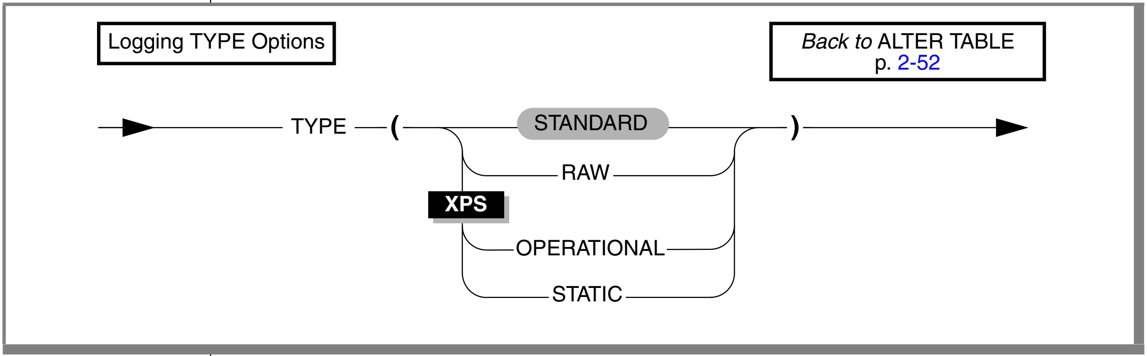
- The named ROW type already exists.
- The named ROW type fields match the column types in the table.
- You have the Usage privilege on the table.

When you use the ADD TYPE clause, you assign a named ROW data type to a table whose columns match the fields of the ROW type. The table cannot be a fragmented table that has rowids.

You cannot combine the ADD TYPE clause with any clause that changes the structure of the table. No other ADD, DROP, or MODIFY clause is valid in the same ALTER TABLE statement that has the ADD TYPE clause. The ADD TYPE clause does not allow you to change column data types. (To change the data type of a column, use the MODIFY clause.)

Logging TYPE Options

Use the Logging TYPE options to specify that the table have particular characteristics that can improve various bulk operations on it.



Other than the default option (STANDARD) that is used for online transaction processing (OLTP) databases, these Logging TYPE options are used primarily to improve performance in data warehousing databases.

A table can have any of the following logging characteristics.

Option	Purpose
STANDARD	Logging table that allows rollback, recovery, and restoration from archives. This is the default. Use this type for recovery and constraints functionality on OLTP databases.
RAW	Nonlogging table that cannot have indexes or referential constraints but can be updated. Use this type for quickly loading data. In XPS, raw tables take advantage of light appends and avoid the overhead of logging, checking constraints, and building indexes.
OPERATIONAL (XPS only)	Logging table that uses light appends and cannot be restored from archive. Use this type on tables that are refreshed frequently. Light appends allow the quick addition of many rows.
STATIC (XPS only)	Nonlogging table that can contain index and referential constraints but cannot be updated. Use this type for read-only operations because there is no logging or locking overhead.

Warning: *Use raw tables for fast loading of data. It is recommended that you alter the logging type to STANDARD and perform a level-0 backup before you use the table in a transaction or modify the data within the table. If you must use a raw table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.*

For more information on these logging types of tables, refer to your *Administrator's Guide*.

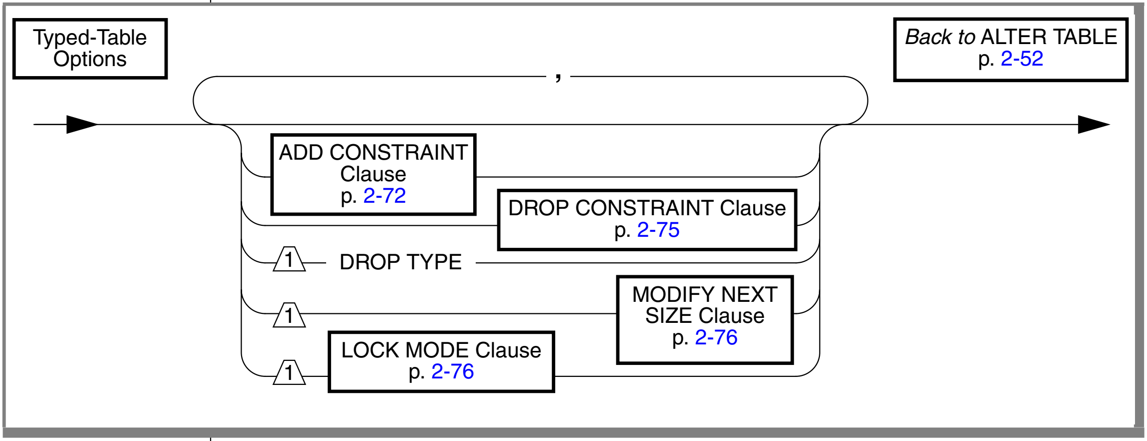
The Logging TYPE options have the following restrictions:

- You must perform a level-0 archive before the logging type of a table can be altered to STANDARD from any other logging type.
- If you want to change the logging type of a table to RAW, you must drop all indexes on the table before you do so.
- If you have triggers defined on the table, you cannot change the logging type to RAW or STATIC. Such tables do not support triggers.
- The table cannot be a SCRATCH or TEMP table, and you cannot change any of these types of tables to a SCRATCH or TEMP table.
- XPS The table cannot have a dependent GK index. ♦

IDS

Typed-Table Options

The Typed-Table options support operations on tables of a ROW data type.


Typed-Table Options
Back to ALTER TABLE p. 2-52
,
ADD CONSTRAINT Clause p. 2-72
DROP CONSTRAINT Clause p. 2-75
1
DROP TYPE
1
MODIFY NEXT SIZE Clause p. 2-76
1
LOCK MODE Clause p. 2-76

In Dynamic Server, the database server performs the actions in the ALTER TABLE statement in the order that you specify. If any action fails, the entire operation is cancelled.

Altering Subtables and Supertables

The following considerations apply to tables that are part of inheritance hierarchies:

- For subtables, ADD CONSTRAINT and DROP CONSTRAINT are not allowed on inherited constraints.
- For supertables, ADD CONSTRAINT and DROP CONSTRAINT propagate to all subtables.

DROP TYPE Option

Use the DROP TYPE option to drop the type from a table. DROP TYPE removes the association between a table and a named-row type. You must drop the type from a typed table before you can modify, drop, or change the data type of a column in the table.

If a table is part of a table hierarchy, you cannot drop its type unless it is the last subtype in the hierarchy. That is, you can only drop a type from a table if that table has no subtables. When you drop the type of a subtable, it is automatically removed from the hierarchy. The table rows are deleted from all indexes defined by its supertables.

Related Information

Related statements: CREATE TABLE, DROP TABLE, LOCK TABLE, and SET Database Object Mode

For discussions of data-integrity constraints and the ON DELETE CASCADE option, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion of database and table creation, see the *IBM Informix Database Design and Implementation Guide*.

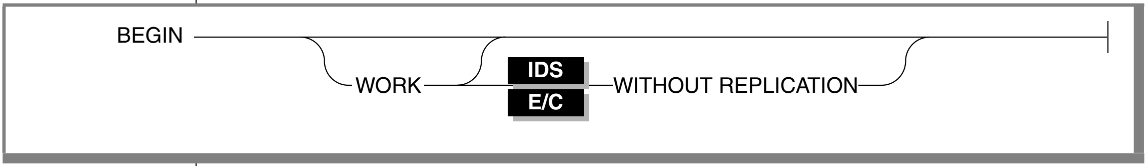
For information on how to maximize performance when you make table modifications, see your *Performance Guide*.

+

BEGIN WORK

Use the BEGIN WORK statement to start a *transaction* (a series of database operations that the COMMIT WORK or ROLLBACK WORK statement terminates). Use the BEGIN WORK WITHOUT REPLICATION statement to start a transaction that does not replicate to other database servers.

Syntax



Usage

Each row that an UPDATE, DELETE, or INSERT statement affects during a transaction is locked and remains locked throughout the transaction. A transaction that contains many such statements or that contains statements that affect many rows can exceed the limits that your operating system or the database server configuration imposes on the maximum number of simultaneous locks.

If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after you begin the transaction. Like other locks, this table lock is released when the transaction terminates. The example of a transaction on "Example of BEGIN WORK" on page 2-84 includes a LOCK TABLE statement.

Important: *Issue the BEGIN WORK statement only if a transaction is not in progress. If you issue a BEGIN WORK statement while you are in a transaction, the database server returns an error.*

E/C

In ESQL/C, if you use the BEGIN WORK statement within a UDR called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. These statements prevent the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

The WORK keyword is optional in a BEGIN WORK statement. The following two statements are equivalent:

```
BEGIN;

BEGIN WORK;
```

ANSI

BEGIN WORK and ANSI-Compliant Databases

In an ANSI-compliant database, you do not need the BEGIN WORK statement because transactions are implicit; every SQL statement occurs within a transaction. The database server generates a warning when you use a BEGIN WORK statement immediately after any of the following statements:

- DATABASE
- COMMIT WORK
- CREATE DATABASE
- ROLLBACK WORK

The database server returns an error when you use a BEGIN WORK statement after any other statement in an ANSI-compliant database.

IDS

E/C

BEGIN WORK WITHOUT REPLICATION

When you use Enterprise Replication for data replication, you can use the BEGIN WORK WITHOUT REPLICATION statement to start a transaction that does not replicate to other database servers.

You cannot execute BEGIN WORK WITHOUT REPLICATION as a stand-alone embedded statement in an ESQL/C application. Instead you must execute this statement indirectly. You can use either of the following methods:

- You can use a combination of the PREPARE and EXECUTE statements to prepare and execute the BEGIN WORK WITHOUT REPLICATION statement.
- You can use the EXECUTE IMMEDIATE statement to prepare and execute BEGIN WORK WITHOUT REPLICATION in a single step.

You cannot use the DECLARE cursor CURSOR WITH HOLD with the BEGIN WORK WITHOUT REPLICATION statement.

For more information about data replication, see the *IBM Informix Dynamic Server Enterprise Replication Guide*.

Example of BEGIN WORK

The following code fragment shows how you might place statements within a transaction. The transaction is made up of the statements that occur between the BEGIN WORK and COMMIT WORK statements. The transaction locks the **stock** table (LOCK TABLE), updates rows in the **stock** table (UPDATE), deletes rows from the **stock** table (DELETE), and inserts a row into the **manufact** table (INSERT).

```
BEGIN WORK;
   LOCK TABLE stock;
   UPDATE stock SET unit_price = unit_price * 1.10
      WHERE manu_code = 'KAR';
   DELETE FROM stock WHERE description = 'baseball bat';
   INSERT INTO manufact (manu_code, manu_name, lead_time)
      VALUES ('LYM', 'LYMAN', 14);
COMMIT WORK;
```

The database server must perform this sequence of operations either completely or not at all. When you include all of these operations within a single transaction, the database server guarantees that all the statements are completely and perfectly committed to disk, or else the database is restored to the same state that it was in before the transaction began.

Related Information

Related statements: COMMIT WORK and ROLLBACK WORK

For discussions of transactions and locking, see the *IBM Informix Guide to SQL: Tutorial*.

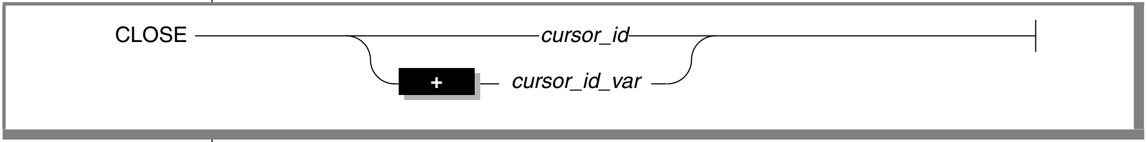
E/C

CLOSE

Use the CLOSE statement when you no longer need to refer to the rows that a select or function cursor retrieved, or to flush and close an insert cursor.

Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of cursor to be closed	Must have been declared.	Identifier, p. 4-189
cursor_id_var	Host variable that contains the value of *cursor_id*	Host variable must be a character data type.	Must conform to language-specific rules for names.

Usage

Closing a cursor makes the cursor unusable for any statements except OPEN or FREE and releases resources that the database server had allocated to the cursor. A CLOSE statement treats a cursor that is associated with an INSERT statement differently than one that is associated with a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.

In a database that is not ANSI-compliant, you can close a cursor that has not been opened or that has already been closed. No action is taken in these cases.

ANSI

In an ANSI-compliant database, the database server returns an error if you close a cursor that was not open. ♦

Closing a Select or Function Cursor

When a cursor identifier is associated with a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, closing the cursor terminates the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.

The database server releases all resources that it might have allocated to the active set of rows, for example, a temporary table that it used to hold an ordered set. The database server also releases any locks that it might have held on rows that were selected through the cursor. If a transaction contains the CLOSE statement, the database server does not release the locks until you execute COMMIT WORK or ROLLBACK WORK.

After you close a select or function cursor, you cannot execute a FETCH statement that names that cursor until you have reopened it.

Closing an Insert Cursor

When a cursor identifier is associated with an INSERT statement, the CLOSE statement writes any remaining buffered rows into the database. The number of rows that were successfully inserted into the database is returned in the third element of the **sqlerrd** array, **sqlca.sqlerrd[2]**, in the **sqlca** structure. For information on how to use SQLERRD to count the total number of rows that were inserted, see "Error Checking" on page 2-546.

The SQLCODE field of the **sqlca** structure, **sqlca.sqlcode**, indicates the result of the CLOSE statement for an insert cursor. If all buffered rows are successfully inserted, SQLCODE is set to zero. If an error is encountered, the **sqlca.sqlcode** field in the SQLCODE is set to a negative error message number.

When SQLCODE is zero, the row buffer space is released, and the cursor is closed; that is, you cannot execute a PUT or FLUSH statement that names the cursor until you reopen it.

Tip: *When you encounter an* ***SQLCODE*** *error, a corresponding* ***SQLSTATE*** *error value also exists. For information about how to get the message text, check the GET DIAGNOSTICS statement.*

If the insert is not successful, the number of successfully inserted rows is stored in **sqlerrd**. Any buffered rows that follow the last successfully inserted row are discarded. Because the insert fails, the CLOSE statement fails also, and the cursor is not closed. For example, a CLOSE statement can fail if insufficient disk space prevents some of the rows from being inserted. In this case, a second CLOSE statement can be successful because no buffered rows exist. An OPEN statement can also be successful because the OPEN statement performs an implicit close.

IDS

Closing a Collection Cursor

You can declare both select and insert cursors on collection variables. Such cursors are called collection cursors. Use the CLOSE statement to deallocate resources that have been allocated for the collection cursor.

For more information on how to use a collection cursor, see "Fetching from a Collection Cursor" on page 2-432 and "Inserting into a Collection Cursor" on page 2-544.

Using End of Transaction to Close a Cursor

The COMMIT WORK and ROLLBACK WORK statements close all cursors except those that are declared with a hold. It is better to close all cursors explicitly, however. For select or function cursors, this action simply makes the intent of the program clear. It also helps to avoid a logic error if the WITH HOLD clause is later added to the declaration of a cursor.

For an insert cursor, it is important to use the CLOSE statement explicitly so that you can test the error code. Following the COMMIT WORK statement, SQLCODE reflects the result of the COMMIT statement, not the result of closing cursors. If you use a COMMIT WORK statement without first using a CLOSE statement, and if an error occurs while the last buffered rows are being written to the database, the transaction is still committed.

For how to use insert cursors and the WITH HOLD clause, see "DECLARE" on page 2-323.

ANSI

In an ANSI-compliant database, a cursor cannot be closed implicitly. You must issue a CLOSE statement. ♦

Related Information

Related statements: DECLARE, FETCH, FLUSH, FREE, OPEN, PUT, and SET AUTOFREE

For an introductory discussion of cursors, see the *IBM Informix Guide to SQL: Tutorial*.

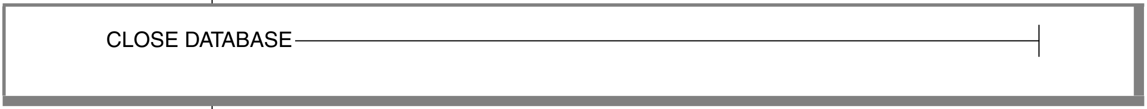
For a more advanced discussion of cursors, see the *IBM Informix ESQL/C Programmer's Manual*.

+

CLOSE DATABASE

Use the CLOSE DATABASE statement to close the current database.

Syntax



Usage

When you issue a CLOSE DATABASE statement, you can issue only the following SQL statements immediately after it:

- CONNECT
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- DISCONNECT

  (The DISCONNECT statement is valid here only if an explicit connection existed before CLOSE DATABASE was executed.)

Issue the CLOSE DATABASE statement before you drop the current database.

If your database supports transaction logging, and if you have started a transaction, you must issue a COMMIT WORK statement before you can use the CLOSE DATABASE statement.

The following example shows how to use the CLOSE DATABASE statement to drop the current database:

```
DATABASE stores_demo
. . .
CLOSE DATABASE
DROP DATABASE stores_demo
```

E/C

In ESQL/C, the CLOSE DATABASE statement cannot appear in a multi-statement PREPARE operation.

If you use the CLOSE DATABASE statement within a UDR called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This action prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning.

When you issue the CLOSE DATABASE statement, any declared cursors are no longer valid. You must re-declare any cursors that you want to use. ♦

ANSI

In an ANSI-compliant database, if no error is encountered while you exit from DB-Access in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction. ♦

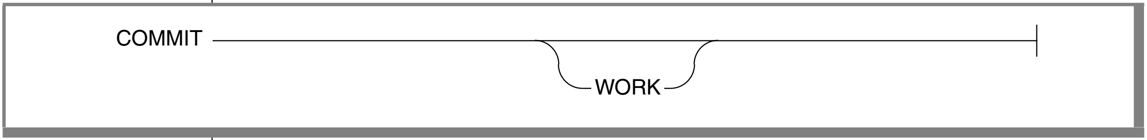
Related Information

Related statements: CONNECT, CREATE DATABASE, DATABASE, DISCONNECT, and DROP DATABASE

COMMIT WORK

Use the COMMIT WORK statement to commit all modifications made to the database from the beginning of a transaction.

Syntax



Usage

The COMMIT WORK statement informs the database server that you reached the end of a series of statements that must succeed as a single unit. The database server takes the required steps to make sure that all modifications that the transaction makes are completed correctly and saved to disk.

Use COMMIT WORK only at the end of a multistatement operation in a database with transaction logging, when you are sure that you want to keep all changes made to the database from the beginning of a transaction.

The COMMIT WORK statement releases all row and table locks.

The WORK keyword is optional in a COMMIT WORK statement. The following two statements are equivalent:

```
COMMIT;
COMMIT WORK;
```

The following example shows a transaction bounded by BEGIN WORK and COMMIT WORK statements.

```
BEGIN WORK;
   DELETE FROM call_type WHERE call_code = 'O';
   INSERT INTO call_type VALUES ('S', 'order status');
COMMIT WORK;
```

In this example, the user first deletes the row from the **call_type** table where the value of the **call_code** column is O. The user then inserts a new row in the **call_type** table where the value of the **call_code** column is S. The database server guarantees that both operations succeed or else neither succeeds.

E/C

In ESQL/C, the COMMIT WORK statement closes all open cursors except those that were declared using the WITH HOLD option. ♦

Issuing COMMIT WORK in a Database That Is Not ANSI Compliant

In a database that is not ANSI compliant, but that supports transaction logging, if you initiate a transaction with a BEGIN WORK statement, you must issue a COMMIT WORK statement at the end of the transaction. If you fail to issue a COMMIT WORK statement in this case, the database server rolls back any modifications that the transaction made to the database.

If you do not issue a BEGIN WORK statement, however, each statement executes within its own transaction. These single-statement transactions do not require either a BEGIN WORK statement or a COMMIT WORK statement.

ANSI

Issuing COMMIT WORK in an ANSI-Compliant Database

In an ANSI-compliant database, you do not need BEGIN WORK to mark the beginning of a transaction. You only need to mark the end of each transaction, because a transaction is always in effect. A new transaction starts automatically after each COMMIT WORK or ROLLBACK WORK statement.

You must, however, issue an explicit COMMIT WORK statement to mark the end of each transaction. If you fail to do so, the database server rolls back any modifications that the transaction made to the database.

DB-Access

In an ANSI-compliant database, however, if no error is encountered while you exit from DB-Access in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction. ♦

Related Information

Related statements: BEGIN WORK, ROLLBACK WORK, and DECLARE

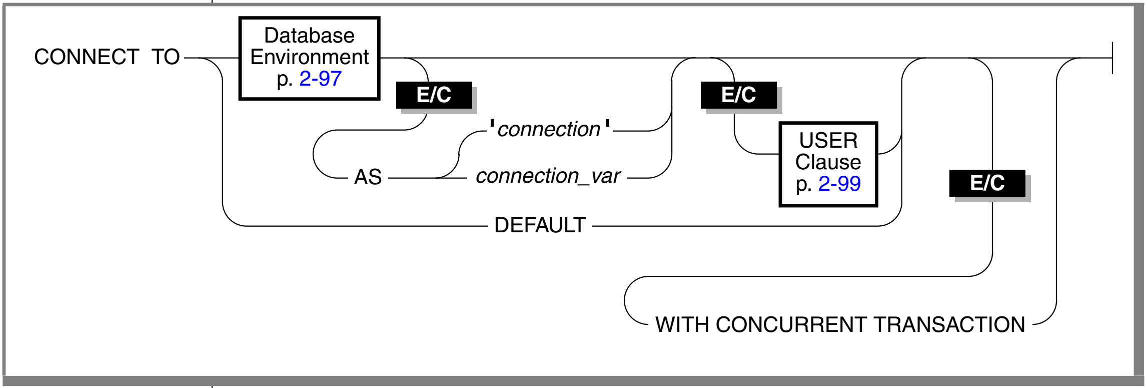
For a discussion of concepts related to transactions, see the *IBM Informix Guide to SQL: Tutorial*.

+

CONNECT

Use the CONNECT statement to connect to a database environment.

Syntax



Element	Purpose	Restrictions	Syntax
connection	Case-sensitive name that you declare here for a connection	Must be unique among connection names.	Quoted String, p. 4-243
connection_var	Host variable that stores the name of *connection*	Must be a fixed-length character data type.	Language specific

Usage

The CONNECT statement connects an application to a *database environment*, which can be a database, a database server, or a database and a database server. If the application successfully connects to the specified database environment, the connection becomes the *current connection* for the application. SQL statements fail if the application has no current connection to a database server. If you specify a database name, the database server opens that database. You cannot include CONNECT within a PREPARE statement.

An application can connect to several database environments at the same time, and it can establish multiple connections to the same database environment, provided each connection has a unique connection name.

UNIX

On UNIX, the only restriction on establishing multiple connections to the same database environment is that an application can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism, examine the **$INFORMIXDIR/etc/sqlhosts** file. For more information on the **sqlhosts** file, refer to your *Administrator's Guide*. ♦

Windows

On Windows, the local connection mechanism is named pipes. Multiple connections to the local server from one client can exist. ♦

Only one connection is current at any time; other connections are dormant. The application cannot interact with a database through a dormant connection. When an application establishes a new connection, that connection becomes current, and the previous current connection becomes dormant. You can make a dormant connection current with the SET CONNECTION statement. See also "SET CONNECTION" on page 2-646.

Privileges for Executing the CONNECT Statement

The current user, or PUBLIC, must have the Connect database privilege on the database specified in the CONNECT statement. The user who executes the CONNECT statement cannot have the same user name as an existing role in the database.

For information on how to use the USER clause to specify an alternate user name when the CONNECT statement connects to a database server on a remote host, see "USER Clause" on page 2-99.

Connection Identifiers

The optional connection name is a unique identifier that an application can use to refer to a connection in subsequent SET CONNECTION and DISCONNECT statements. If the application does not provide a connection name (or a connection-host variable), it can refer to the connection using the database environment. If the application makes more than one connection to the same database environment, however, each connection must have a unique connection name.

After you associate a connection name with a connection, you can refer to the connection using only that connection name.

Connection Context

Each connection encompasses a set of information that is called the *connection context*. The connection context includes the name of the current user, the information that the database environment associates with this name, and information on the state of the connection (such as whether an active transaction is associated with the connection). The connection context is saved when an application becomes dormant, and this context is restored when the application becomes current again. (For more information, see "Making a Dormant Connection the Current Connection" on page 2-646.)

DEFAULT Option

Use the DEFAULT option to request a connection to a default database server, called a *default connection*. The default database server can be either local or remote. To designate the default database server, set its name in the environment variable **INFORMIXSERVER**. This form of the CONNECT statement does not open a database.

If you select the DEFAULT option for the CONNECT statement, you must use the DATABASE statement or the CREATE DATABASE statement to open or create a database in the default database environment.

The Implicit Connection with DATABASE Statements

If you do not execute a CONNECT statement in your application, the first SQL statement must be one of the following *database statements* (or a single statement PREPARE for one of the following statements):

- DATABASE
- CREATE DATABASE
- DROP DATABASE

If one of these database statements is the first SQL statement in an application, the statement establishes a connection to a database server, which is known as an *implicit* connection. If the database statement specifies only a database name, the database server name is obtained from the **DBPATH** environment variable. This situation is described in "Specifying the Database Environment" on page 2-98.

An application that makes an implicit connection can establish other connections explicitly (using the CONNECT statement) but cannot establish another implicit connection unless the original implicit connection is disconnected. An application can terminate an implicit connection using the DISCONNECT statement.

After *any* implicit connection is made, that connection is considered to be the default connection, regardless of whether the database server is the default that the **INFORMIXSERVER** environment variable specifies. This feature allows the application to refer to the implicit connection if additional explicit connections are made, because the implicit connection has no identifier.

For example, if you establish an implicit connection followed by an explicit connection, you can make the implicit connection current by issuing the SET CONNECTION DEFAULT statement. This means, however, that once you establish an implicit connection, you cannot use the CONNECT DEFAULT statement, because the implicit connection is now the default connection.

The database statements can always be used to open a database or create a new database on the current database server.

WITH CONCURRENT TRANSACTION Option

The WITH CONCURRENT TRANSACTION clause lets you switch to a different connection while a transaction is active in the current connection. If the current connection was *not* established using the WITH CONCURRENT TRANSACTION clause, you cannot switch to a different connection if a transaction is active; the CONNECT or SET CONNECTION statement fails, returning an error, and the transaction in the current connection continues to be active.

In this case, the application must commit or roll back the active transaction in the current connection before it switches to a different connection.

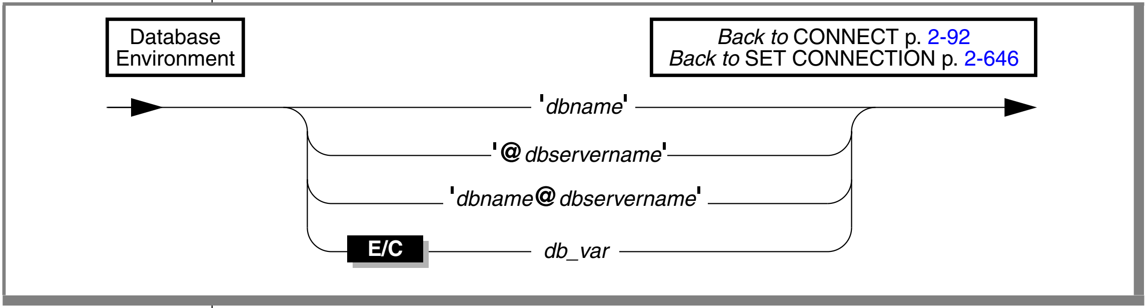
The WITH CONCURRENT TRANSACTION clause supports the concept of multiple concurrent transactions, where each connection can have its own transaction and the COMMIT WORK and ROLLBACK WORK statements affect only the current connection.The WITH CONCURRENT TRANSACTION clause does not support global transactions in which a single transaction spans databases over multiple connections. The COMMIT WORK and ROLLBACK WORK statements do not act on databases across multiple connections.

The following example illustrates how to use the WITH CONCURRRENT TRANSACTION clause:

```
main()
{
EXEC SQL connect to 'a@srv1' as 'A';
EXEC SQL connect to 'b@srv2' as 'B' with concurrent transaction;
EXEC SQL connect to 'c@srv3' as 'C' with concurrent transaction;

/*
   Execute SQL statements in connection 'C' , starting a
   transaction
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'

/*
   Execute SQL statements starting a transaction in 'B'.
   Now there are two active transactions, one each in 'B'
   and 'C'.
*/

EXEC SQL set connection 'A'; -- switch to connection 'A'

/*
   Execute SQL statements starting a transaction in 'A'.
   Now there are three active transactions, one each in 'A',
   'B' and 'C'.
*/

EXEC SQL set connection 'C'; -- ERROR, transaction active in 'A'


/*
   SET CONNECTION 'C' fails (current connection is still 'A')
   The transaction in 'A' must be committed/rolled back since
   connection 'A' was started without the CONCURRENT TRANSACTION
   clause.
*/

EXEC SQL commit work;-- commit tx in current connection ('A')

/*
   Now, there are two active transactions, in 'B' and in 'C',
   which must be committed/rolled back separately
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'
EXEC SQL commit work;        -- commit tx in current connection ('B')

EXEC SQL set connection 'C'; -- go back to connection 'C'
EXEC SQL commit work;      -- commit tx in current connection ('C')

EXEC SQL disconnect all;
}
```

Warning: *When an application uses the WITH CONCURRENT TRANSACTION clause to establish multiple connections to the same database environment, a deadlock condition can occur.*

Database Environment


Database Environment
Back to CONNECT p. 2-92
Back to SET CONNECTION p. 2-646
'*dbname*'
'@*dbservername*'
'*dbname*@*dbservername*'
E/C
db_var

Element	Purpose	Restrictions	Syntax
db_var	Host variable that contains a valid database environment (in one of the formats in the syntax diagram)	Must be a fixed-length character data type, whose contents are in a format from the syntax diagram.	Language specific
dbname	Database to which to connect	Must already exist.	Identifier, p. 4-189
dbservername	Name of the database server to which a connection is made	Must already exist; blank space is not valid between @ symbol and *dbservername*. See also "Restrictions on dbservername."	Identifier, p. 4-189

If the **DELIMIDENT** environment variable is set, any quotation (') marks in the database environment must be single. If **DELIMIDENT** is not set, then either single (') or double (") quotation marks are valid here.

Restrictions on dbservername

If you specify *dbservername*, it must satisfy the following restrictions.

- If the database server that you specify is not online, you receive an error.

UNIX

- On UNIX, the database server that you specify in *dbservername* must match the name of a database server in the **sqlhosts** file. ♦

Windows

- On Windows, *dbservername* must match the name of a database server in the **sqlhosts** subkey in the registry. It is recommended that you use the **setnet32** utility to update the registry. ♦

Specifying the Database Environment

You can specify a database server and a database, or a database server only, or a database only. How a database is located and opened depends on whether you specify a database server name in the database environment expression.

Only Database Server Specified

The *@dbservername* option establishes a connection to the database server only; it does not open a database. When you use this option, you must subsequently use the DATABASE or CREATE DATABASE statement (or a PREPARE statement for one of these statements and an EXECUTE statement) to open a database.

Database Server and Database Specified

If you specify both a database server and a database, your application connects to the database server, which locates and opens the database.

Only Database Specified

The *dbname* option establishes a connection to the default database server or to another database server in the **DBPATH** environment variable. It also locates and opens the named database. (The same is true of the *db_var* option if this specifies only a database name.)

If you specify only *dbname*, its database server is read from the **DBPATH** environment variable. The database server in the **INFORMIXSERVER** environment variable is always added before the **DBPATH** value.

UNIX

On UNIX, set the **INFORMIXSERVER** and **DBPATH** environment variables as the following example (for the C shell) shows:

```
setenv INFORMIXSERVER srvA
setenv DBPATH //srvB://srvC
```

♦

Windows

On Windows, choose **Start→Programs→Informix→setnet32** from the Task Bar and set the **INFORMIXSERVER** and **DBPATH** environment variables:

```
    set INFORMIXSERVER = srvA
    set DBPATH = //srvA://srvB://srvC
```

♦

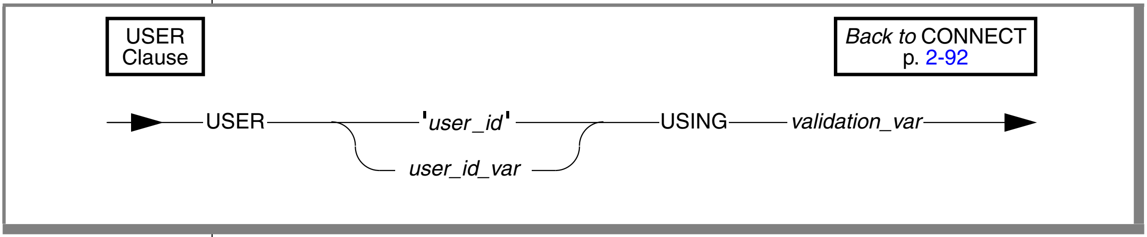
The next example shows the resulting **DBPATH** that your application uses:

```
    //srvA://srvB://srvC
```

The application first establishes a connection to the database server that **INFORMIXSERVER** specifies. The database server uses parameters in the configuration file to locate the database. If the database does not reside on the default database server, or if the default database server is not online, the application connects to the next database server in **DBPATH**. In the previous example, that database server would be **srvB**.

USER Clause

The USER clause specifies information that is used to determine whether the application can access the target computer on a remote host.



Element	Purpose	Restrictions	Syntax
user_id	Valid login name	See "Restrictions on the User Identifier Parameter" on page 2-100.	Quoted String, p. 4-243
user_id_var	Host variable that contains *user_id*	Must be a fixed-length character data type; same restrictions as *user_id*.	Language specific
validation_var	Host variable that contains a valid password for login name in *user_id* or *user_id_var*	Must be a fixed-length character data type. See "Restrictions on the Validation Variable Parameter" on page 2-100.	Language specific

The USER clause is required when the CONNECT statement connects to the database server on a remote host. Subsequent to the CONNECT statement, all database operations on the remote host use the specified user name.

Restrictions on the Validation Variable Parameter

UNIX

On UNIX, the password stored in *validation_var* must be a valid password and must exist in the **/etc/passwd** file. If the application connects to a remote database server, the password must exist in this file on both the local and remote database servers. ♦

Windows

On Windows, the password stored in *validation_var* must be a valid password and must be the one entered in **User Manager**. If the application connects to a remote database server, the password must exist in the domain of both the client and the server. ♦

Restrictions on the User Identifier Parameter

UNIX

On UNIX, the login name you specify in *user_id* must be a valid login name and must exist in the **/etc/passwd** file. If the application connects to a remote server, the login name must exist in this file on both the local and remote database servers. ♦

Windows

On Windows, the login name that you specify in *user_id* must be a valid login name and must exist in **User Manager**. If the application connects to a remote server, the login name must exist in the domain of both the client and the server. ♦

The connection is rejected if the following conditions occur:

- The specified user lacks the privileges to access the database named in the database environment.
- The specified user does not have the required permissions to connect to the remote host.
- You supply a USER clause but do not include the USING *validation_var* phrase.

E/C

X/O

In compliance with the X/Open specification for the CONNECT statement, the ESQL/C preprocessor allows a CONNECT statement that has a USER clause without the USING *validation_var* specification. If the *validation_var* is not present, however, the database server rejects the connection at runtime. ♦

Use of the Default User ID

If you do not supply the USER clause, the default user ID is used to attempt the connection. The default user ID is the login name of the user running the application. In this case, you obtain network permissions with the standard authorization procedures. For example, on UNIX, the default user ID must match a user ID in the **/etc/hosts.equiv** file. On Windows, you must be a member of the domain, or if the database server is installed locally, you must be a valid user on the computer where it is installed.

Related Information

Related Statements: DISCONNECT, SET CONNECTION, DATABASE, and CREATE DATABASE

For more information about **sqlhosts**, refer to your *Administrator's Guide*.

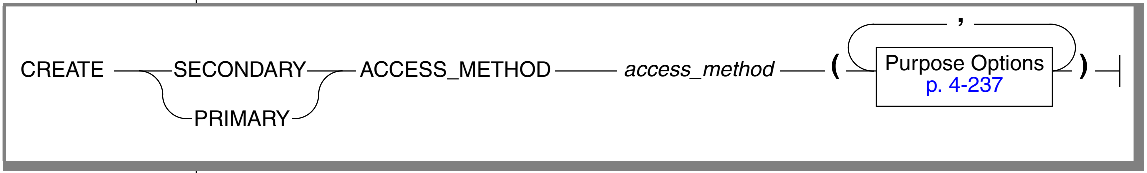
+

IDS

CREATE ACCESS_METHOD

Use the CREATE ACCESS_METHOD statement to register a new access method in the **sysams** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
access _method	Name declared here for the new access method	Must be unique among access-method names in the **sysams** system catalog table.	Database Object Name, p. 4-46

Usage

The CREATE ACCESS_METHOD statement adds a user-defined access method to a database.

When you create an access method, you specify purpose functions or methods, purpose flags, or purpose values as attributes of the access method, and you associate purpose keywords in the **sysams** system catalog table with user-defined functions or methods.

The **am_getnext** keyword is required in the Purpose Options list. You must use this to specify a UDR (or the name of a method) to scan for the next item that satisfies a query. For information on how to set purpose options, refer to "Purpose Options" on page 4-237.

The PRIMARY keyword specifies a user-defined primary-access method for a virtual table. The SECONDARY keyword specifies creating a user-defined secondary-access method for a virtual index. The SECONDARY keyword (and creating virtual indexes) is not supported in the Java Virtual-Table Interface.

You must have the DBA or Resource privilege to create an access method.

The following statement creates a secondary-access method named **T-tree** that resides in an sbspace:

```
CREATE SECONDARY ACCESS_METHOD T_tree
(
am_getnext = ttree_getnext,
am_unique,
am_cluster,
am_sptype = 'S'
);
```

In the preceding example, the **am_getnext** keyword is associated with the user-defined function **ttree_getnext( )**. The **T_tree** access method supports unique keys and clustering.

The following statement creates a primary-access method named **am_tabprops** that resides in an extspace.

```
CREATE PRIMARY ACCESS_METHOD am_tabprops
(
am_open = FS_open,
am_close = FS_close,
am_beginscan = FS_beginScan,
am_create = FS_create,
am_scancost = FS_scanCost,
am_endscan = FS_endScan,
am_getnext = FS_getNext,
am_getbyid = FS_getById,
am_drop = FS_drop,
am_rowids,
am_sptype = ’x’
);
```

Related Information

Related statements: ALTER ACCESS_METHOD and DROP ACCESS_METHOD

For detailed information about how to set purpose-option specifications, see “Purpose Options” on page 4-237.

For more information on primary-access methods, see the *IBM Informix Virtual-Table Interface Programmer’s Guide*.

For more information on secondary-access methods, see the *IBM Informix Virtual-Index Interface Programmer’s Guide* (C only).

For a discussion of privileges, see the GRANT or REVOKE statements or the *IBM Informix Database Design and Implementation Guide*.

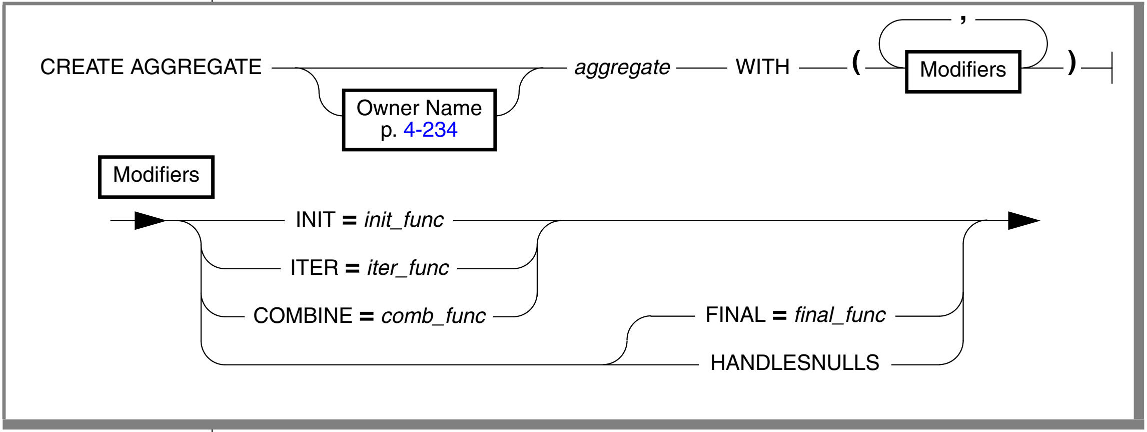
+

IDS

CREATE AGGREGATE

Use the CREATE AGGREGATE statement to create a new aggregate function and register it in the **sysaggregates** system catalog table. User-defined aggregates extend the functionality of the database server by performing aggregate computations that the user implements.

Syntax



Element	Purpose	Restrictions	Syntax
aggregate	Name of the new aggregate	Must be unique among names of built-in aggregates and UDRs	Identifier, p. 4-189
comb_func	Function that merges one partial result into the other and returns the updated partial result	Must specify the combined function both for parallel queries and for sequential queries	Database Object Name, p. 4-46
final_func	Function that converts a partial result into the result type	If this is omitted, then the returned value is the final result of *iter_func*	Database Object Name, p. 4-46
init_func	Function that initializes the data structures required for the aggregate computation	Must be able to handle NULL arguments	Database Object Name, p. 4-46
iter_func	Function that merges a single value with a partial result and returns updated partial result	Must specify an iterator function. If *init_func* is omitted, *iter_func* must be able to handle NULL arguments	Database Object Name, p. 4-46

Usage

You can specify the INIT, ITER, COMBINE, FINAL, and HANDLESNULLS modifiers in any order.

Important: *You must specify the ITER and COMBINE modifiers in a CREATE AGGREGATE statement. You do not have to specify the INIT, FINAL, and HANDLESNULLS modifiers in a CREATE AGGREGATE statement.*

The ITER, COMBINE, FINAL, and INIT modifiers specify the support functions for a user-defined aggregate. These support functions do not have to exist at the time you create the user-defined aggregate.

If you omit the HANDLESNULLS modifier, rows with NULL aggregate argument values do not contribute to the aggregate computation. If you include the HANDLESNULLS modifier, you must define all the support functions to handle NULL values as well.

Extending the Functionality of Aggregates

Dynamic Server provides two ways to extend the functionality of aggregates. Use the CREATE AGGREGATE statement only for the second of the two cases.

- Extensions of built-in aggregates

  A built-in aggregate is an aggregate that the database server provides, such as **COUNT**, **SUM**, or **AVG**. These support only built-in data types. To extend a built-in aggregate so that it supports a user-defined data type (UDT), you must create user-defined routines that overload the binary operators for that aggregate. For further information on extending built-in aggregates, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

- Creation of user-defined aggregates

  A user-defined aggregate is an aggregate that you define to perform an aggregate computation that the database server does not provide. You can use user-defined aggregates with built-in data types, extended data types, or both. To create a user-defined aggregate, use the CREATE AGGREGATE statement. In this statement, you name the new aggregate and specify the support functions that compute the aggregate result. These support functions perform initialization, sequential aggregation, combination of results, and type conversion.

Example of Creating a User-Defined Aggregate

The following example defines a user-defined aggregate named **average:**

```
CREATE AGGREGATE average
   WITH (
      INIT = average_init,
      ITER = average_iter,
      COMBINE = average_combine,
      FINAL = average_final
      )
```

Before you use the **average** aggregate in a query, you must also use CREATE FUNCTION statements to create the support functions specified in the CREATE AGGREGATE statement.

The following table gives an example of the task that each support function might perform for **average**.

Keyword	Support Function	Effect
INIT	**average_init**	Allocates and initializes an extended data type storing the current sum and the current row count
ITER	**average_iter**	For each row, adds the value of the expression to the current sum and increments the current row count by one
COMBINE	**average_combine**	Adds the current sum and the current row count of one partial result to the other and returns the updated result
FINAL	**average_final**	Returns the ratio of the current sum to the current row count and converts this ratio to the result type

Parallel Execution

The database server can break up an aggregate computation into several pieces and compute them in parallel. The database server uses the INIT and ITER support functions to compute each piece sequentially. Then the database server uses the COMBINE function to combine the partial results from all the pieces into a single result value. Whether an aggregate is parallel is an optimization decision that is transparent to the user.

Related Information

Related statements: CREATE FUNCTION and DROP AGGREGATE

For information about how to invoke a user-defined aggregate, see "User-Defined Aggregates" on page 4-173 in the Expression segment.

For a description of the **sysaggregates** system catalog table that stores data about user-defined aggregates, see the *IBM Informix Guide to SQL: Reference*.

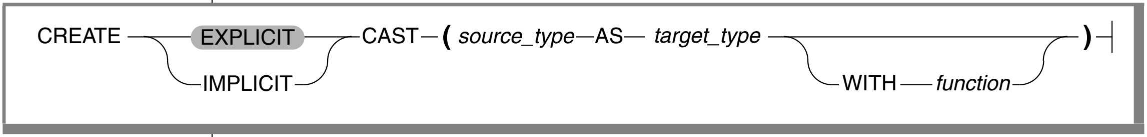
For a discussion of user-defined aggregates, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

+
IDS

CREATE CAST

Use the CREATE CAST statement to register a cast that converts data from one data type to another.

Syntax



Element	Purpose	Restrictions	Syntax
function	User-defined function that you register to implement the cast	See "WITH Clause" on page 2-111.	Database Object Name, p. 4-46
source_type	Data type to be converted	Must exist in the database at the time when the cast is registered. See also "Source and Target Data Types" on page 2-109.	Data Type, p. 4-49
target_type	Data type that results from the conversion	The same restrictions that apply for the *source_type* (as listed above) also apply for the *target_type*.	Data Type, p. 4-49

Usage

A cast is a mechanism that the database server uses to convert one data type to another. The database server uses casts to perform the following tasks:

- To compare two values in the WHERE clause of a SELECT, UPDATE, or DELETE statement
- To pass values as arguments to user-defined routines
- To return values from user-defined routines

To create a cast, you must have the necessary privileges on both the *source data type* and the *target data type*. All users have permission to use the built-in data types. To create a cast to or from an OPAQUE, DISTINCT, or named ROW data type, however, requires the Usage privilege on that data type.

The CREATE CAST statement registers a cast in the **syscasts** system catalog table. For more information on **syscasts**, see the chapter on system catalog tables in the *IBM Informix Guide to SQL: Reference*.

Source and Target Data Types

The CREATE CAST statement defines a cast that converts a *source data type* to a *target data type*. Both the *source data type* and *target data type* must exist in the database when you execute the CREATE CAST statement to register the cast. The *source data type* and the *target data type* have the following restrictions:

- Either the *source data type* or the *target data type*, but not both, can be a built-in type.
- Neither the *source data type* nor the *target data type* can be a distinct type of the other.
- Neither the *source data type* nor the *target data type* can be a collection data type.

Explicit and Implicit Casts

To process queries with multiple data types often requires casts that convert data from one data type to another. You can use the CREATE CAST statement to create the following kinds of casts:

- Use the CREATE EXPLICIT CAST statement to define an *explicit* cast.
- Use the CREATE IMPLICIT CAST statement to define an *implicit* cast.

Explicit Casts

An explicit cast is a cast that you must specifically invoke, with either the CAST AS keywords or with the cast operator (::). The database server does *not* automatically invoke an explicit cast to resolve data type conversions. The EXPLICIT keyword is optional; by default, the CREATE CAST statement creates an explicit cast.

The following CREATE CAST statement defines an explicit cast from the **rate_of_return** opaque data type to the **percent** distinct data type:

```
CREATE EXPLICIT CAST (rate_of_return AS percent
   WITH rate_to_prcnt)
```

The following SELECT statement explicitly invokes this explicit cast in its WHERE clause to compare the **bond_rate** column (of type **rate_of_return**) to the **initial**_APR column (of type **percent**):

```
SELECT bond_rate FROM bond
WHERE bond_rate::percent > initial_APR
```

Implicit Casts

The database server invokes built-in casts to convert from one built-in data type to another built-in type that is not directly substitutable. For example, the database server performs conversion of a character type such as CHAR to a numeric type such as INTEGER through a built-in cast.

An implicit cast is a cast that the database server can invoke automatically when it encounters data types that cannot be compared with built-in casts. This type of cast enables the database server to automatically handle conversions between other data types.

To define an implicit cast, specify the IMPLICIT keyword in the CREATE CAST statement. For example, the following CREATE CAST statement specifies that the database server should automatically use the **prcnt_to_char( )** function to convert from the CHAR data type to a distinct data type, **percent**:

```
CREATE IMPLICIT CAST (CHAR AS percent WITH char_to_prcnt)
```

This cast only supports automatic conversion *from* the CHAR data type *to* **percent**. For the database server to convert *from* **percent** *to* CHAR, you also need to define another implicit cast, as follows:

```
CREATE IMPLICIT CAST (percent AS CHAR WITH prcnt_to_char)
```

The database server automatically invokes the **char_to_prcnt( )** function to evaluate the WHERE clause of the following SELECT statement:

```
SELECT commission FROM sales_rep WHERE commission > "25%"
```

Users can also invoke implicit casts explicitly. For more information on how to explicitly invoke a cast function, see “Explicit Casts” on page 2-109.

When a built-in cast does not exist for conversion between data types, you can create user-defined casts to make the necessary conversion.

WITH Clause

The WITH clause of the CREATE CAST statement specifies the name of the user-defined function to invoke to perform the cast. This function is called the cast function. You must specify a *function name* unless the *source data type* and the *target data typ*e have identical representations. Two data types have identical representations when the following conditions are met:

- Both data types have the same length and alignment
- Both data types are passed by reference or both are passed by value

The cast function must be registered in the same database as the cast at the time the cast is invoked, but need not exist when the cast is created. The CREATE CAST statement does not check permissions on the specified *function name*, or even verify that the cast function exists. Each time a user invokes the cast explicitly or implicitly, the database server verifies that the user has the Execute privilege on the cast function.

Related Information

Related statements: CREATE FUNCTION, CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, CREATE ROW TYPE, and DROP CAST

For more information about data types, casting, and conversion, see the Data Types segment in this manual and the *IBM Informix Guide to SQL: Reference*.

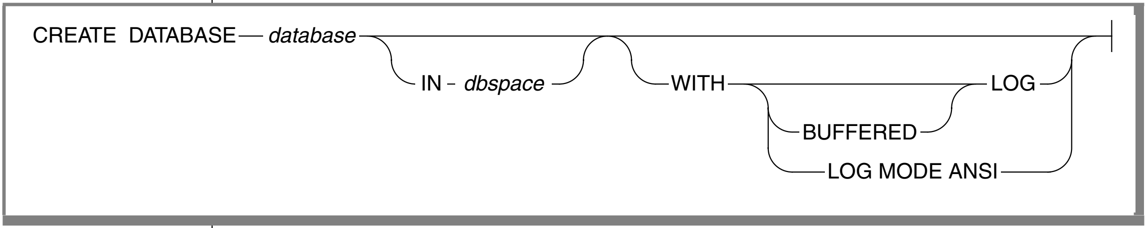
For examples that show how to create and use casts, see the *IBM Informix Database Design and Implementation Guide*.

+

CREATE DATABASE

Use the CREATE DATABASE statement to create a new database.

Syntax



Element	Purpose	Restrictions	Syntax
database	Name that you declare here for the new database that you are creating	Must be unique among the names of databases on the database server.	Database Name, p. 4-44
dbspace	The dbspace to store the data for this database; default is the root dbspace	Must already exist.	Identifier, p. 4-189

Usage

This statement is an extension to ANSI-standard syntax. (The ANSI/ISO standard for the SQL language does not specify any syntax for construction of a database, the process by which a database comes into existence.)

The *database* that CREATE DATABASE specifies becomes the current database.

The database name that you use must be unique within the database server environment in which you are working. The database server creates the system catalog tables that describe the structure of the database.

When you create a database, you alone can access it. It remains inaccessible to other users until you, as DBA, grant database privileges. For information on how to grant database privileges, see "GRANT" on page 2-459.

E/C

In ESQL/C, the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation. ♦

If you do not specify the dbspace, the database server creates the system catalog tables in the **root** dbspace. The following statement creates the **vehicles** database in the **root** dbspace:

```
CREATE DATABASE vehicles
```

The following statement creates the **vehicles** database in the **research** dbspace:

```
CREATE DATABASE vehicles IN research
```

XPS

In Extended Parallel Server you can create a database in the dbspace of the primary coserver (coserver 1) only. ♦

Logging Options

The logging options of the CREATE DATABASE statement determine the type of logging that is done for the database.

In the event of a failure, the database server uses the log to re-create all committed transactions in your database.

If you do not specify the WITH LOG option, you cannot use transactions or the statements that are associated with databases that have logging (BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET IMPLICIT TRANSACTION, SET LOG, and SET ISOLATION).

XPS

If you are using Extended Parallel Server, the CREATE DATABASE statement always creates a database with logging. If you do not specify the WITH LOG option, the unbuffered log type is used by default. ♦

Designating Buffered Logging

The following example creates a database that uses a buffered log:

```
CREATE DATABASE vehicles WITH BUFFERED LOG
```

If you use a buffered log, you marginally enhance the performance of logging at the risk of not being able to re-create the last few transactions after a failure. (See the discussion of buffered logging in the *IBM Informix Database Design and Implementation Guide*.)

ANSI

ANSI-Compliant Databases

When you use the LOG MODE ANSI option in the CREATE DATABASE statement, the database that you create is an ANSI-compliant database. The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG MODE ANSI
```

ANSI-compliant databases are different from databases that are not ANSI compliant in several ways, including the following features:

- All statements are automatically contained in transactions.
- All databases use unbuffered logging.
- Owner-naming is enforced.

  You must use the owner name when you refer to each table, view, synonym, index, or constraint, unless you are the owner.
- For databases, the default isolation level is REPEATABLE READ.
- Default privileges on objects differ from those in databases that are not ANSI compliant. Users do not receive PUBLIC privilege to tables and synonyms by default.

Other slight differences exist between databases that are ANSI compliant and those that are not. These differences are noted with the related SQL statement in this manual. For a detailed discussion of the differences between ANSI compliant databases and databases that are not ANSI-compliant, see the *IBM Informix Database Design and Implementation Guide*.

Creating an ANSI-compliant database does not mean that you automatically get warnings for Informix extensions to the ANSI/ISO standard for SQL syntax when you run the database. You must also use the **-ansi** flag or the **DBANSIWARN** environment variable to receive such warnings.

For additional information about **-ansi** and **DBANSIWARN**, see the *IBM Informix Guide to SQL: Reference*.

Related Information

Related statements: CLOSE DATABASE, CONNECT, DATABASE, and DROP DATABASE

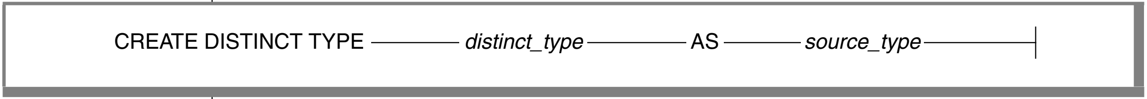
+

IDS

CREATE DISTINCT TYPE

Use the CREATE DISTINCT TYPE statement to create a new distinct data type.

Syntax



Element	Purpose	Restrictions	Syntax
distinct_type	Name that you declare here for the new distinct data type	In an ANSI-compliant database, the combination of the owner and data type must be unique within the database. In a database that is not ANSI compliant, the name must be unique among names of data types in the database.	Data Type, p. 4-49
source_type	Name of an existing type on which the new type is based	Must be either a built-in data type or one created with the CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, or CREATE ROW TYPE statement.	Data Type, p. 4-49

Usage

A *distinct type* is a data type based on a built-in data type or on an existing opaque data type, a named-row data type, or another distinct data type. Distinct data types are strongly typed. Although the distinct type has the same physical representation as data of its source type, values of the two types cannot be compared without an explicit cast from one type to the other

To create a distinct type in a database, you must have the Resource privilege. Any user with the Resource privilege can create a distinct type from one of the built-in data types, which user **informix** owns.

Important: *You cannot create a distinct type on the SERIAL or SERIAL8 data type.*

To create a distinct type from an opaque type, a named-ROW type, or another distinct type, you must be the owner of the data type or have the Usage privilege on the data type.

Once a distinct type is defined, only the type owner and the DBA can use it. The owner of the type can grant other users the Usage privilege on the type.

A distinct type has the same storage structure as its source type. The following statement creates the distinct type **birthday**, based on the built-in DATE data type:

```
CREATE DISTINCT TYPE birthday AS DATE
```

Although Dynamic Server uses the same storage format for the distinct type as it does for its source type, a distinct type and its source type cannot be compared in an operation unless one type is explicitly cast to the other type.

Privileges on Distinct Types

To create a distinct type, you must have the Resource privilege on the database. When you create the distinct type, only you, the owner, have Usage privilege on this type. Use the GRANT or REVOKE statements to grant or revoke Usage privilege to other database users.

To find out what privileges exist on a particular type, check the **sysxtdtypes** system catalog table for the owner name and the **sysxtdtypeauth** system catalog table for additional data type privileges that might have been granted. For more information on system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

DB

The DB-Access utility can also display privileges on distinct types. ♦

Support Functions and Casts

When you create a distinct type, Dynamic Server automatically defines two explicit casts:

- A cast from the distinct type to its source type
- A cast from the source type to the distinct type

Because the two data types have the same representation (the same length and alignment), no support functions are required to implement the casts.

You can create an implicit cast between a distinct type and its source type. To create an implicit cast, use the Table Options clause to specify the format of the external data. You must first drop the default explicit cast, however, between the distinct type and its source type.

All support functions and casts that are defined on the source type can be used on the distinct type. Casts and support functions that are defined on the distinct type, however, Use the Table Options clause to specify the format of the external data.are not available to the source type.

Manipulating Distinct Types

When you compare or manipulate data of a distinct type and its source type, you must explicitly cast one type to the other in the following situations:

- To insert or update a column of one type with values of the other type
- To use a relational operator to add, subtract, multiply, divide, compare, or otherwise manipulate two values, one of the source type and one of the distinct type

For example, suppose you create a distinct type, dist_type, that is based on the NUMERIC data type. You then create a table with two columns, one of type dist_type and one of type NUMERIC.

```
CREATE DISTINCT TYPE dist_type AS NUMERIC;
CREATE TABLE t(col1 dist_type, col2 NUMERIC);
```

To directly compare the distinct type and its source type or assign a value of the source type to a column of the distinct type, you must cast one type to the other, as the following examples show:

```
INSERT INTO tab (col1) VALUES (3.5::dist_type);

SELECT col1, col2
   FROM t WHERE (col1::NUMERIC) > col2;

SELECT col1, col2, (col1 + col2::dist_type) sum_col
   FROM tab;
```

Related Information

Related statements: CREATE CAST, CREATE FUNCTION, CREATE OPAQUE TYPE, CREATE ROW TYPE, DROP TYPE, and DROP ROW TYPE

For information and examples that show how to use and cast distinct types, see the *IBM Informix Guide to SQL: Tutorial*.

For more information on when you might create a distinct type, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

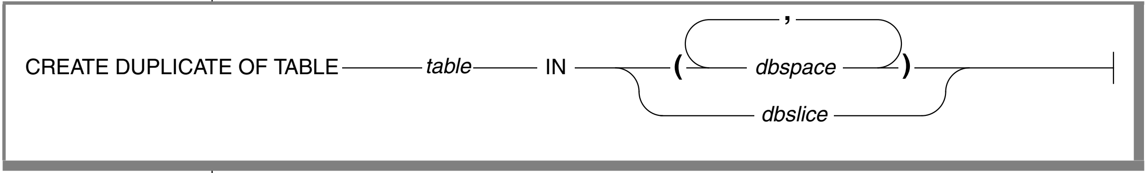
+

XPS

CREATE DUPLICATE

Use the CREATE DUPLICATE statement to create a duplicate copy of an existing table for read-only use in a specified dbslice or in specified dbspaces across coservers.

Syntax



Element	Description	Restrictions	Syntax
dbslice	Name of a dbslice in which to duplicate one fragment of *table*	Must exist and must contain at most one dbspace on each target coserver.	Database Object Name
dbspace	Name of a dbspace in which to duplicate one fragment of *table*	Must exist and must not already contain an original or duplicate fragment of *table.*	Database Object Name
table	Name of the original table from which to create a duplicate	Must already exist in the database. See also "Supported Operations" on page 2-120.	Database Object Name

Usage

If the original *table* resides entirely on a single coserver, you can create duplicate copies of small tables across coservers for read-only use. For each attached index of the original table, a similarly defined index is built on each table duplicate, using the same dbspaces as the table.

Because query operators read the local copy of the table, duplicating small tables across coservers might improve the performance of some queries.

If a local copy of a duplicated table exists but is not available because the dbspace that stores it is offline (or for some similar reason), a query that requires access to the table fails. The database server does not attempt to access the original table.

The location of a duplicated table can be either a dbslice or a comma-separated list of dbspaces. You can combine dbslices and lists of dbspaces in a single CREATE DUPLICATE statement.

- If the original table is not fragmented, the dbspace list need provide only a single dbspace on each coserver.

  For example, if the table **tab1** is not fragmented, enter the following statement to create a duplicate on the remaining three of the four coservers if the original table is stored in the dbspace **db1** on coserver 1 and **db2** is on coserver 2, **db3** is on coserver 3, and **db4** is on coserver 4.

  ```
  CREATE DUPLICATE OF TABLE tab1 IN (db2, db3, db4)
  ```

- If the original table is fragmented with one fragment in the first dbspace of several dbslices that contain dbspaces on all coservers, you can create duplicate copies of the table in the remaining dbspaces of the dbslice.

  For example, you might create the **tab3** table in the first dbspace of three dbslices, each of which contains a dbspace on each coserver, as follows:

  ```
  CREATE TABLE tab3 (...)
     FRAGMENT BY HASH (....) IN dbsl1.1, dbsl2.1, dbsl3.1
  ```

  To duplicate the **tab3** table across the remaining coservers, use the following statement:

  ```
  CREATE DUPLICATE OF TABLE tab3 IN dbsl1, dbsl2, dbsl3
  ```

- You can mix dbslice names and dbspace lists in the same CREATE DUPLICATE statement. For example, instead of using dbspaces in a dbslice, for the previous example you might enter the following statement in which **dbsp2a** is on coserver 2, **dbsp3a** is on coserver 3, and **dbsp4a** is on coserver 4:

  ```
  CREATE DUPLICATE OF  TABLE tab3 IN
     dbsl1, dbsl2, (dbsp2a, dbsp3a, dbsp4a)
  ```

The first fragment of the original table is duplicated into **dbsl1**, which contains a dbspace on each coserver, the second fragment into **dbsl2**, which also contains a dbspace on each coserver, and the third fragment into the list of dbspaces.

Only one fragment of a duplicated table can reside in any single dbspace. You cannot list an existing dbspace of the duplicated table in the list of dbspaces into which it is duplicated, but it is not an error for an existing dbspace to be a member of a dbslice that specifies duplication dbspaces. Matching dbspaces in the dbslice are ignored.

The CREATE DUPLICATE statement requires the ALTER privilege.

Supported Operations

The following operations are permitted on duplicated tables:

- SELECT
- UPDATE STATISTICS
- LOCK and UNLOCK
- SET RESIDENCY
- DROP TABLE

You cannot duplicate a table in certain circumstances. The table must not:

- Have GK or detached indexes
- Use range fragmentation
- Be a temporary table
- Be a violations or diagnostic table
- Contain BYTE, TEXT, or SERIAL columns
- Have referential constraints

The CREATE DUPLICATE statement does not support incremental duplication. It also does not support multiple duplicates of the same table on a single coserver, nor duplication of tables that are fragmented across coservers.

If you need to take a dbspace offline and it contains a copy of a duplicated table, or if you need to update data in a duplicated table, first drop all duplicates of the table, as described in "DROP DUPLICATE."

Related Statement

DROP DUPLICATE

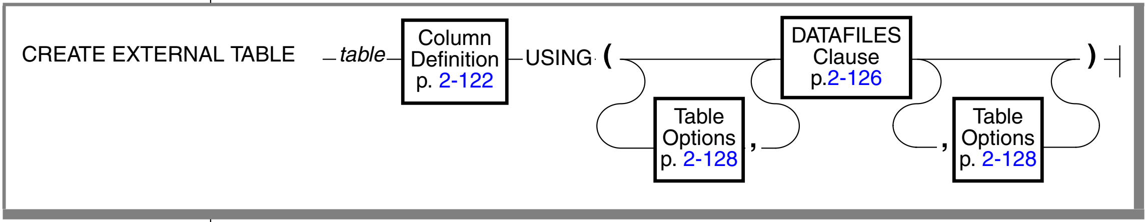
+

XPS

CREATE EXTERNAL TABLE

Use the CREATE EXTERNAL TABLE statement to define an external source that is not part of your database to load and unload data for your database.

Syntax



Element	Purpose	Restrictions	Syntax
table	Name declared here for a table to store external data	Must be unique among names of tables, views, and synonyms in the current database.	Database Object Name, p. 4-46

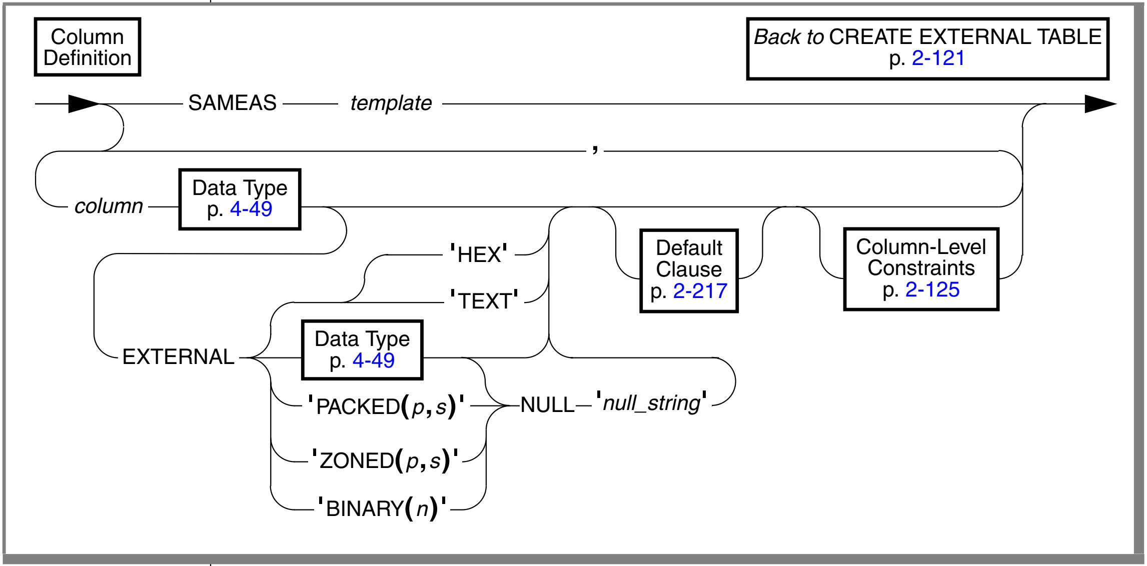
Usage

The left-hand portion of the syntax diagram declares the name of the table and defines its columns and any column-level constraints.

The portion that follows the USING keyword specifies external files that the database server opens when you use the external table, and additional options for characteristics of the external table.

After executing the CREATE EXTERNAL TABLE statement, you can move data to and from the external source with an INSERT INTO ... SELECT statement. See the section "INTO EXTERNAL Clause" on page 2-635 for more information about loading the results of a query into an external table.

Column Definition



Element	Purpose	Restrictions	Syntax
column	One column name for each column of the external table	For each column, you must specify a built-in data type	Identifier, p. 4-189
n	Number of 8-bit bytes to represent the integer	For FIXED format binary integers; *big-endian* byte order	*n*=2 for 16-bit integers; *n*=4 for 32-bit integers
p	Precision (total number of digits)	For FIXED-format files only	Literal Number, p. 4-216
s	Scale (digits in fractional part)	For FIXED-format files only	Literal Number, p. 4-216
null_string	Value to be interpreted as NULL	See "Defining NULL Values" on page 2-123.	Quoted String, p. 4-243
template	Existing table with the same schema as the external table	Cannot be subset of columns nor differ in any column data type	Database Object Name, p. 4-46

Using the SAMEAS Clause

The SAMEAS *template* clause uses all the column names and data types from the *template* table in the definition of new table. You cannot, however, use indexes in the external table definition, and you cannot use the SAMEAS clause for FIXED-format files.

Using the EXTERNAL Keyword

Use the EXTERNAL keyword to specify a data type for each column of your external table that has a data type different from the internal table. For example, you might have a VARCHAR column in the internal table that you want to map to a CHAR column in the external table.

You must specify an external type for every column that is in fixed format. You cannot specify an external type for delimited format columns except for BYTE and TEXT columns where your specification is optional. For more information, see "TEXT and HEX External Types" on page 2-124.

Integer Data Types

Besides valid Informix integer data types, you can specify packed decimal, zoned decimal, and IBM-format binary representation of integers. For packed or zoned decimal, specify *precision* (total number of digits in the number) and *scale* (number of digits that are to the right of the decimal point). Packed decimal representation can store two digits, or a digit and a sign, in each byte. Zoned decimal requires (p + 1) bytes to store p digits and the sign.

Big-Endian Format

The database server also supports two IBM-format binary representations of integers: BINARY(2) for 16-bit integer storage and BINARY(4) for 32-bit integer storage. The most significant byte of each number has the lowest address; that is, binary-format integers are stored big-end first (big-endian format) in the manner of IBM and Motorola processors. Intel processors and some others store binary-format integers little-end first, a storage method that the database server does not support for external data.

Defining NULL Values

The packed decimal, zoned decimal, and binary data types do not have a natural NULL value, so you must define a value to be interpreted as a NULL when loading or unloading data from an external source. You can define the *null_string* as a number outside the set of numbers stored in the data file (for example, -9999.99). You can also define a bit pattern in the field as a hexadecimal pattern, such as 0xffff, that is to be interpreted as a NULL.

The database server uses the NULL representation for a FIXED-format external table to both interpret values as the data is loaded into the database and to format NULL values into the appropriate data type when data is unloaded to an external table.

The following examples are of column definitions with NULL values for a FIXED-format external table:

```
i smallint external “binary (2)” null “-32767”
li integer external “binary (4)” null “-99999”
d decimal (5,2) external “packed (5,2)” null “0xffffff”
z decimal (4,2) external “zoned (4,2)” null “0x0f0f0f0f”
zl decimal (3,2) external “zoned (3,2)” null “-1.00”
```

If the packed decimal or zoned decimal is stored with all bits cleared to represent a NULL value, the *null_string* can be defined as `0x0`. The following rules apply to the value assigned to a *null_string*:

- The NULL representation must fit into the length of the external field.
- If a bit pattern is defined, the *null_string* is not case sensitive.
- If a bit pattern is defined, the *null_string* must begin with `0x`.
- For numeric fields, the left-most fields are assigned zeros by the database server if the bit pattern does not fill the entire field.
- If the NULL representation is not a bit pattern, the NULL value must be a valid number for that field.

Warning: *If a row that contains a NULL value is unloaded into an external table and the column that receives the NULL value has no NULL value defined, the database server inserts a zero into the column.*

TEXT and HEX External Types

An Informix BYTE or TEXT column can be encoded in either the TEXT or HEX external type. You can use only delimited BYTE and TEXT formats with these external types. Fixed formats are not allowed. In addition, you cannot use these external types with any other type of delimited-format columns (such as character columns).

You do not need to specify these external types. If you do not define an external column specifically, Informix TEXT columns default to TEXT and Informix BYTE columns default to HEX.

The database server interprets two adjacent field delimiters as a NULL value.

User-defined delimiters are limited to one byte each. During unloading, delimiters and backslash (\ ) symbols are escaped. During loading, any character that follows a backslash is interpreted as a literal. In TEXT format, nonprintable characters are directly embedded in the data file. For delimiter rules in a multibyte locale, see the *IBM Informix GLS User's Guide*.

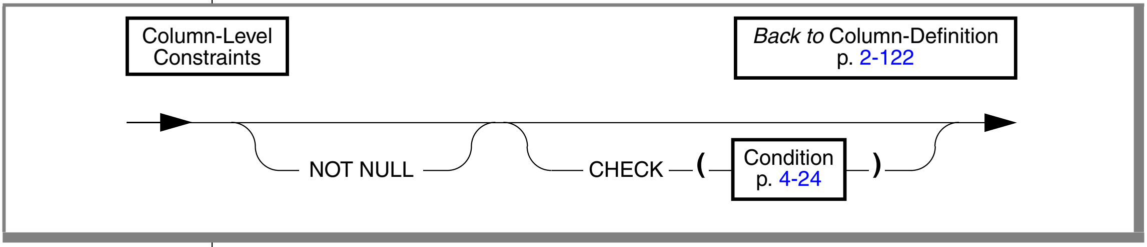
For more information on BYTE and TEXT data, see your *Administrator's Guide*.

Manipulating Data in Fixed Format Files

For files in FIXED format, you must declare the column name and the EXTERNAL item for each column to set the name and number of characters. For FIXED-format files, the only data type allowed is CHAR. You can use the keyword NULL to specify what string to interpret as a NULL value.

Column-Level Constraints

Use column-level constraints to limit the type of data that is allowed in a column. Constraints at the column level are limited to a single column.



Using the Not-Null Constraint

If you do not indicate a default value for a column, the default is NULL *unless* you place a not-null constraint on the column. In that case, no default value exists for the column. If you place a not-null constraint on a column (and no default value is specified), the data in the external table must have a value set for the column when loading through the external table.

When no reject file exists and no value is encountered, the database server returns an error and the loading stops. When a reject file exists and no value is encountered, the error is reported in the reject file and the load continues.

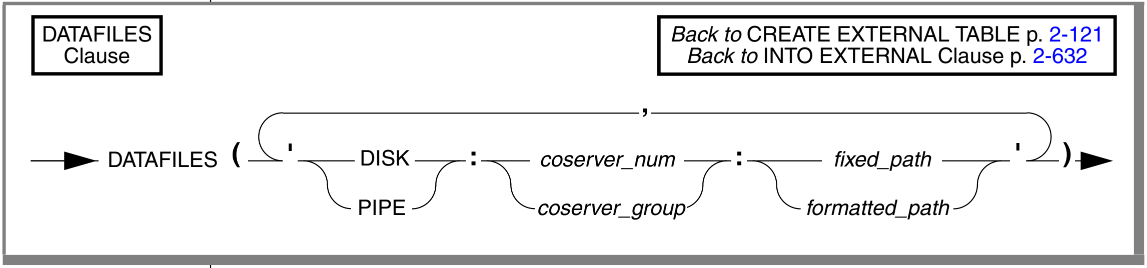
Using the CHECK Constraint

Check constraints allow you to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement. When a reject file does not exist and a row evaluates to *false* for any check constraint defined on a table during an insert or update, the database server returns an error. When there is a reject file and a row evaluates to *false* for a check constraint defined on the table, the error is reported in the reject file and the statement continues to execute.

Check constraints are defined with *search conditions*. The search condition cannot contain subqueries, aggregates, host variables, or SPL routines. In addition, it cannot include the built-in functions CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY. When you define a check constraint at the column level, the only column that the check constraint can check against is the column itself. In other words, the check constraint cannot depend upon values in other columns of the table.

DATAFILES Clause

The DATAFILES clause specifies external files that are opened when you use external tables.



Element	Purpose	Restrictions	Syntax
coserver_group	Coserver group that contains the external data	Must exist.	Identifier, p. 4-189
coserver_num	Numeric ID of coserver containing the external data	Must exist.	Literal Number, p. 4-216
fixed_path	Pathname for input or output files in the definition of the external table	Must exist.	Must conform to operating-system rules.
formatted_path	Formatted pathname that uses pattern-matching characters	Must exist.	Must conform to operating-system rules.

You can use cogroup names and coserver numbers when you describe the input or output files for the external table definition. You can identify the DATAFILES either by coserver number or by cogroup name. A coserver number contains only digits. A cogroup name is a valid identifier that begins with a letter but otherwise contains any combination of letters, digits, and underscore symbols.

If you use only some of the available coservers for reading or writing files, you can designate these coservers as a cogroup using **onutil** and then use the cogroup name, rather than explicitly specifying each coserver and file separately. Whenever you use all coservers to manage external files, you can use the predefined *coserver_group*.

For examples of the DATAFILES clause, see "Examples" on page 2-131.

Using Formatting Characters

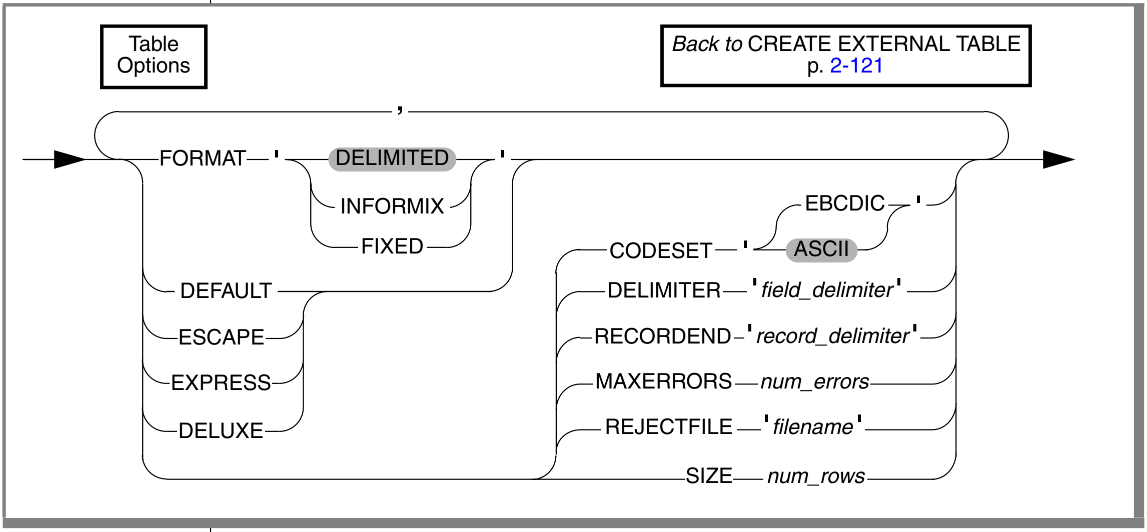
You can use a formatted pathname to designate a filename. If you use a formatted pathname, you can take advantage of the substitution characters *%c*, *%n*, and *%r(first ... last)*.

Formatting String	Effect
%c	Replaced with the number of the coserver that manages the file
%n	Replaced with the name of the node on which the coserver that manages the file resides
%r(first ... last)	Specifies multiple files on a single coserver

Important: *The formatted pathname option does not support the* %o *formatting string.*

Table Options

These options specify additional characteristics that define the table.



Element	Purpose	Restrictions	Syntax
field_delimiter	Character to separate fields. Default is pipe (|) character	For nonprinting characters, use octal	Quoted String, p. 4-243
filename	Full pathname for conversion error messages from coservers	See "Reject Files" on page 2-130.	Must conform to operating-system rules.
num_errors	Errors per coserver before load operations are terminated	Value ignored unless **MAXERRORS** is set	Literal Number, p. 4-216
num_rows	Approximate number of rows contained in the external table	Cannot be a negative number	Literal Number, p. 4-216
quoted_string	ASCII character that represents the escape	Only a single character is valid	Quoted String, p. 4-243
record_delimiter	Character to separate records. Default is Newline (\n).	For nonprinting characters, use octal	Quoted String, p. 4-243

The *num_errors* specification is ignored during unload tasks. If **MAXERRORS** environment variable is not set, the database server processes all data in load operations, regardless of the number of errors or *num_errors* value.

If the **RECORDEND** environment variable is not set, *record_delimiter* defaults to the Newline character (\n). To specify a nonprinting character as the record delimiter or field delimiter, you must encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.

Use the table options keywords as the following table describes. You can use each keyword whenever you plan to load or unload data unless only one of the two modes is specified.

Keyword	Purpose
CODESET	Specifies the type of code set of the data
DEFAULT (load only)	Specifies replacing missing values in delimited input files with column defaults (if they are defined) instead of NULLs, so input files can be sparsely populated. Files do not need an entry for every column in the file where a default is the value to be loaded.
DELIMITER	Specifies the character that separates fields in a delimited text file
DELUXE (load only)	Sets a flag causing the database server to load data in deluxe mode Deluxe mode is required for loading into STANDARD tables.
ESCAPE	Defines a character to mark ASCII special characters in ASCII-text-based data files
EXPRESS	Sets a flag that causes the database server to attempt to load data in express mode. If you request express mode but indexes or unique constraints exist on the table or the table contains BYTE or TEXT data, or the target table is not RAW or OPERATIONAL, the load stops with an error message that reports the problem.
FORMAT	Specifies the format of the data in the data files
MAXERRORS	Sets the number of errors that are allowed per coserver before the database server stops the load
RECORDEND	Specifies the character that separates records in a delimited text file
REJECTFILE	Sets the full pathname where all coservers write data-conversion errors. If not specified or if files cannot be opened, any error ends the load job abnormally. See also "Reject Files" on page 2-130.
SIZE	The approximate number of rows in the external table. This can improve performance when external table is used in a join query.

Important: *Check constraints on external tables are designed to be evaluated only when loading data. The database server cannot enforce check constraints on external tables because the data can be freely altered outside the control of the database server. If you want to restrict rows that are written to an external table during unload, use a* WHERE *clause to filter the rows.*

Reject Files

Rows that have conversion errors during a load or rows that violate check constraints on the external table are written to a reject file on the coserver that performs the conversion. Each coserver manages its own reject file. The REJECTFILE clause declares the name of the reject file on each coserver.

You can use the formatting characters `%c` and `%n` (but not `%r`) in the filename format. Use the `%c` formatting characters to make the filenames unique. For more information on how to format characters, see the section "Using Formatting Characters" on page 2-127.

If you perform another load to the same table during the same session, any earlier reject file of the same name is overwritten.

Reject file entries have the following format:

```
coserver-number, filename, record, reason-code,
   field-name: bad-line
```

The following table describes these elements of the reject file:

Element	Purpose
coserver-number	Number of the coserver from which the file is read
filename	Name of the input file
record	Record number in the input file where the error was detected
reason-code	Description of the error
field-name	External field name where the first error in the line occurred, or '<none>' if the rejection is not specific to a particular column
bad-line	Line that caused the error (delimited or fixed-position character files only): up to 80 characters

The reject file writes the *coserver-number, filename, record*, *field-name,* and *reason-code* in ASCII. The *bad-line* information varies with the type of input file.

- For delimited files or fixed-position character files, up to 80 characters of the bad-line are copied directly into the reject file.
- For Informix internal data files, the bad-line is not placed in the reject file because you cannot edit the binary representation in a file; but the Use the Table Options clause to specify the format of the external data.*coserver-number, filename, record, reason-code*, and *field-name* are still reported in the reject file so you can isolate the problem.

Errors that can cause a row to be rejected include the following.

Error Text	**Explanation**
CONSTRAINT *constraint name*	This constraint was violated.
CONVERT_ERR	Any field encounters a conversion error.
MISSING_DELIMITER	No delimiter was found.
MISSING_RECORDEND	No recordend was found.
NOT NULL	A NULL was found in *field-name*.
ROW_TOO_LONG	The input record is longer than 32 kilobytes.

Examples

The examples in this section show how to specify the DATAFILES field.

Assume that the database server is running on four nodes, and one file is to be read from each node. All files have the same name. The DATAFILES specification can then be as follows:

```
DATAFILES ("DISK:cogroup_all:/work2/unload.dir/mytbl")
```

Now, consider a system with 16 coservers where only three coservers have tape drives attached (for example, coservers 2, 5, and 9). If you define a cogroup for these coservers before you run load and unload commands, you can use the cogroup name rather than a list of individual coservers when you execute the commands. To set up the cogroup, run **onutil**.

```
% onutil
1> create cogroup tape_group
2> from coserver.2, coserver.5, coserver.9;
Cogroup successfully created.
```

Then define the file locations for named pipes:

```
DATAFILES ("PIPE:tape_group:/usr/local/TAPE.%c")
```

The filenames expand as follows:

```
DATAFILES ("pipe:2:/usr/local/TAPE.2",
      "pipe:5:/usr/local/TAPE.5",
      "pipe:9:/usr/local/TAPE.9")
```

If, instead, you want to process three files on each of two coservers, define the files as follows:

```
DATAFILES ("DISK:1:/work2/extern.dir/mytbl.%r(1..3)",
      "DISK:2:/work2/extern.dir/mytbl.%r(4..6)")
```

The expanded list follows:

```
DATAFILES ("disk:1:/work2/extern.dir/mytbl.1",
      "disk:1:/work2/extern.dir/mytbl.2",
      "disk:1:/work2/extern.dir/mytbl.3",
      "disk:2:/work2/extern.dir/mytbl.4",
      "disk:2:/work2/extern.dir/mytbl.5",
      "disk:2:/work2/extern.dir/mytbl.6")
```

Related Information

Related statements: INSERT and SET PLOAD FILE

See also the "INTO Table Clauses" of SELECT.

For more information on external tables, refer to your *Administrator's Reference*.

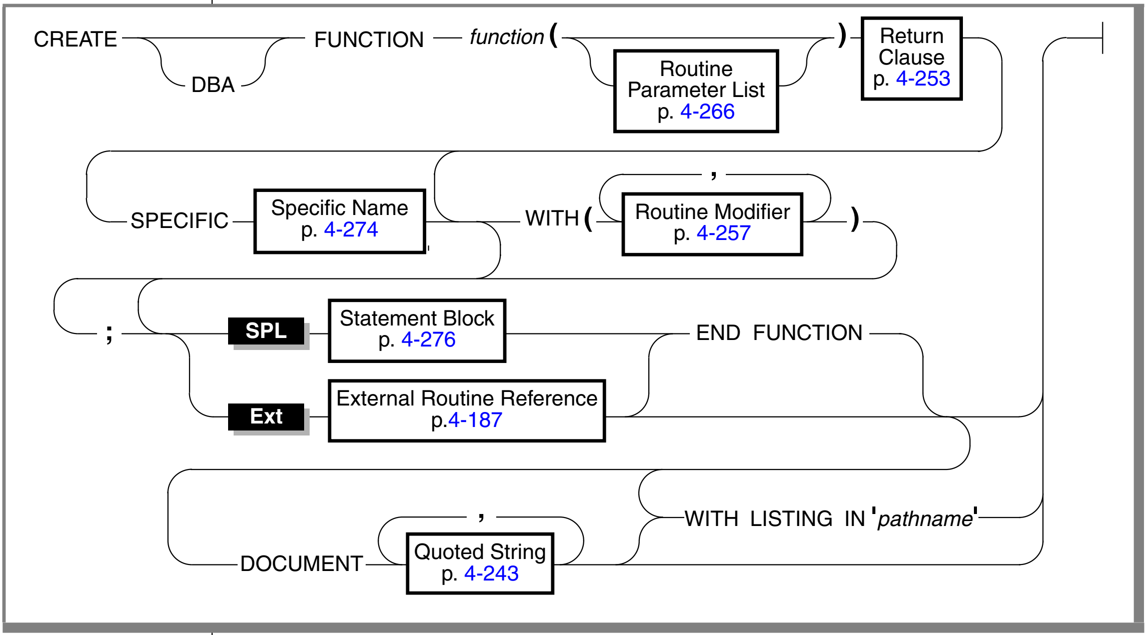
+

IDS

CREATE FUNCTION

Use the CREATE FUNCTION statement to create a user-defined function, register an external function, or to write and register an SPL function.

Syntax



Element	Purpose	Restrictions	Syntax
function	Name of new function that is defined here	You must have the appropriate language privileges. See "GRANT" on page 2-459 and "Naming a Function" on page 2-135.	Database Object Name, p. 4-46
pathname	Pathname to a file in which compile-time warnings are stored	The specified pathname must exist on the computer where the database resides.	The path and filename must conform to your operating-system rules.

***Tip:** If you are trying to create a function from text of source code that is in a separate file, use the CREATE FUNCTION FROM statement.*

Usage

The database server supports user-defined functions written in the following languages:

- Stored Procedure Language (SPL)

  An SPL function can return one or more values.

- One of the external languages (C or Java) that Dynamic Server supports (*external functions*)

  An external function must return exactly one value.

For information on how this manual uses the terms UDR, function, and procedure as well as recommended usage, see "Relationship Between Routines, Functions, and Procedures" on page 2-183 and "Using CREATE PROCEDURE Versus CREATE FUNCTION" on page 2-183, respectively.

The entire length of a CREATE FUNCTION statement must be less than 64 kilobytes. This length is the literal length of the statement, including whitespace characters such as blank spaces and tabs.

E/C

You can use a CREATE FUNCTION statement only within a PREPARE statement. If you want to create a user-defined function for which the text is known at compile time, you must put the text in a file and specify this file with the CREATE FUNCTION FROM statement. ♦

IDS

Functions use the collating order that was in effect when they were created. See SET COLLATION for information about using non-default collation ♦

Privileges Necessary for Using CREATE FUNCTION

You must have the Resource privilege on a database to create a function within that database.

Ext

Before you can create an external function, you must also have the Usage privilege on the language in which you will write the function. For more information, see "GRANT" on page 2-459. ♦

SPL

By default, the Usage privilege on SPL is granted to PUBLIC. You must also have at least the Resource privilege on a database to create an SPL function within that database. ♦

DBA Keyword and Privileges on the Created Function

The level of privilege necessary to execute a UDR depends on whether the UDR is created with the DBA keyword.

If you create a UDR with the DBA keyword, it is known as a DBA-privileged UDR. You need the DBA privilege to create or execute a DBA-privileged UDR.

If you omit the DBA keyword, the UDR is known as an owner-privileged UDR.

ANSI

If you create an owner-privileged UDR in an ANSI-compliant database, anyone can execute the UDR. ♦

If you create an owner-privileged UDR in a database that is not ANSI compliant, the NODEFDAC environment variable prevents privileges on that UDR from being granted to PUBLIC. If this environment variable is set, the owner of a UDR must grant the Execute privilege for that UDR to other users.

Ext

If an external function has a negator function, you must grant the Execute privilege on both the external function and its negator function before users can execute the external function. ♦

Naming a Function

Because Dynamic Server offers *routine overloading*, you can define more than one function with the same name, but different parameter lists. You might want to overload functions in the following situations:

- You create a user-defined function with the same name as a built-in function (such as **equal( )**) to process a new user-defined data type.
- You create *type hierarchies*, in which subtypes inherit data representation and functions from supertypes.
- You create *distinct types*, which are data types that have the same internal storage representation as an existing data type, but have different names and cannot be compared to the source type without casting. Distinct types inherit support functions from their source types.

For a brief description of the routine signature that uniquely identifies each user-defined function, see "Routine Overloading and Naming UDRs with a Routine Signature" on page 4-48.

Using the SPECIFIC Clause to Specify a Specific Name

You can specify a specific name for a user-defined function. A specific name is a name that is unique in the database. A specific name is useful when you are overloading a function.

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of the UDR. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the UDR. Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all of the UDRs stored in the database.

For example, the following query obtains a description of the SPL function **update_by_pct**, that "SPL Functions" on page 2-137 shows:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid
      --join between the two catalog tables
   AND p.procname = 'update_by_pct'
      -- look for procedure named update_by_pct
   AND b.datakey  = 'D'-- want user document;
```

The preceding query returns the following text:

```
USAGE: Update a price by a percentage
Enter an integer percentage from 1 - 100
and a part id number
```

A UDR or application program can query the system catalog tables to fetch the DOCUMENT clause and display it for a user.

Ext

You can use a DOCUMENT clause at the end of the CREATE FUNCTION statement, whether or not you use the END FUNCTION keywords. ♦

WITH LISTING IN Clause

The WITH LISTING IN clause specifies a filename where compile time warnings are sent. After you compile a UDR, this file holds one or more warning messages.

If you do not use the WITH LISTING IN clause, the compiler does not generate a list of warnings.

UNIX

If you specify a filename but not a directory, this listing file is created in your home directory on the computer where the database resides. If you do not have a home directory on this computer, the file is created in the root directory (the directory named "/"). ♦

Windows

If you specify a filename but not a directory, this listing file is created in your current working directory if the database is on the local computer. Otherwise, the default directory is **%INFORMIXDIR%\bin**. ♦

SPL

SPL Functions

SPL functions are UDRs written in SPL that return one or more values. To write and register an SPL function, use a CREATE FUNCTION statement. Embed appropriate SQL and SPL statements between the CREATE FUNCTION and END FUNCTION keywords. You can also follow the function with the DOCUMENT and WITH FILE IN options.

SPL functions are parsed, optimized (as far as possible), and stored in the system catalog tables in executable format. The body of an SPL function is stored in the **sysprocbody** system catalog table. Other information about the function is stored in other system catalog tables, including **sysprocedures**, **sysprocplan**, and **sysprocauth**. For more information about these system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

The END FUNCTION keywords are required in every SPL function, and a semicolon (**;**) must follow the clause that immediately precedes the statement block. The following code example creates an SPL function:

```
CREATE FUNCTION update_by_pct ( pct INT, pid CHAR(10))
   RETURNING INT;
   DEFINE n INT;
   UPDATE inventory SET price = price + price * (pct/100)
      WHERE part_id = pid;
   LET n = price;
   RETURN price;
END FUNCTION
   DOCUMENT "USAGE: Update a price by a percentage",
         "Enter an integer percentage from 1 - 100",
         "and a part id number"
   WITH LISTING IN '/tmp/warn_file'
```

For more information on how to write SPL functions, see the chapter about SPL routines in *IBM Informix Guide to SQL: Tutorial*.

See also the section "Transactions in SPL Routines" on page 4-280.

You can include valid SQL or SPL language statements in SPL functions. See, however, the following sections in Chapter 4 that describe restrictions on SQL and SPL statements within SPL routines: "Subset of SPL Statements Valid in the Statement Block" on page 4-276; "SQL Statements Not Valid in an SPL Statement Block" on page 4-277; and "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279.

Ext

External Functions

External functions are functions you write in an external language (that is, a programming language other than SPL) that Dynamic Server supports.

C

To create a C user-defined function

1. Write the C function.
2. Compile the function and store the compiled code in a shared library (the shared-object file for C).
3. Register the function in the database server with the CREATE FUNCTION statement.

Java

To create a user-defined function written in the Java language

1. Write a Java static method, which can use the JDBC functions to interact with the database server.
2. Compile the Java source file and create a **.jar** file (the shared-object file for Java).
3. Execute the **install_jar( )** procedure with the EXECUTE PROCEDURE statement to install the jar file in the current database.
4. If the UDR uses user-defined types, create a map between SQL data types and Java classes.

   Use the **setUDTExtName( )** procedure that is explained in "EXECUTE PROCEDURE" on page 2-414.
5. Register the UDR with the CREATE FUNCTION statement.

Rather than storing the body of an external routine directly in the database, the database server stores only the pathname of the shared-object file that contains the compiled version of the routine. When it executes the external routine, the database server invokes the external object code.

The database server stores information about an external function in system catalog tables, including **sysprocbody** and **sysprocauth**. For more information on the system catalog, see the *IBM Informix Guide to SQL: Reference*.

C

Example of Registering a C User-Defined Function

The following example registers an external C user-defined function named **equal( )** in the database. This function takes two arguments of the type **basetype1** and returns a single Boolean value. The external routine reference name specifies the path to the C shared library where the function object code is actually stored. This library contains a C function **basetype1_equal( )**, which is invoked during execution of the **equal( )** function.

```
CREATE FUNCTION equal ( arg1 basetype1, arg2 basetype1)
RETURNING BOOLEAN;
EXTERNAL NAME
"/usr/lib/basetype1/lib/libbtype1.so(basetype1_equal)"
LANGUAGE C
END FUNCTION
```

Java

Example of Registering a User-Defined Function Written in the Java Language

The following CREATE FUNCTION statement registers the user-defined function, **sql_explosive_reaction( )**. This function is discussed in "sqlj.install_jar" on page 2-418.

```
CREATE FUNCTION sql_explosive_reaction(int) RETURNS int
   WITH (class="jvp")
   EXTERNAL NAME "course_jar:Chemistry.explosiveReaction"
   LANGUAGE JAVA
```

This function returns a single INTEGER value. The EXTERNAL NAME clause specifies that the Java implementation of the **sql_explosive_reaction( )** function is a method called **explosiveReaction( )**, which resides in the **Chemistry** Java class that resides in the **course_jar** jar file.

Ownership of Created Database Objects

The user who creates an owner-privileged UDR owns any database objects that are created by the UDR when the UDR is executed, unless another owner is specified for the created database object. In other words, the UDR owner, not the user who executes the UDR, is the owner of any database objects created by the UDR unless another owner is specified in the statement that creates the database object.

For example, assume that user **mike** creates this user-defined function:

```
CREATE FUNCTION func1 () RETURNING INT;
   CREATE TABLE tab1 (colx INT);
   RETURN 1;
END FUNCTION
```

If user **joan** now executes function **func1**, user **mike**, not user **joan**, is the owner of the newly created table **tab1**.

In the case of a DBA-privileged UDR, however, the user who executes a UDR (rather than the UDR owner) owns any database objects created by the UDR, unless another owner is specified for the database object within the UDR.

For example, assume that user **mike** creates this user-defined function:

```
CREATE DBA FUNCTION func2 () RETURNING INT;
   CREATE TABLE tab2 (coly INT);
   RETURN 1;
END FUNCTION
```

If user **joan** now executes function **func2**, user **joan**, not user **mike**, is the owner of the newly created table **tab2**.

See also the section "Support for Roles and User Identity" on page 4-280.

Related Information

Related statements: ALTER FUNCTION, ALTER ROUTINE, CREATE PROCEDURE, CREATE FUNCTION FROM, DROP FUNCTION, DROP ROUTINE, GRANT, EXECUTE FUNCTION, PREPARE, REVOKE, and UPDATE STATISTICS

Chapter 3 of this manual describes the syntax of the SPL language. For a discussion on how to create and use SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion on how to create and use external routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For information about how to create C UDRs, see the *IBM Informix DataBlade API Programmer's Guide*.

For more information on the **NODEFDAC** environment variable and the related system catalog tables (**sysprocedures**, **sysprocplan**, **sysprocbody** and **sysprocauth)**, see the *IBM Informix Guide to SQL: Reference*.

+

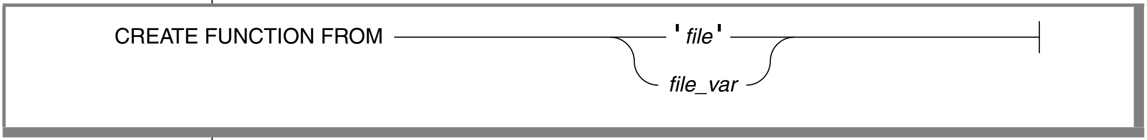
IDS

E/C

CREATE FUNCTION FROM

Use the CREATE FUNCTION FROM statement to access a user-defined function whose CREATE FUNCTION statement resides in a separate file.

Syntax



Element	**Purpose**	**Restrictions**	**Syntax**
file	Path and filename of a file that contains the full CREATE FUNCTION statement text. Default pathname is current directory.	Must exist and contain exactly one CREATE FUNCTION statement.	Must conform to operating-system rules.
file_var	Variable storing value of *file*	Same as for *file*.	Language specific

Usage

An ESQL/C program cannot directly create a user-defined function. That is, it cannot contain the CREATE FUNCTION statement.

To create these functions within an ESQL/C program

1. Create a source file with the CREATE FUNCTION statement.
2. Use the CREATE FUNCTION FROM statement to send the contents of this source file to the database server for execution.

   The file that you specify in the *file* parameter can contain only one CREATE FUNCTION statement.

For example, suppose that the following CREATE FUNCTION statement is in a separate file, called **del_ord.sql**:

```
CREATE FUNCTION delete_order( p_order_num int)
   RETURNING int, int;
   DEFINE item_count int;
   SELECT count(*) INTO item_count FROM items
      WHERE order_num = p_order_num;
   DELETE FROM orders WHERE order_num = p_order_num;
   RETURN p_order_num, item_count;
END FUNCTION;
```

In the ESQL/C program, you can access the **delete_order()** SPL function with the following CREATE FUNCTION FROM statement:

```
EXEC SQL create function from 'del_ord.sql';
```

If you are not sure whether the UDR in the file is a user-defined function or a user-defined procedure, use the CREATE ROUTINE FROM statement.

The filename that you provide is relative. If you provide a simple filename with no pathname (as in the preceding example), the client application looks for the file in the current directory.

Important: *The ESQL/C preprocessor does not process the contents of the file that you specify. It just sends the contents to the database server for execution. Therefore, there is no syntactic check that the file that you specify in CREATE FUNCTION FROM actually contains a CREATE FUNCTION statement. To improve readability of the code, however, It is recommended that you match these two statements.*

Related Information

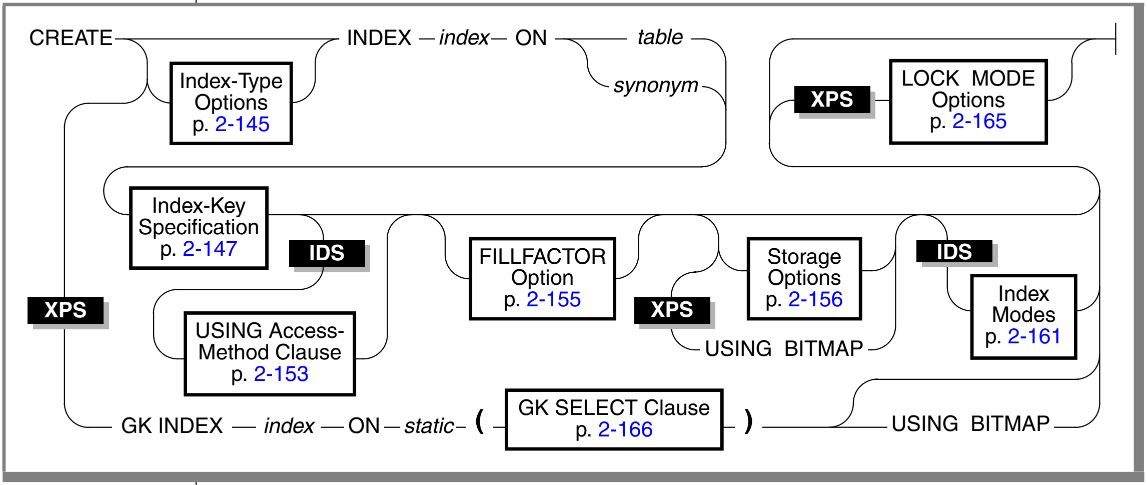
Related statements: CREATE FUNCTION, CREATE PROCEDURE, CREATE PROCEDURE FROM, and CREATE ROUTINE FROM

+

CREATE INDEX

Use the CREATE INDEX statement to create an index for one or more columns in a table, or on values returned by a UDR using columns as arguments.

Syntax


CREATE
INDEX
index
ON
table
synonym
Index-Type Options p. 2-145
XPS
LOCK MODE Options p. 2-165
Index-Key Specification p. 2-147
IDS
USING Access-Method Clause p. 2-153
XPS
FILLFACTOR Option p. 2-155
XPS
Storage Options p. 2-156
USING BITMAP
IDS
Index Modes p. 2-161
GK INDEX
index
ON
static
(
GK SELECT Clause p. 2-166
)
USING BITMAP

Element	Purpose	Restrictions	Syntax
index	Name declared here for a new index	Must be unique among index names in the database	Database Object Name, p. 4-46
static	Table on which a Generalized Key index is created	Table must exist and be static; it cannot be a virtual table	Database Object Name, p. 4-46
synonym, *table*	Name or synonym of a standard or temporary *table* to be indexed	Synonym and its table must exist in the current database	Database Object Name, p. 4-46

Usage

When you issue the CREATE INDEX statement, the table is locked in exclusive mode. If another process is using the table, CREATE INDEX returns an error.

IDS

Indexes use the collation that was in effect when CREATE INDEX executed. ♦

A *secondary-access method* (sometimes referred to as an *index-access method*) is a set of database server functions that build, access, and manipulate an index structure such as a B-tree, R-tree, or an index structure that a DataBlade module provides, in order to speed up the retrieval of data.

IDS

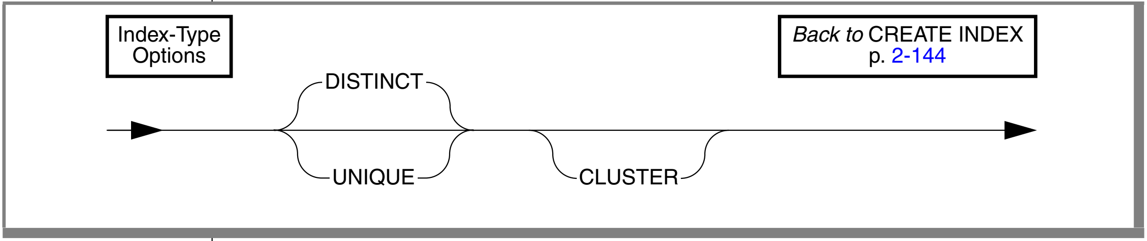
Neither *synonym* nor *table* can refer to a virtual table. ♦

XPS

If you are using Extended Parallel Server, use the USING BITMAP keywords to store the list of records in each key of the index as a compressed bitmap. The storage option is not compatible with a bitmap index because bitmap indexes must be fragmented in the same way as the table. ♦

Index-Type Options

The index-type options let you specify the characteristics of the index.



UNIQUE or DISTINCT Option

Use the UNIQUE or DISTINCT keyword to require that the column(s) on which the index is based accepts only unique data. If you do not specify the UNIQUE or DISTINCT keyword, the index allows duplicate values in the indexed column. The following example creates a unique index:

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
```

A unique index prevents duplicate values in the **customer_num** column. A column with a unique index can have, at most, one NULL value.

The DISTINCT keyword is a synonym for the keyword UNIQUE, so the following statement has the same effect as the previous example:

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num)
```

The index in both examples is maintained in ascending order, which is the default order.

You can also prevent duplicates in a column or set of columns by creating a unique constraint with the CREATE TABLE or ALTER TABLE statement. You cannot specify an R-tree secondary-access method for a UNIQUE index key. For more information on how to create unique constraints, see the CREATE TABLE or ALTER TABLE statements.

How Indexes Affect Primary-Key, Unique, and Referential Constraints

The database server creates internal B-tree indexes for primary-key, unique, and referential constraints. If a primary-key, unique, or referential constraint is added after the table is created, any user-created indexes on the constrained columns are used, if appropriate. An appropriate index is one that indexes the same columns that are used in the primary-key, referential, or unique constraint. If an appropriate user-created index is not available, the database server creates a nonfragmented internal index on the constrained column or columns.

CLUSTER Option

Use the CLUSTER keyword to reorder the rows of the table in the order that the index designates. The CREATE CLUSTER INDEX statement fails if a CLUSTER index already exists on the same table.

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode)
```

This statement creates an index on the **customer** table that physically orders the table by zip code.

If the CLUSTER option is specified in addition to fragments on an index, the data is clustered only within the context of the fragment and not globally across the entire table.

IDS

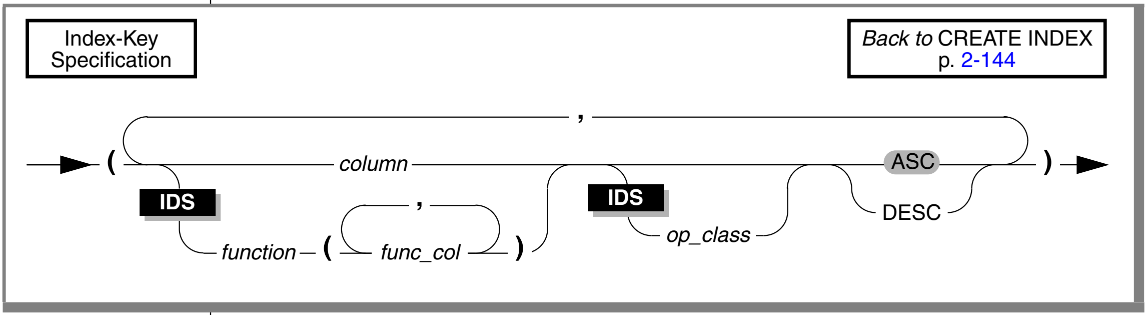
Some secondary-access methods (such as R-tree) do not support clustering. Before you specify CLUSTER for your index, be sure that it uses an access method that supports clustering. ♦

XPS

If you are using Extended Parallel Server, you cannot use the CLUSTER option on STANDARD tables. In addition, you cannot use the CLUSTER option and storage options in the same CREATE INDEX statement (see "Storage Options" on page 2-156). When you create a clustered index the **constrid** of any unique or referential constraints on the associated table changes. The **constrid** is stored in the **sysconstraints** system catalog table. ♦

Index-Key Specification

Use the Index-Key Specification portion of the CREATE INDEX statement to specify the key value for the index, an operator class, and whether the index will be sorted in ascending or descending order.



Element	Purpose	Restrictions	Syntax
column	Column or columns used as a key to this index	See "Using a Column as the Index Key" on page 2-148.	Identifier, p. 4-189
function	User-defined function used as a key to this index	Must be a nonvariant function that does not return a large object data type. Cannot be a built-in algebraic, exponential, log, or hex function.	Database Object Name, p. 4-46
func_col	Column(s) on which the user-defined function acts	See "Using a Column as the Index Key" on page 2-148.	Identifier, p. 4-189
op_class	Operator class associated with *column* or *function* for this index	If the secondary-access method in the USING clause has no default operator class, you must specify one here. (See "Using an Operator Class" on page 2-152.)	Identifier, p. 4-189

The index-key value can be one or more columns that contain built-in data types. When multiple columns are listed, the concatenation of the set of columns is treated as a single composite column for indexing.

IDS

In addition, the index-key value can be one of the following types:

- A column of type LVARCHAR(*size*), if *size* is fewer than 387 bytes
- One or more columns that contain user-defined data types
- One or more values that a user-defined function returns (referred to as a *functional index*)
- A combination of columns and functions ♦

Using a Column as the Index Key

These restrictions apply to a column or columns specified as the index key:

- All the columns must exist and must be in the table being indexed.
- The maximum number of columns and total width of all columns depends on the database server. See "Creating Composite Indexes" on page 2-149.
- You cannot add an ascending index to a column or column list that already has a unique constraint on it. See "Using the ASC and DESC Sort-Order Options" on page 2-149.
- You cannot add a unique index to a column or column list that has a primary-key constraint on it. The reason is that defining the column or column list as the primary key causes the database server to create a unique internal index on the column or column list. So you cannot create another unique index on this column or column list with the CREATE INDEX statement.
- The number of indexes that you can create on the same column or the same set of columns is restricted. See "Restrictions on the Number of Indexes on a Set of Columns" on page 2-152.

IDS

- You cannot create an index on a column of an external table.
- The column cannot be of a collection data type. ♦

IDS

Using a Function as an Index Key

You can create functional indexes within an SPL routine.

You can also create an index on a nonvariant user-defined function that does not return a large object.

A functional index can be a B-tree index, an R-tree index, or a user-defined secondary-access method.

Functional indexes are indexed on the value that the specified function returns, rather than on the value of a column. For example, the following statement creates a functional index on table **zones** using the value that the function **Area()** returns as the key:

```
CREATE INDEX zone_func_ind ON zones (Area(length,width));
```

Creating Composite Indexes

A *simple* index lists only one *column* (or for IDS, only one *column* or *function*) in its Index Key Specification. Any other index is a *composite* index. You should list the columns in a composite index in the order from most-frequently used to least-frequently used.

IDS

If you use SET COLLATION to specify a non-default locale, you can create multiple indexes on the same set of columns, using different collations. (Such indexes would be useful only on NCHAR or NVARCHAR columns.) ♦

The following example creates a composite index using the **stock_num** and **manu_code** columns of the **stock** table:

```
CREATE UNIQUE INDEX st_man_ix ON stock (stock_num, manu_code)
```

The UNIQUE keyword prevents any duplicates of a given combination of **stock_num** and **manu_code**. The index is in ascending order by default.

XPS

You can include up to 16 columns in a composite index. The total width of all indexed columns in a single COMPOSITE index cannot exceed 380 bytes. ♦

IDS

An *index key part* is either a column in a table, or the result of a user-defined function on one or more columns. A composite index can have up to 16 key parts that are columns, or up to 341 key parts that are values returned by a UDR. This limit is language-dependent, and applies to UDRs written in SPL or Java; functional indexes based on C language UDRs can have up to 102 key parts. A composite index can have any of the following items as an index key:

- One or more columns
- One or more values that a user-defined function returns (referred to as a functional index)
- A combination of columns and user-defined functions

The total width of all indexed columns in a single CREATE INDEX statement cannot exceed 390 bytes, except for functional indexes of Dynamic Server, whose language-dependent limits are described earlier in this section. ♦

Using the ASC and DESC Sort-Order Options

The ASC option specifies an index maintained in ascending order; this is the default order. The DESC option can specify an index that is maintained in descending order. These ASC and DESC options are valid with B-trees only.

Effects of Unique Constraints on Sort Order Options

When a column or list of columns is defined as unique in a CREATE TABLE or ALTER TABLE statement, the database server implements that UNIQUE CONSTRAINT by creating a unique ascending index. Thus, you cannot use the CREATE INDEX statement to add an ascending index to a column or column list that is already defined as unique.

However, you can create a descending index on such columns, and you can include such columns in composite ascending indexes in different combinations. For example, the following sequence of statements is valid:

```
CREATE TABLE customer (
   customer_num   SERIAL(101) UNIQUE,
   fname          CHAR(15),
   lname          CHAR(15),
   company        CHAR(20),
   address1       CHAR(20),
   address2       CHAR(20),
   city           CHAR(15),
   state          CHAR(2),
   zipcode        CHAR(5),
   phone          CHAR(18)
   )

CREATE INDEX c_temp1 ON customer (customer_num DESC)
CREATE INDEX c_temp2 ON customer (customer_num, zipcode)
```

In this example, the **customer_num** column has a unique constraint placed on it. The first CREATE INDEX statement places an index sorted in descending order on the **customer_num** column. The second CREATE INDEX includes the **customer_num** column as part of a composite index. For more information on composite indexes, see "Creating Composite Indexes" on page 2-149.

Bidirectional Traversal of Indexes

If you do not specify the ASC or DESC keywords when you create an index on a column, key values are stored in ascending order by default; but the bidirectional-traversal capability of the database server lets you create just one index on a column and use that index for queries that specify sorting of results in either ascending or descending order of the sort column.

Because of this capability, it does not matter whether you create a single-column index as an ascending or descending index. Whichever storage order you choose for an index, the database server can traverse that index in ascending or descending order when it processes queries.

If you create a composite index on a table, however, the ASC and DESC keywords might be required. For example, if you want to enter a SELECT statement whose ORDER BY clause sorts on multiple columns and sorts each column in a different order and you want to use an index for this query, you need to create a composite index that corresponds to the ORDER BY columns. For example, suppose that you want to enter the following query:

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code ASC, unit_price DESC
```

This query sorts first in ascending order by the value of the **manu_code** column and then in descending order by the value of the **unit_price** column. To use an index for this query, you need to issue a CREATE INDEX statement that corresponds to the requirements of the ORDER BY clause. For example, you can enter either of the following statements to create the index:

```
CREATE INDEX stock_idx1 ON stock
   (manu_code ASC, unit_price DESC);
CREATE INDEX stock_idx2 ON stock
   (manu_code DESC, unit_price ASC);
```

The composite index that was used for this query (**stock_idx1** or **stock_idx2**) cannot be used for queries in which you specify the same sort direction for the two columns in the ORDER BY clause. For example, suppose that you want to enter the following queries:

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code ASC, unit_price ASC;
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code DESC, unit_price DESC;
```

If you want to use a composite index to improve the performance of these queries, you need to enter one of the following CREATE INDEX statements. You can use either one of the created indexes (**stock_idx3** or **stock_idx4**) to improve the performance of the preceding queries.

```
CREATE INDEX stock_idx3 ON stock
   (manu_code ASC, unit_price ASC);
CREATE INDEX stock_idx4 ON stock
   (manu_code DESC, unit_price DESC);
```

You can create no more than one ascending index and one descending index on a single column. Because of the bidirectional-traversal capability of the database server, you only need to create one of the indexes. Creating both would achieve exactly the same results for an ascending or descending sort on the **stock_num** column.

Restrictions on the Number of Indexes on a Set of Columns

You can create multiple indexes on a set of columns, provided that each index has a unique combination of ascending and descending columns. For example, to create all possible indexes on the **stock_num** and **manu_code** columns of the **stock** table, you could create four indexes:

- The **ix1** index on both columns in ascending order
- The **ix2** index on both columns in descending order
- The **ix3** index on **stock_num** in ascending order and on **manu_code** in descending order
- The **ix4** index on **stock_num** in descending order and on **manu_code** in ascending order

Because of the bidirectional-traversal capability of the database server, you do not need to create these four indexes. You only need to create two indexes:

- The **ix1** and **ix2** indexes achieve the same results for sorts in which the user specifies the same sort direction (ascending or descending) for both columns, so you only need one index of this pair.
- The **ix3** and **ix4** indexes achieve the same results for sorts in which the user specifies different sort directions for the two columns (ascending on the first column and descending on the second column or vice versa). Thus, you only need to create one index of this pair. (See also "Bidirectional Traversal of Indexes" on page 2-150.)

IDS

Dynamic Serve can also suppport multiple indexes on the same combination of ascending and descending columns, if each index has a different collating order; see "SET COLLATION" on page 2-643.

IDS

Using an Operator Class

An *operator class* is the set of operators associated with a secondary-access method for query optimization and building the index. You must specify an operator class when you create an index if either one of the following is true:

- No default operator class for the secondary-access method exists. (A user-defined access method can provide no default operator class.)
- You want to use an operator class that is different from the default operator class that the secondary-access method provides.

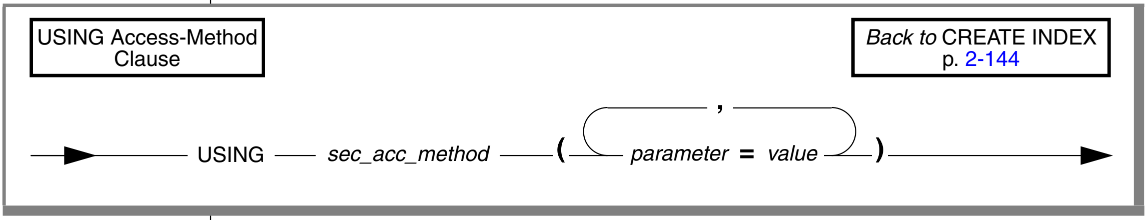
If you use an alternative access method, and if the access method has a default operator class, you can omit the operator class here; but if you do not specify an operator class and the secondary-access method does not have a default operator class, the database server returns an error. For more information, see "Default Operator Classes" on page 2-180. The following CREATE INDEX statement creates a B-tree index on the **cust_tab** table that uses the **abs_btree_ops** operator class for the **cust_num** key:

```
CREATE INDEX c_num1_ix ON cust_tab (cust_num abs_btree_ops);
```

IDS

USING Access-Method Clause

The USING clause specifies the secondary-access method for the new index.



Element	Purpose	Restrictions	Syntax
parameter	Secondary-access-method parameter for this index	See the user documentation for your user-defined access method.	Quoted String, p. 4-243
sec_acc_method	Secondary-access method for this index	Method can be a B-tree, R-tree, or user-defined access method, such as one that a DataBlade module defines.	Identifier, p. 4-189
value	Value of the specified *parameter*	Must be a valid literal value for *parameter* in this secondary-access method.	Quoted String, p. 4-243 or Literal Number, p. 4-216

A *secondary-access method* is a set of routines that perform all of the operations needed for an index, such as create, drop, insert, delete, update, and scan.

The database server provides the following secondary-access methods:

- The generic B-tree index is the built-in secondary-access method.

  A B-tree index is good for a query that retrieves a range of data values. The database server implements this secondary-access method and registers it as **btree** in the system catalog tables.
- The R-tree method is a registered secondary-access method.

  An R-tree index is good for searches on multidimensional data. The database server registers this secondary-access method as **rtree** in the system catalog tables of a database. An R-tree secondary-access method is not valid for a UNIQUE index key. For more information on R-tree indexes, see the *IBM Informix R-Tree Index User's Guide*.

The access method that you specify must be a valid access method in the **sysams** system catalog table. The default secondary-access method is B-tree.

If the access method is B-tree, you can create only one index for each unique combination of ascending and descending columnar or functional keys with operator classes. (This does not apply to other secondary-access methods.) By default, CREATE INDEX creates a generic B-tree index. If you want to create an index with a secondary-access method other than B-tree, you must specify the name of the secondary-access method in the USING clause.

Some user-defined access methods are packaged as DataBlades. Some DataBlade modules provide indexes that require specific parameters when you create them. For more information about user-defined access methods, refer to your access-method or DataBlade documentation.

The following example (for a database that implements R-tree indexes) creates an R-tree index on the **location** column that contains an opaque data type, **point**, and performs a query with a filter on the **location** column.

```
CREATE INDEX loc_ix ON TABLE emp (location) USING rtree;
SELECT name FROM emp WHERE location N_equator_equals point('500, 0');
```

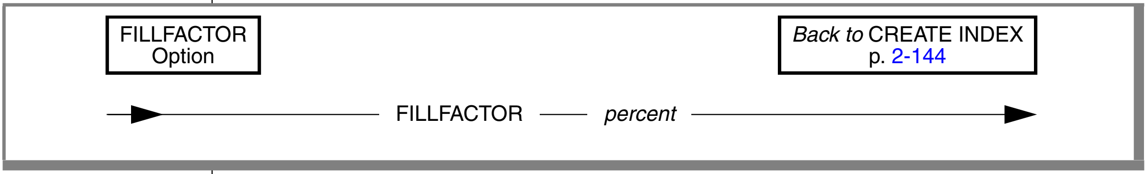
The following CREATE INDEX statement creates an index that uses the secondary-access method **fulltext**, which takes two parameters: WORD_SUPPORT and PHRASE_SUPPORT. It indexes a table **t**, which has two columns: **i**, an integer column, and **data**, a TEXT column.

```
CREATE INDEX tx ON t(data)
   USING fulltext (WORD_SUPPORT=‘PATTERN’,
   PHRASE_SUPPORT=’MAXIMUM’);
```

FILLFACTOR Option

The FILLFACTOR option takes effect only when you build an index on a table that contains more than 5,000 rows and uses more than 100 table pages, when you create an index on a fragmented table, or when you create a fragmented index on a nonfragmented table.

Use the FILLFACTOR option to provide for expansion of an index at a later date or to create compacted indexes.



Element	Purpose	Restrictions	Syntax
percent	Percentage of each index page that is filled by index data when the index is created. The default is `90`.	1 ≤ *percent* ≤ `100`	Literal Number, p. 4-216

When the index is created, the database server initially fills only that percentage of the nodes specified with the FILLFACTOR value.

The FILLFACTOR can also be set as a parameter in the ONCONFIG file. The FILLFACTOR clause on the CREATE INDEX statement overrides the setting in the ONCONFIG file. For more information about the ONCONFIG file and the parameters you can use, see your *Administrator's Guide*.

Providing a Low Percentage Value

If you provide a low percentage value, such as `50`, you allow room for growth in your index. The nodes of the index initially fill to a certain percentage and contain space for inserts. The amount of available space depends on the number of keys in each page as well as the percentage value.

For example, with a 50-percent FILLFACTOR value, the page would be half full and could accommodate doubling in size. A low percentage value can result in faster inserts and can be used for indexes that you expect to grow.

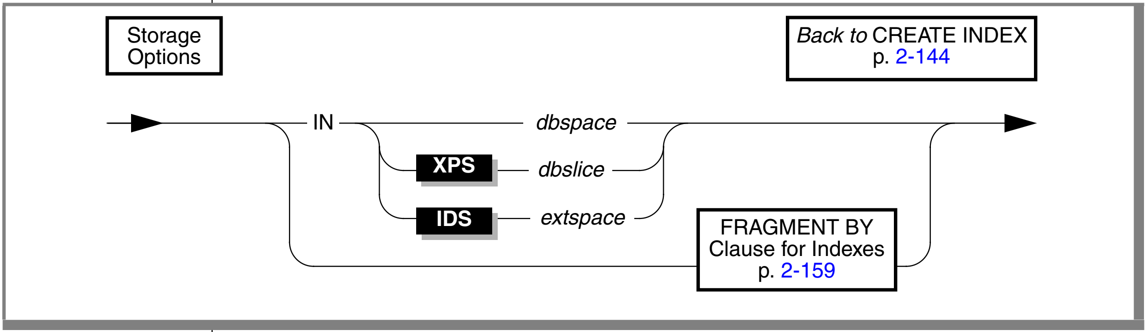
Providing a High Percentage Value

If you provide a high percentage value, such as `99`, your indexes are compacted, and any new index inserts result in splitting nodes. The maximum density is achieved with 100 percent. With a 100-percent FILLFACTOR value, the index has no room available for growth; any additions to the index result in splitting the nodes.

A 99-percent FILLFACTOR value allows room for at least one insertion per node. A high percentage value can result in faster selects and can be used for indexes that you do not expect to grow or for mostly read-only indexes.

Storage Options

The storage options specify the distribution scheme of an index. You can use the IN clause to specify a storage space for the entire index, or you can use the FRAGMENT BY clause to fragment the index across multiple storage spaces.


Storage Options
Back to CREATE INDEX p. 2-144
IN
dbspace
XPS
dbslice
IDS
extspace
FRAGMENT BY Clause for Indexes p. 2-159

Element	Purpose	Restrictions	Syntax
dbslice	The dbslice that contains all of the index fragments	Must exist.	Identifier, p. 4-189
dbspace	The dbspace in which to store the index	Must exist.	Identifier, p. 4-189
extspace	Name assigned by the **onspaces** command to a storage area outside the database server	Must exist.	See the documentation for your access method.

When you specify one of the storage options, you create a *detached index*. Detached indexes are indexes that are created with a specified distribution scheme. Even if the distribution scheme specified for the index is identical to that specified for the table, the index is still considered to be detached. If the distribution scheme of a table changes, all detached indexes continue to use their own distribution scheme.

XPS

For information on locally-detached and globally-detached indexes, see "FRAGMENT BY Clause for Indexes" on page 2-159. If you are using Extended Parallel Server, you cannot use the CLUSTER option and storage options in the same CREATE INDEX statement. See "CLUSTER Option" on page 2-146. ♦

IDS

In some earlier releases of Dynamic Server, if you did not use the storage options to specify a distribution scheme, then by default the index inherited the distribution scheme of the table on which it was built. Such an index is called an *attached index*. In this release, CREATE INDEX creates new indexes as detached indexes by default, but supports existing attached indexes that were created by earlier release versions. An attached index is created in the same dbspace (or dbspaces, if the table is fragmented) as the table on which it is built. If the distribution scheme of a table changes, all attached indexes start using the new distribution scheme.

Only B-tree indexes that are nonfragmented and that are on nonfragmented tables can be attached. All other indexes, including extensibility related indexes, such as R-trees and UDT indexes, must be detached. You cannot create an attached index using a collating order different from that of the table, nor different from what **DB_LOCALE** specifies. For information on how to create attached indexes, see the description of the **DEFAULT_ATTACH** environment variable in *IBM Informix Guide to SQL: Reference*. ♦

IN Clause

Use the IN clause to specify a storage space to hold the entire index. The storage space that you specify must already exist.

Use the IN *dbspace* clause to specify the dbspace where you want your index to reside. When you use this clause, you create a detached index.

The IN *dbspace* clause allows you to isolate an index. For example, if the **customer** table is created in the **custdata** dbspace, but you want to create an index in a separate dbspace called **custind**, use the following statements:

```
CREATE TABLE customer
   . . .
   IN custdata EXTENT SIZE 16

CREATE INDEX idx_cust ON customer (customer_num)
   IN custind
```

XPS

Storing an Index in a dbslice

If you are using Extended Parallel Server, the IN *dbslice* clause allows you to fragment an index across multiple dbspaces. The database server fragments the table by round-robin in the dbspaces that make up the dbslice at the time when the table is created.

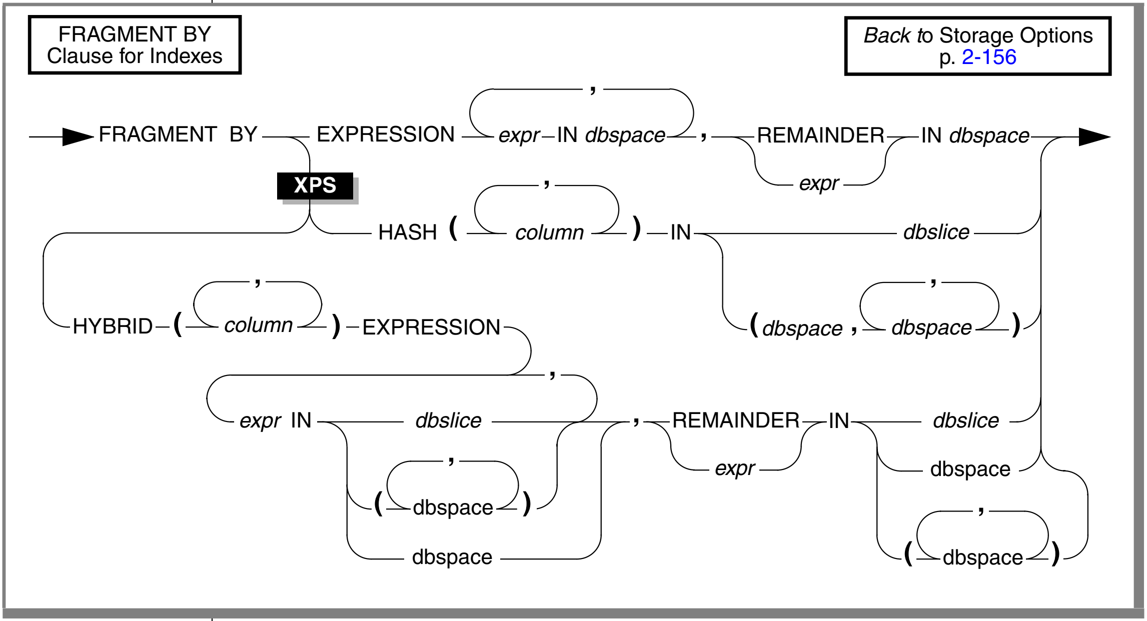
IDS

Storing an Index in an extspace

In general, use this option in conjunction with the "USING Access-Method Clause" on page 2-153. You can also store an index in an sbspace. For more information, refer to the user documentation for your custom access method.

FRAGMENT BY Clause for Indexes

Use the FRAGMENT BY clause to create a detached index and to define its fragmentation strategy across multiple dbspaces.



Element	Purpose	Restrictions	Syntax
column	Column on which to fragment the index	Must exist in the current table	Identifier, p. 4-189
dbslice	Dbslice storing all the index fragments	Must exist	Identifier, p. 4-189
dbspace	Dbspace in which to store the index fragment that *expr* defines	You must specify at least 2, but no more than 2,048 *dbspace* names	Identifier, p. 4-189
expr	Expression defining which index keys each fragment stores	See "Restrictions on Fragmentation Expressions," page 2-160.	Expression, p. 4-67; Condition, p. 4-24

To specify a fragmented index, the IN keyword introduces the storage space where an index fragment is to be stored. If you list multiple *dbspace* names after the IN keyword, use parentheses to delimit the *dbspace* list.

Restrictions on Fragmentation Expressions

The following restrictions apply to the expression:

- Each fragment expression can contain columns only from the current table, with data values only from a single row.
- The columns contained in a fragment expression must be the same as the indexed columns or a subset of the indexed columns.
- The expression must return a Boolean (true or false) value.
- No subqueries, aggregates, user-defined routines, nor references to fields of a ROW type column are allowed.
- The built-in CURRENT, DATE, and TIME functions are not allowed.

XPS

You can fragment indexes on any column of a table, even if the table spans multiple coservers. The columns that you specify in the FRAGMENT BY clause do not have to be part of the index key.

Detached indexes can be either locally detached or globally detached. A *locally detached index* is an index in which, for each data tuple in a table, the corresponding index tuple is guaranteed to be on the same coserver. The table and index fragmentation strategies do not have to be identical as long as co-locality can be guaranteed. If the data tuple and index tuple co-locality do not exist, then the index is a *globally-detached index*. For performance implications of globally-detached indexes, see your *Performance Guide*.

For more information on expression, hash, and hybrid distribution schemes, see "Fragmenting by EXPRESSION" on page 2-239, "Fragmenting by HASH" on page 2-242, and "Fragmenting by HYBRID" on page 2-243, respectively, in the description of the CREATE TABLE statement. ♦

Fragmentation of System Indexes

System indexes (such as those used in referential constraints and unique constraints) utilize user indexes if they exist. If no user indexes can be utilized, system indexes remain nonfragmented and are moved to the dbspace where the database was created.

To fragment a system index, create the fragmented index on the constraint columns, and then add the constraint using the ALTER TABLE statement.

IDS

Fragmentation of Unique Indexes

You can fragment unique indexes only with a table that uses an expression-based distribution scheme. The columns referenced in the fragment expression must be part of the indexed columns. If your CREATE INDEX statement fails to meet either of these restrictions, the CREATE INDEX fails, and work is rolled back.

Fragmentation of Indexes on Temporary Tables

You can fragment a unique index on a temporary table only if the underlying table uses an expression-based distribution scheme. That is, the CREATE Temporary TABLE statement that defines the temporary table must specify an explicit expression-based distribution scheme.

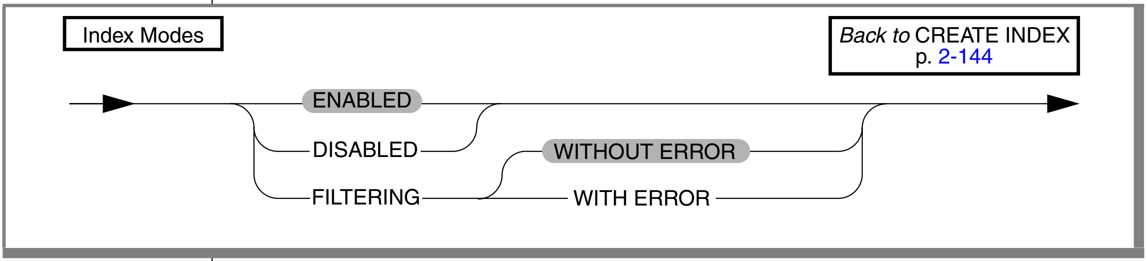
If you try to create a fragmented, unique index on a temporary table for which you did not specify a fragmentation strategy when you created the table, the database server creates the index in the first dbspace that the **DBSPACETEMP** environment variable specifies.For more information on the **DBSPACETEMP** environment variable, see the *IBM Informix Guide to SQL: Reference*.

For more information on the default storage characteristics of temporary tables, see "Where Temporary Tables are Stored" on page 2-266.

IDS

Index Modes

Use the index modes to control the behavior of the index during INSERT, DELETE, and UPDATE operations.



The following table explains the index modes.

Mode	Purpose
DISABLED	The database server does not update the index after insert, delete, and update operations that modify the base table. The optimizer does not use the index during the execution of queries.
ENABLED	The database server updates the index after insert, delete, and update operations that modify the base table. The optimizer uses the index during query execution. If an insert or update operation causes a duplicate key value to be added to a unique index, the statement fails.
FILTERING	The database server updates a unique index after insert, delete, and update operations that modify the base table. (This option is not available with duplicate indexes.) The optimizer uses the index during query execution. If an insert or update operation causes a duplicate key value to be added to a unique index in filtering mode, the statement continues processing, but the bad row is written to the violations table associated with the base table. Diagnostic information about the unique-index violation is written to the diagnostics table associated with the base table.

If you specify filtering for a unique index, you can also specify one of the following error options.

Error Option	Purpose
WITHOUT ERROR	A unique-index violation during an insert or update operation returns no integrity-violation error to the user.
WITH ERROR	Any unique-index violation during an insert or update operation returns an integrity-violation error to the user.

Specifying Modes for Unique Indexes

You must observe the following rules when you specify modes for unique indexes in CREATE INDEX statements:

- You can set the mode of a unique index to enabled, disabled, or filtering.
- If you do not specify a mode, then by default the index is enabled.
- For an index set to filtering mode, if you do not specify an error option, the default is WITHOUT ERROR.
- When you add a new unique index to an existing base table and specify the disabled mode for the index, your CREATE INDEX statement succeeds even if duplicate values in the indexed column would cause a unique-index violation.
- When you add a new unique index to an existing base table and specify the enabled or filtering mode for the index, your CREATE INDEX statement succeeds provided that no duplicate values exist in the indexed column that would cause a unique-index violation. However, if any duplicate values exist in the indexed column, your CREATE INDEX statement fails and returns an error.
- When you add a new unique index to an existing base table in the enabled or filtering mode, and duplicate values exist in the indexed column, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.

Adding a Unique Index When Duplicate Values Exist in the Column

If you attempt to add a unique index in the enabled mode but receive an error message because duplicate values are in the indexed column, take the following steps to add the index successfully:

1. Add the index in the disabled mode. Issue the CREATE INDEX statement again, but this time specify the DISABLED keyword.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.

3. Issue a SET Database Object Mode statement to switch the mode of the index to enabled. When you issue this statement, existing rows in the target table that violate the unique-index requirement are duplicated in the violations table. However, you receive an integrity-violation error message, and the index remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the unique-index requirement.
6. After you fix all the nonconforming rows in the target table, issue the SET Database Object Mode statement again to switch the disabled index to the enabled mode. This time the index is enabled, and no integrity violation error message is returned because all rows in the target table now satisfy the new unique-index requirement.

Specifying Modes for Duplicate Indexes

You must observe the following rules when you specify modes for duplicate indexes in CREATE INDEX statements:

- You can set a duplicate index to enabled or disabled mode. Filtering mode is available only for unique indexes.
- If you do not specify the mode of a duplicate index, by default the index is enabled.

IDS

How the Database Server Treats Disabled Indexes

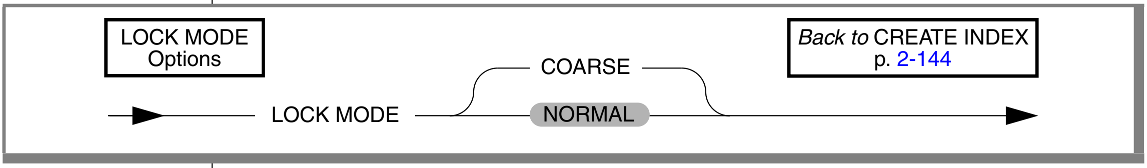
Whether a disabled index is a unique or duplicate index, the database server effectively ignores the index during data-manipulation (DML) operations.

When an index is disabled, the database server stops updating it and stops using it during queries, but the catalog information about the disabled index is retained. You cannot create a new index on a column or set of columns if a disabled index on that column or set of columns already exists. Similarly, you cannot create an active (enabled) unique, foreign-key, or primary-key constraint on a column or set of columns if the indexes on which the active constraint depends are disabled.

XPS

LOCK MODE Options

The LOCK MODE options specify the locking granularity of the index.



In COARSE lock mode, index-level locks are acquired on the index instead of item-level or page-level locks. This mode reduces the number of lock calls on an index. Use the coarse-lock mode when you know the index is not going to change, that is, when read-only operations are performed on the index.

If you specify no lock mode, the default is NORMAL. That is, the database server places item-level or page-level locks on the index as necessary.

XPS

Generalized-Key Indexes

If you are using Extended Parallel Server, you can create generalized-key (GK) indexes. Keys in a conventional index consist of one or more columns of the STATIC table that is being indexed. A GK index stores information about the records in a STATIC table based on the results of a query.

GK indexes provide a form of pre-computed index capability that allows faster query processing, especially in data-warehousing environments. The optimizer can use the GK index to improve performance.

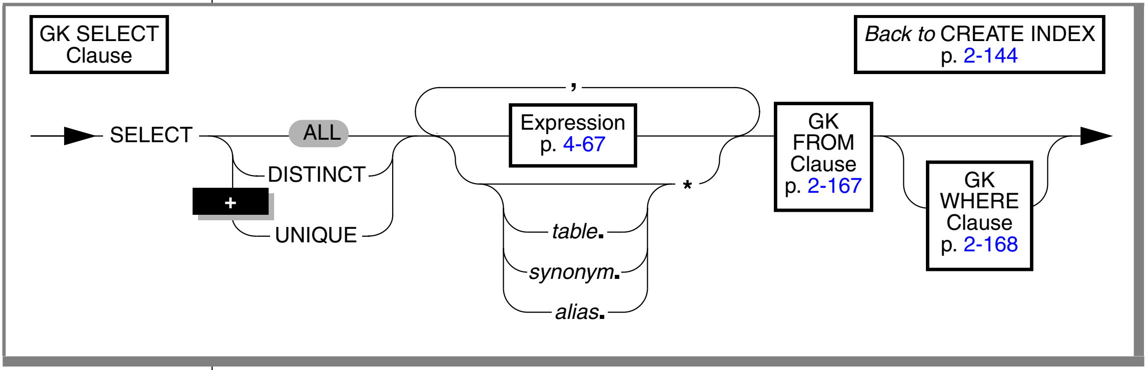
A GK index is *defined on* a table when that table is the one being indexed. A GK index *depends on* a table when the table appears in the FROM clause of the index. Before you create a GK index, keep the following issues in mind:

- All tables used in a GK index must be STATIC tables. If you try to change the type of a table to a non-static type while a GK index depends on that table, the database server returns an error.
- Any table involved in a GK index must be a STATIC type. UPDATE, DELETE, INSERT, and LOAD operations are not valid on such a table until the dependent GK index is dropped and the table type changes.

Key-only index scans are not available with GK indexes.

SELECT Clause for Generalized-Key Index

If you are using Extended Parallel Server, the options of the GK SELECT clause are a subset of the options of "SELECT" on page 2-581. The GK SELECT clause has the following syntax.

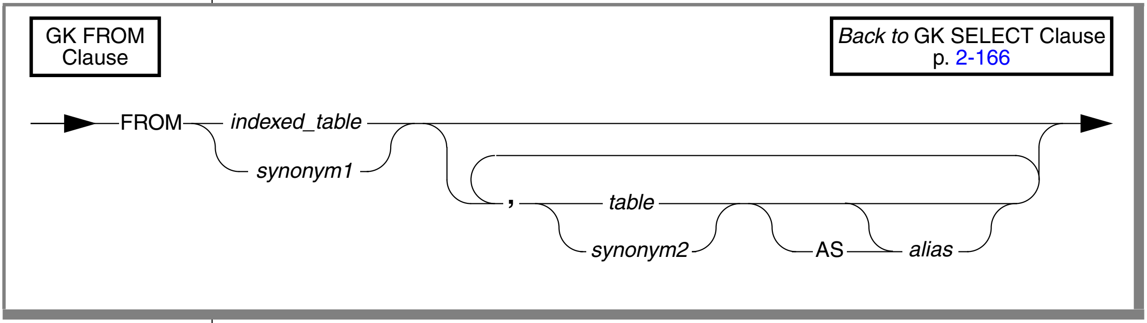


Element	Purpose	Restrictions	Syntax
alias	Temporary name assigned to the table in the FROM clause	You cannot use an alias for the table on which the index is built	Identifier, p. 4-189
synonym, table	Synonym or table from which to retrieve data	The synonym and the table to which it points must exist	Database Object Name, p. 4-46

The following restrictions apply to expressions in the GK SELECT clause:

- It cannot refer to any SPL routine.
- It cannot include the USER, TODAY, CURRENT, DBINFO built-in functions, nor any function that refers to a point in time or interval.

FROM Clause for Generalized-Key Index



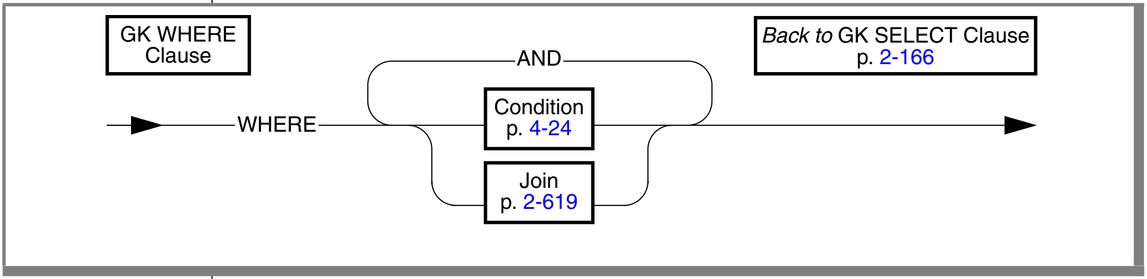
Element	Purpose	Restrictions	Syntax
alias	Temporary name for a table	You cannot use an alias with *indexed_table*.	Identifier, p. 4-189
indexed_table, *synonym1*	Table on which the index is being built	The FROM clause must include the indexed table.	Database Object Name, p. 4-46
synonym2, *table*	Synonym or table from which to retrieve data	The synonym and the table to which it points must exist.	Database Object Name, p. 4-46

All tables that appear in the FROM clause must be local static tables. That is, no views, non-static, or remote tables are allowed.

The tables that are mentioned in the FROM clause must be *transitively joined on key* to the indexed table. Table **A** is transitively joined on key to table **B** if **A** and **B** are joined with equal joins on the unique-key columns of **A**. For example, suppose tables **A**, **B**, and **C** each have **col1** as a primary key. In the following example, **B** is joined on key to **A** and **C** is joined on key to **B**. **C** is transitively joined on key to **A**.

```
CREATE GK INDEX gki
(SELECT A.col1, A.col2 FROM A, B, C
 WHERE A.col1 = B.col1 AND B.col1 = C.col1)
```

WHERE Clause for Generalized-Key Index



The WHERE clause for a GK index has the following limitations:

- It cannot include USER, TODAY, CURRENT, nor DBINFO built-in functions, nor any functions that refer to time or a time interval.
- It cannot refer to any SPL routine.
- It cannot have any subqueries.
- It cannot use any aggregate function.
- It cannot have any IN, LIKE, or MATCH clauses.

Related Information

Related statements: ALTER INDEX, CREATE OPCLASS, CREATE TABLE, DROP INDEX, RENAME INDEX, and SET Database Object Mode

For a discussion of the structure of indexes, see your *Administrator's Reference*.

For a discussion of the different types of indexes and information about performance issues with indexes, see your *Performance Guide*.

GLS

For a discussion of the GLS aspects of the CREATE INDEX statement, see the *IBM Informix GLS User's Guide*. ♦

For information about operator classes, refer to the CREATE OPCLASS statement and *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

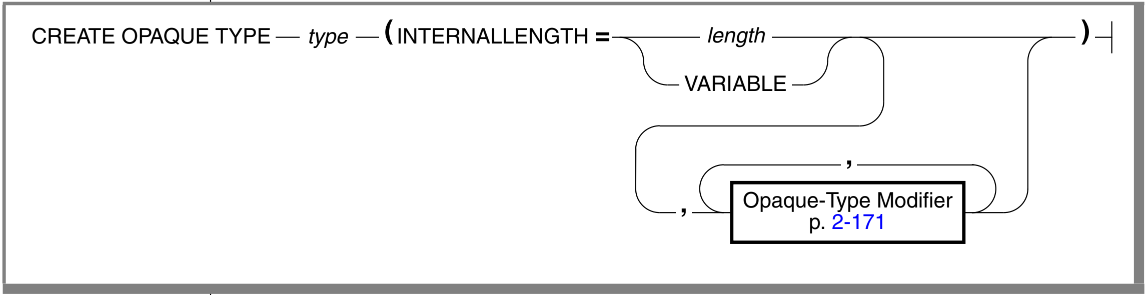
For information about the indexes that DataBlade modules provide, refer to your DataBlade module user's guide.

+
IDS

CREATE OPAQUE TYPE

Use the CREATE OPAQUE TYPE statement to create an opaque data type.

Syntax



Element	Purpose	Restrictions	Syntax
length	Number of bytes needed to store a value of this data type	Positive integer returned when **sizeof( )** directive is applied to the type structure.	Literal Number, p. 4-216
type	Name that you declare here for the new opaque data type	Must be unique among data type names in the database.	Identifier, p. 4-189; Data Type, p. 4-49

Usage

The CREATE OPAQUE TYPE statement registers a new opaque type in the **sysxtdtypes** system catalog table.

To create an opaque type, you must have the Resource privilege on the database. When you create the opaque type, only you, the owner, have the Usage privilege on this type. Use the GRANT or REVOKE statements to grant or revoke the Usage privilege to other database users.

To view the privileges on a data type, check the **sysxtdtypes** system catalog table for the owner name and the **sysxtdtypeauth** system catalog table for additional type privileges that might have been granted.

For details of system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

DB

The DB-Access utility can also display privileges on opaque types. ♦

Declaring a Name for an Opaque Type

The name that you declare for an opaque data type is an SQL identifier. When you create an opaque type, the name must be unique within a database.

ANSI

When you create an opaque type in an ANSI-compliant database, *owner***.***type* combination must be unique within the database.

The owner name is case sensitive. If you do not put quotes around the owner name, the name of the opaque-type owner is stored in uppercase letters. ♦

INTERNALLENGTH Modifier

The INTERNALLENGTH modifier specifies the storage size that is required for the opaque type as fixed length or varying length.

Fixed-Length Opaque Types

A fixed-length opaque type has an internal structure of fixed size. To create a fixed-length opaque type, specify the size of the internal structure, in bytes, for the INTERNALLENGTH modifier. The next example creates a fixed-length opaque type called **fixlen_typ** and allocates 8 bytes for this type.

```
CREATE OPAQUE TYPE fixlen_typ(INTERNALLENGTH=8, CANNOTHASH)
```

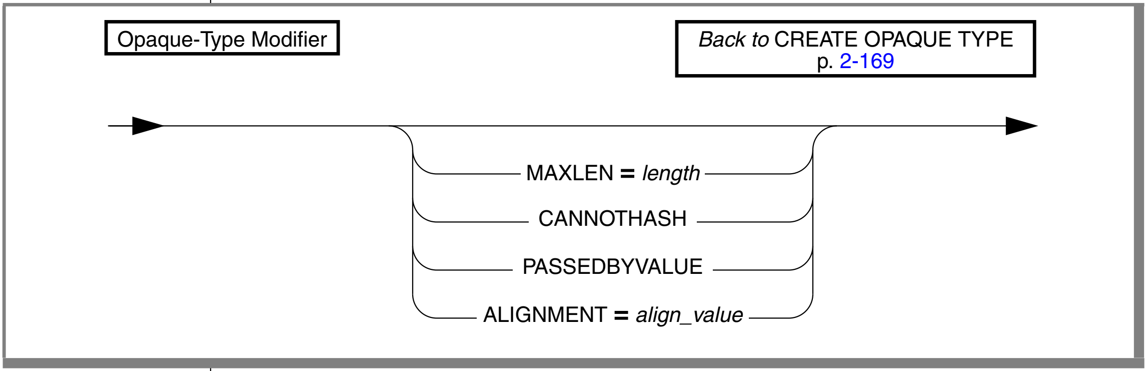
Varying-Length Opaque Types

A varying-length opaque type has an internal structure whose size might vary from one value to another. For example, the internal structure of an opaque type might hold the actual value of a string up to a certain size but beyond this size it might use an LO-pointer to a CLOB to hold the value.

To create a varying-length opaque data type, use the VARIABLE keyword with the INTERNALLENGTH modifier. The following statement creates a variable-length opaque type called **varlen_typ**:

```
CREATE OPAQUE TYPE varlen_typ
(INTERNALLENGTH=VARIABLE, MAXLEN=1024)
```

Opaque-Type Modifier



Element	Purpose	Restrictions	Syntax
align_value	Byte boundary on which to align an opaque type that is passed to a user-defined routine. Default is 4 bytes.	Must be 1, 2, 4, or 8, depending on the C definition of the opaque type and hardware and compiler used to build the object file for the type.	Literal Number, p. 4-216
length	Maximum length to allocate for instances of varying-length opaque types. Default is 2 kilobytes.	Must be a positive integer £ 32 kilobytes. Do not specify for fixed-length data types. Values that exceed this length return errors.	Literal Number, p. 4-216

Modifiers can specify the following optional information for opaque types:

- MAXLEN specifies the maximum length for varying-length types.
- CANNOTHASH specifies that the database server cannot use the built-in hash function on the opaque type.
- ALIGNMENT specifies the byte boundary on which the database server aligns the opaque type.
- PASSEDBYVALUE specifies that an opaque type of 4 bytes or fewer is passed by value.

By default, opaque types are passed to user-defined routines by reference.

Defining an Opaque Type

To define the opaque type to the database server, you must provide the following information in the C or Java language:

- A data structure that serves as the internal storage of the opaque type

  The internal storage details of the type are hidden, or opaque. Once you define a new opaque type, the database server can manipulate it without knowledge of the C or Java structure in which it is stored.

- Support functions that allow the database server to interact with this internal structure

  The support functions tell the database server how to interact with the internal structure of the type. These support functions must be written in the C or Java programming language.

- Additional user-defined functions that other support functions or end users can invoke to operate on the opaque type (optional)

  Possible support functions include operator functions and cast functions. Before you can use these functions in SQL statements, they must be registered with the appropriate CREATE CAST, CREATE PROCEDURE, or CREATE FUNCTION statement.

The following table summarizes the support functions for an opaque type.

Function	Description	Invoked
input( )	Converts the opaque type from its external LVARCHAR representation to its internal representation	When a client application sends a character representation of the opaque type in an INSERT, UPDATE, or LOAD statement
output( )	Converts the opaque type from its internal representation to its external LVARCHAR representation	When the database server sends a character representation of the opaque type as a result of a SELECT or FETCH statement
receive( )	Converts the opaque type from its internal representation on the client computer to its internal representation on the server computer Provides platform-independent results regardless of differences between client and server computer types	When a client application sends an internal representation of the opaque type in an INSERT, UPDATE, or LOAD statement

(1 of 3)

Function	Description	Invoked
send( )	Converts the opaque type from its internal representation on the server computer to its internal representation on the client computer Provides platform-independent results regardless of differences between client and database server computer types	When the database server sends an internal representation of the opaque type as a result of a SELECT or FETCH statement
db_receive( )	Converts the opaque type from its internal representation on the local database to the DBSENDRECV type for transfer to an external database on the local server	When a local database receives a dbsendrecv type from an external database on the local database server
db_send( )	Converts the opaque type from its internal representation on the local database to the DBSENDRECV type for transfer to an external database on the local server	When a local database sends a dbsendrecv type to an external database on the local database server
server_receive( )	Converts the opaque type from its internal representation on the local server computer to the SRVSENDRECV type for transfer to a remote database server Use any name for this function.	When the local database server receives a srvsendrecv type from a remote database server
server_send( )	Converts the opaque type from its internal representation on the local server computer to the SRVSENDRECV type for transfer to a remote database server Use any name for this function.	When the local database server sends a srvsendrecv type to a remote database server
import( )	Performs any tasks needed to convert from the external (character) representation of an opaque type to the internal format for a bulk copy	When DB-Access (LOAD) or the High-Performance Loader (HPL) initiates a bulk copy from a text file to a database
export ( )	Performs any tasks needed to convert from the internal representation of an opaque type to the external (character) format for a bulk copy	When DB-Access (UNLOAD) or the High Performance Loader initiates a bulk copy from a database to a text file
importbinary( )	Performs any tasks needed to convert from the internal representation of an opaque type on the client computer to the internal representation on the server computer for a bulk copy	When DB-Access (LOAD) or the High Performance Loader initiates a bulk copy from a binary file to a database

(2 of 3)

Function	Description	Invoked
exportbinary( )	Performs any tasks needed to convert from the internal representation of an opaque type on the server computer to the internal representation on the client computer for a bulk copy	When DB-Access (UNLOAD) or the High Performance Loader initiates a bulk copy from a database to a binary file
assign()	Performs any processing required before storing the opaque type to disk This support function must be named **assign( )**.	When the database server executes INSERT, UPDATE, or LOAD, before it stores the opaque type to disk
destroy()	Performs any processing necessary before removing a row that contains the opaque type This support function must be named **destroy( ).**	When the database server executes the DELETE or DROP TABLE, before it removes the opaque type from disk
lohandles()	Returns a list of the LO-pointer structures (pointers to smart large objects) in an opaque type	When the database server must search opaque types for references to smart large objects: when **oncheck** runs, or an archive is performed
compare()	Compares two values of the opaque type and returns an integer value to indicate whether the first value is less than, equal to, or greater than the second value	When the database server encounters an ORDER BY, UNIQUE, DISTINCT, or UNION clause in a SELECT statement, or when CREATE INDEX creates a B-tree index

(3 of 3)

After you write the necessary support functions for the opaque type, use the CREATE FUNCTION statement to register these support functions in the same database as the opaque type. Certain support functions convert other data types to or from the new opaque type. After you create and register these support functions, use the CREATE CAST statement to associate each function with a particular cast. The cast must be registered in the same database as the support function.

After you have written the necessary C language or Java language source code to define an opaque data type, you then use the CREATE OPAQUE TYPE statement to register the opaque data type in the database.

Related Information

Related statements: CREATE CAST, CREATE DISTINCT TYPE, CREATE FUNCTION, CREATE ROW TYPE, CREATE TABLE, and DROP TYPE

For a description of an opaque type, see the *IBM Informix Guide to SQL: Reference*.

For information on how to define an opaque type, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For information on how to use the Java language to define an opaque type, see the *J/Foundation Developer's Guide*.

For information about the GLS aspects of the CREATE OPAQUE TYPE statement, refer to the *IBM Informix GLS User's Guide*.

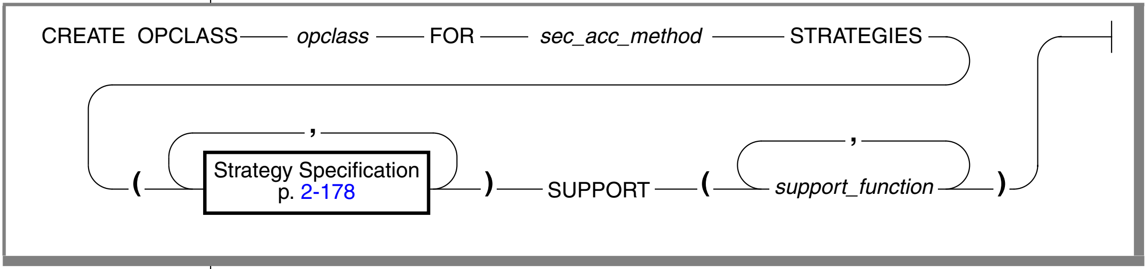
+

IDS

CREATE OPCLASS

Use the CREATE OPCLASS statement to create an *operator class* for a *secondary-access method*.

Syntax



Element	Purpose	Restrictions	Syntax
opclass	Name that you declare here for operator class	Must be unique among operator classes within the database.	Database Object
sec_acc_method	Secondary-access method with which the specified operator class is being associated	Must already exist and must be registered in the **sysams** system catalog table.	Identifier, p. 4-189
support_function	Support function that the secondary-access method requires	Must be listed in the order expected by the access method.	Identifier, p. 4-189

Usage

An *operator class* is the set of operators that Dynamic Server associates with the specified secondary-access method for query optimization and building the index. A secondary-access method (sometimes referred to as an *index access method*) is a set of database server functions that build, access, and manipulate an index structure such as a B-tree, R-tree, or an index structure that a DataBlade module provides.

The database server provides the B-tree and R-tree secondary-access methods. For more information on the **btree** secondary-access method, see "Default Operator Classes" on page 2-180.

Define a new operator class when you want one of the following:

- An index to use a different order for the data than the sequence that the default operator class provides
- A set of operators that is different from any existing operator classes that are associated with a particular secondary-access method

You must have the Resource privilege or be the DBA to create an operator class. The actual name of an operator class is an SQL identifier. When you create an operator class, *opclass name* must be unique within a database.

ANSI

When you create an operator class in an ANSI-compliant database, *owner.opclass_name* must be unique within the database. The owner name is case sensitive. If you do not put quotes around the owner name, the name of the operator-class owner is stored in uppercase letters. ♦

The following CREATE OPCLASS statement creates a new operator class called **abs_btree_ops** for the **btree** secondary-access method:

```
CREATE OPCLASS abs_btree_ops FOR btree
   STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte, abs_gt)
   SUPPORT (abs_cmp)
```

An operator class has two kinds of operator-class functions:

- Strategy functions

  Specify strategy functions of an operator class in the STRATEGY clause of the CREATE OPCLASS statement. In the preceding CREATE OPCLASS code example, the **abs_btree_ops** operator class has five strategy functions.
- Support functions

  Specify support functions of an operator class in the SUPPORT clause. In the preceding CREATE OPCLASS code example, the **abs_btree_ops** operator class has one support function.

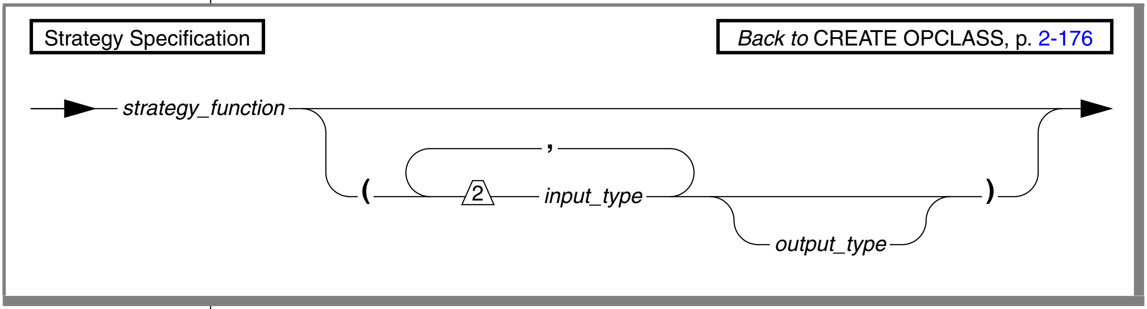
STRATEGIES Clause

Strategy functions are functions that end users can invoke within an SQL statement to operate on a data type. The query optimizer uses the strategy functions to determine if a particular index can be used to process a query.

If an index exists on a column or user-defined function in a query, and the qualifying operator in the query matches one of the strategy functions in the Strategy Specification list, the optimizer considers using the index for the query. For more information on query plans, see your *Performance Guide*.

When you create a new operator class, you specify the strategy functions for the secondary-access method in the STRATEGIES clause. The Strategy Specification lists the name of each strategy function. List these functions in the order that the secondary-access method expects. For the specific order of strategy operators for the default operator classes for a B-tree index and an R-tree index, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Strategy Specification



Element	Purpose	Restrictions	Syntax
input_type	Data type of the input argument for the strategy function for which you want to use a specific secondary-access method	A strategy function takes two input arguments and one optional output argument.	Data Type, p. 4-49
output_type	Data type of the optional output argument for the strategy function	This is an optional output argument for side-effect indexes.	Data Type, p. 4-49
strategy_function	Name of a strategy function to associate with the specified operator class	Must be listed in the order that the specified secondary-access method expects.	Database Object Name, p. 4-46

The *strategy_function* is an external function. The CREATE OPCLASS statement does not verify that a user-defined function of the name you specify exists. However, for the secondary-access method to use the strategy function, the external function must be:

- Compiled in a shared library
- Registered in the database with the CREATE FUNCTION statement

Optionally, you can specify the signature of a strategy function in addition to its name. A strategy function requires two input parameters and an optional output parameter. To specify the function signature, specify:

- An *input data type* for each of the two input parameters of the strategy function, in the order that the strategy function uses them
- Optionally, one *output data type* for an output parameter of the strategy function

You can specify UDTs as well as built-in data types. If you do not specify the function signature, the database server assumes that each strategy function takes two arguments of the same data type and returns a BOOLEAN value.

Indexes on Side-Effect Data

Side-effect data are additional values that a strategy function returns after a query that contains the strategy function. For example, an image DataBlade module might use a *fuzzy* index to search image data. The index ranks the images according to how closely they match the search criteria. The database server returns the rank value as side-effect data with the qualifying images.

SUPPORT Clause

Support functions are functions that the secondary-access method uses internally to build and search the index. Specify these functions for the secondary-access method in the SUPPORT clause of CREATE OPCLASS.

You must list the names of the support functions in the order that the secondary-access method expects. For the specific order of support operators for the default operator classes for a B-tree index and an R-tree index, refer to "Default Operator Classes" on page 2-180.

The support function is an external function. CREATE OPCLASS does not verify that a named support function exists. For the secondary-access method to use a support function, however, the support function must be:

- Compiled in a shared library
- Registered in the database with the CREATE FUNCTION statement

Default Operator Classes

Each secondary-access method has a default operator class that is associated with it. By default, the CREATE INDEX statement associates the default operator class with an index. For example, the following CREATE INDEX statement creates a B-tree index on the **zipcode** column and automatically associates the default B-tree operator class with this column:

```
CREATE INDEX zip_ix ON customer(zipcode)
```

For each of the secondary-access methods that Dynamic Server provides, it provides a *default operator class*, as follows:

- The default B-tree operator class is a built-in operator class.

  The database server implements the operator-class functions for this operator class and registers it as **btree_ops** in the system catalog tables of a database.
- The default R-tree operator class is a registered operator class.

  The database server registers this operator class as **rtree_ops** in the system catalog tables. The database server does *not* implement the operator-class functions for the default R-tree operator class.

Important: *To use an R-tree index, you must install a spatial DataBlade module such as the Geodetic DataBlade module or any other third-party DataBlade module that implements the R-tree index. These implement the R-tree operator-class functions.*

DataBlade modules can provide other types of secondary-access methods. If a DataBlade module provides a secondary-access method, it might also provide a default operator class. For more information, refer to your DataBlade module user's guide.

Related Information

Related statements: CREATE FUNCTION, CREATE INDEX, and DROP OPCLASS

For information on support functions and how to create and extend an operator class, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For more about R-tree indexes, see the *IBM Informix R-Tree Index User's Guide*.

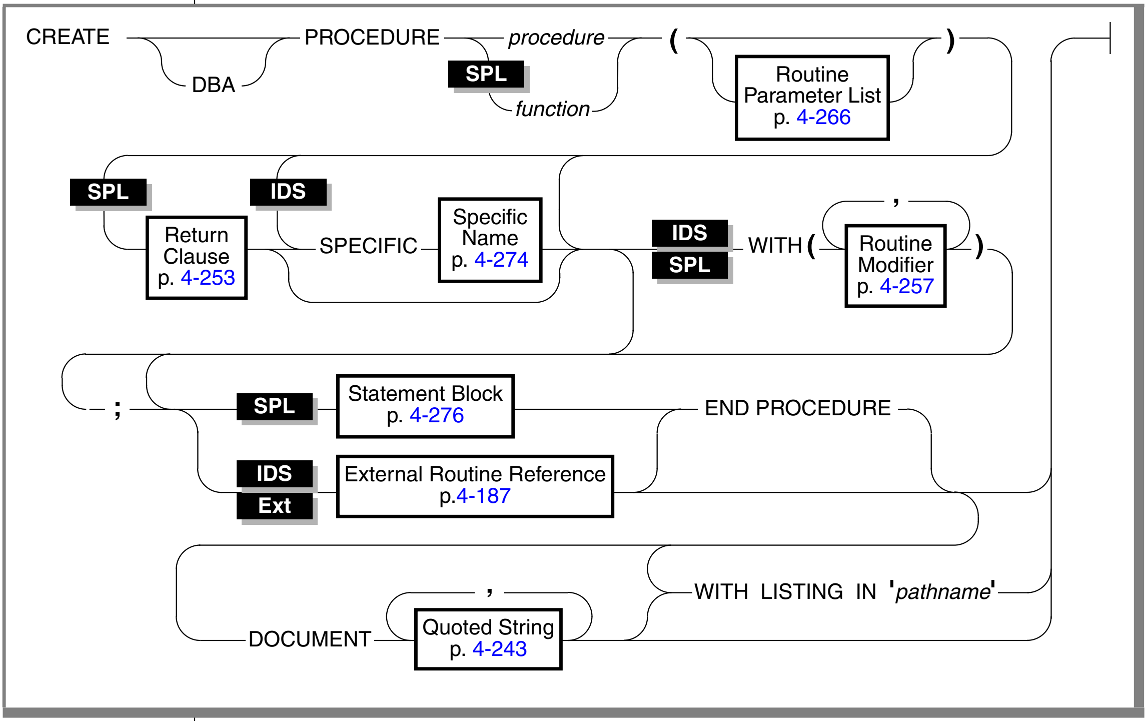
For information about the GLS aspects of the CREATE OPCLASS statement, refer to the *IBM Informix GLS User's Guide*.

+

CREATE PROCEDURE

Use the CREATE PROCEDURE statement to create a user-defined procedure. (To create a procedure from text of source code that is in a separate file, use the CREATE PROCEDURE FROM statement.)

Syntax



Element	Purpose	Restrictions	Syntax
function, procedure	Name declared here for a new SPL function or procedure	(XPS) The name must be unique among all SPL routines in the database. (IDS) See "Naming a Procedure in Dynamic Server" on page 2-185.	Database Object Name, p. 4-46
pathname	File to store compile-time warnings	Must exist on the computer where the database resides.	Must conform to naming rules of operating system.

Usage

The entire length of a CREATE PROCEDURE statement must be less than 64 kilobytes. This length is the literal length of the CREATE PROCEDURE statement, including blank space, tabs, and other whitespace characters.

E/C

In ESQL/C, you can use CREATE PROCEDURE only as text within a PREPARE statement. If you want to create a procedure for which the text is known at compile time, you must use a CREATE PROCEDURE FROM statement. ♦

IDS

Routines use the collating order that was in effect when they were created. See SET COLLATION for information about using non-default collation ♦

Using CREATE PROCEDURE Versus CREATE FUNCTION

XPS

In Extended Parallel Server, besides using this statement to create SPL procedures, yoau must use CREATE PROCEDURE to write and register an SPL routine that returns one or more values (that is, an SPL function). Extended Parallel Server does not support the CREATE FUNCTION statement. ♦

IDS

In Dynamic Server, although you can use CREATE PROCEDURE to write and register an SPL routine that returns one or more values (that is, an SPL function), it is recommended that you use CREATE FUNCTION instead. To register an external function, you must use CREATE FUNCTION.

Use the CREATE PROCEDURE statement to write and register an SPL procedure or to register an external procedure. ♦

For information on how terms such as user-defined procedures and user-defined functions are used in this manual, see "Relationship Between Routines, Functions, and Procedures" on page 2-183.

Tip: *If you are trying to create a procedure from text that is in a separate file, use the CREATE PROCEDURE FROM statement.*

Relationship Between Routines, Functions, and Procedures

A *procedure* is a routine that can accept arguments but does not return any values. A *function* is a routine that can accept arguments and returns one or more values. *User-defined routine (UDR)* is a generic term that includes both user-defined procedures and user-defined functions. For information about named and unnamed returned values, see "Return Clause" on page 4-253.

You can write a UDR in SPL (*SPL routine*) or in an external language (*external routine)* that the database server supports. Where the term UDR appears in the manual, it can refer to both SPL routines and external routines.

The term *user-defined procedure* refers to SPL procedures and external procedures. *User-defined function* refers to SPL functions and external functions.

SPL

In earlier IBM Informix products, the term *stored procedure* was used for both SPL procedures and SPL functions. In this manual, the term *SPL routine* replaces the term stored procedure. When it is necessary to distinguish between an SPL function and an SPL procedure, the manual does so. ♦

IDS

The term *external routine* applies to an external procedure or an external function. When it is necessary to distinguish between an external function and an external procedure, the manual does so. ♦

XPS

Extended Parallel Server does not support external routines, but the term user-defined routine (UDR) encompasses both *SPL routines* and *external routines*. Wherever the term UDR appears, it is applicable to SPL routines. ♦

Privileges Necessary for Using CREATE PROCEDURE

You must have the Resource privilege on a database to create a user-defined procedure within that database.

Ext

Before you can create an external procedure, you must also have the Usage privilege on the language in which you will write the procedure. For more information, see "GRANT" on page 2-459. ♦

SPL

By default, the Usage privilege on SPL is granted to PUBLIC. You must also have at least the Resource privilege on a database to create an SPL procedure within that database. ♦

DBA Keyword and Privileges on the Created Procedure

If you create a UDR with the DBA keyword, it is known as a *DBA-privileged UDR*. You need the DBA privilege to create or execute a DBA-privileged UDR. If you omit the DBA keyword, the UDR is known as an *owner-privileged* UDR.

ANSI

If you create an owner-privileged UDR in an ANSI-compliant database, anyone can execute the UDR. ♦

If you create an owner-privileged UDR in a database that is not ANSI compliant, the **NODEFDAC** environment variable prevents privileges on that UDR from being granted to PUBLIC. If this environment variable is set, the owner of a UDR must grant the Execute privilege for that UDR to other users.

XPS

Naming a Procedure in Extended Parallel Server

In Extended Parallel Server, the name for any SPL routine that you create must be unique among the names of all SPL routines in the database.

IDS

Naming a Procedure in Dynamic Server

Because Dynamic Server offers *routine overloading*, you can define more than one user-defined routine (UDR) with the same name, but different parameter lists. You might want to overload UDRs in the following situations:

- You create a UDR with the same name as a built-in routine (such as **equal( )**) to process a new user-defined data type.
- You create *type hierarchies* in which subtypes inherit data representation and UDRs from supertypes.
- You create *distinct types*, which are data types that have the same internal storage representation as an existing data type, but have different names and cannot be compared to the source type without casting. Distinct types inherit UDRs from their source types.

For a brief description of the routine signature that uniquely identifies each UDR, see "Routine Overloading and Naming UDRs with a Routine Signature" on page 4-48.

Using the SPECIFIC Clause to Specify a Specific Name

You can specify a specific name for a user-defined procedure. A specific name is a name that is unique in the database. A specific name is useful when you are overloading a procedure.

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of a UDR. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the UDR.

Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all the UDRs stored in the database. A UDR or application program can query the system catalog tables to fetch the DOCUMENT clause and display it for a user.

For example, to find the description of the SPL procedure **raise_prices**, shown in "SPL Procedures" on page 2-187, enter a query such as this example:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid
      --join between the two catalog tables
   AND p.procname = 'raise_prices'
      -- look for procedure named raise_prices
   AND b.datakey  = 'D';-- want user document
```

The preceding query returns the following text:

```
USAGE: EXECUTE PROCEDURE raise_prices( xxx )
xxx = percentage from 1 - 100
```

Ext

You can use a DOCUMENT clause at the end of the CREATE PROCEDURE statement, whether or not you use the END PROCEDURE keywords. ♦

Using the WITH LISTING IN Option

The WITH LISTING IN clause specifies a filename where compile time warnings are sent. After you compile a UDR, this file holds one or more warning messages. This listing file is created on the computer where the database resides.

If you do not use the WITH LISTING IN clause, the compiler does not generate a list of warnings.

UNIX

If you specify a filename but not a directory, this listing file is created in your home directory on the computer where the database resides. If you do not have a home directory on this computer, the file is created in the root directory (the directory named "/"). ♦

Windows

If you specify a filename but not a directory, this listing file is created in your current working directory if the database is on the local computer. Otherwise, the default directory is **%INFORMIXDIR%\bin**. ♦

SPL

SPL Procedures

SPL procedures are UDRs written in Stored Procedure Language (SPL) that do not return a value. To write and register an SPL routine, use the CREATE PROCEDURE statement. Embed appropriate SQL and SPL statements between the CREATE PROCEDURE and END PROCEDURE keywords. You can also follow the UDR definition with the DOCUMENT and WITH FILE IN options.

SPL routines are parsed, optimized (as far as possible), and stored in the system catalog tables in executable format. The body of an SPL routine is stored in the **sysprocbody** system catalog table. Other information about the routine is stored in other system catalog tables, including **sysprocedures**, **sysprocplan**, and **sysprocauth**.

If the Statement Block portion of the CREATE PROCEDURE statement is empty, no operation takes place when you call the procedure. You might use such a procedure in the development stage when you want to establish the existence of a procedure but have not yet coded it.

If you specify an optional clause after the parameter list, you must place a semicolon after the clause that immediately precedes the Statement Block.

The following example creates an SPL procedure:

```
CREATE PROCEDURE raise_prices ( per_cent INT )
   UPDATE stock SET unit_price =
      unit_price + (unit_price * (per_cent/100) );
END PROCEDURE
   DOCUMENT "USAGE: EXECUTE PROCEDURE raise_prices( xxx )",
   "xxx = percentage from 1 - 100 "
   WITH LISTING IN '/tmp/warn_file'
```

IDS

Ext

External Procedures

External procedures are procedures you write in an external language that the database server supports.

To create a C user-defined procedure

1. Write a C function that does not return a value.
2. Compile the C function and store the compiled code in a shared library (the shared-object file for C).
3. Register the C function in the database server with the CREATE PROCEDURE statement.

To create a user-defined procedure written in the Java language

1. Write a Java static method, which can use the JDBC functions to interact with the database server.
2. Compile the Java source and create a jar file (the shared-object file).
3. Execute the **install_jar( )** procedure with the EXECUTE PROCEDURE statement to install the jar file in the current database.
4. If the UDR uses user-defined types, create a mapping between SQL data types and Java classes, using the **setUDTExtName( )** procedure that is explained in "EXECUTE PROCEDURE" on page 2-414.
5. Register the UDR with the CREATE PROCEDURE statement. (If an external routine returns a value, you must register it with the CREATE FUNCTION statement, rather than CREATE PROCEDURE.)

Rather than storing the body of an external routine directly in the database, the database server stores only the pathname of the shared-object file that contains the compiled version of the routine. The database server executes an external routine by invoking the external object code.

Registering a User-Defined Procedure

C

This example registers a C user-defined procedure named **check_owner( )** that takes one argument of the type LVARCHAR. The external routine reference specifies the path to the C shared library where the procedure object code is stored. This library contains a C function **unix_owner( )**, which is invoked during execution of the **check_owner( )** procedure.

```
CREATE PROCEDURE check_owner ( owner lvarchar )
   EXTERNAL NAME "/usr/lib/ext_lib/genlib.so(unix_owner)"
   LANGUAGE C
END PROCEDURE
```

♦

Java

This example registers a user-defined procedure named **showusers( )** that is written in the Java language:

```
CREATE PROCEDURE showusers()
   WITH (CLASS = "jvp") EXTERNAL NAME 'admin_jar:admin.showusers'
   LANGUAGE JAVA
```

The EXTERNAL NAME clause specifies that the Java implementation of the **showusers( )** procedure is a method called **showusers( )**, which resides in the **admin** Java class that resides in the **admin_jar** jar file. ♦

Ownership of Created Database Objects

The user who creates an owner-privileged UDR owns any database objects that the UDR creates when the UDR is executed, unless another owner is specified for the created database object. In other words, the UDR owner, not the user who executes the UDR, is the owner of any database objects created by the UDR unless another owner is specified in the statement that creates the database object.

In the case of a DBA-privileged UDR, however, the user who executes the UDR, not the UDR owner, owns any database objects that the UDR created unless another owner is specified for the database object within the UDR.

For examples, see "Ownership of Created Database Objects" on page 2-140 in the description of the CREATE FUNCTION statement.

XPS

Using sysbdopen( ) and sysdbclose( ) Stored Procedures

To set the initial environment for one or more sessions, create and install the **sysdbopen( )** SPL procedure. The main function of these procedures is to initialize a session's properties without requiring the properties to be explicitly defined within the session. These procedures are executed whenever users connect to a database where the procedures are installed. Such procedures are useful if users access databases through client applications that cannot modify application code or set environment options or environment variables.

You can also create the **sysdbclose( )** SPL procedure which is executed when a user disconnects from the database.

You can include valid SQL or SPL language statements that are appropriate when a database is opened or closed. See the following sections for restrictions on SQL and SPL statements within SPL routines:

- "Subset of SPL Statements Valid in the Statement Block" on page 4-276
- "SQL Statements Not Valid in an SPL Statement Block" on page 4-277
- "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279

Important: *The* ***sysdbopen()*** *and* ***sysdbclose()*** *procedures are exceptions to the scope rule for stored procedures. In ordinary user-created stored procedures, the scope of variables and statements is local. SET PDQPRIORITY and SET ENVIRONMENT statement settings do not persist when these SPL procedures exit. However, in* ***sysdbopen()*** *and* ***sysdbclose()*** *procedures, statements that set the session environment remain in effect until another statement resets the options.*

For example, you might create the following procedure, which sets the isolation level to Dirty Read and turns on the **IMPLICIT_PDQ** environment option, to be executed when any user connect to the database:

```
create procedure public.sysdbopen()
   set role engineer;
end procedure
```

Procedures do not accept arguments or return values. The **sysdbopen()** and **sysdbclose()** procedures must be executed from the connection coserver and must be installed in each database where you want to execute them. You can create the following four SPL procedures.

Procedure Name	Description
***user*.sysdbopen()**	This procedure is executed when the specified ***user*** opens the database as the current database.
public.sysdbopen()	If no ***user*.sysdbopen()** procedure applies, this procedure is executed when any user opens the database as the current database. To avoid duplicating SPL code, you can call this procedure from a user-specific procedure.
***user*.sysdbclose()**	This procedure is executed when the specified ***user*** closes the database, disconnects from the database server, or the user session ends. If the **sysdbclose()** procedure did not exist when a session opened the database, however, it is not executed when the session closes the database.
public.sysdbclose()	If no ***user*.sysdbopen()** procedure applies, this procedure is executed when the specified ***user*** closes the database, disconnects from the database server, or the user session ends. If the **sysdbclose()** procedure did not exist when a session opened the database, however, it is not executed when the session closes the database.

See also the section "Transactions in SPL Routines" on page 4-280.

Make sure that you set permissions appropriately to allow intended users to execute the SPL procedure statements. For example, if the SPL procedure executes a command that writes output to a local directory, permissions must be set to allow users to write to this directory. If you want the procedure to continue if permission failures occur, include an ON EXCEPTION error handler for this condition.

See also the section "Support for Roles and User Identity" on page 4-280.

Warning: *If a **sysdbopen()** procedure fails, the database cannot be opened. If a **sysdbclose()** procedure fails, the failure is ignored. While you are writing and debugging a **sysdbopen()** procedure, set the **DEBUG** environment variable to NODBPROC before you connect to the database. When **DEBUG** is set to NODBPROC the procedure is not executed, and failures cannot prevent the database from opening. Failures from these procedures can be generated by the system or simulated by the procedures with the SPL statement RAISE EXCEPTION. For more information, refer to "RAISE EXCEPTION" on page 3-43.*

Only a user with DBA privileges can install these procedures. For security reasons, non-DBAs cannot prevent execution of these procedures. For some applications, however, such as *ad hoc* query applications, users can execute commands and SQL statements that subsequently change the environment. For general information about how to write and install SPL procedures, refer to the chapter about SPL routines in *IBM Informix Guide to SQL: Tutorial*.

Related Information

Related statements: ALTER FUNCTION, ALTER PROCEDURE, ALTER ROUTINE, CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE FROM, DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, EXECUTE FUNCTION, EXECUTE PROCEDURE, GRANT, PREPARE, REVOKE, and UPDATE STATISTICS

For a discussion of how to create and use SPL routines, see the *IBM Informix Guide to SQL: Tutorial*. For a discussion of external routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

For information about how to create C UDRs, see the *IBM Informix DataBlade API Programmer's Guide*. For more information on the **NODEFDAC** environment variable and the related system catalog tables (**sysprocedures**, **sysprocplan**, **sysprocbody** and **sysprocauth**), see the *IBM Informix Guide to SQL: Reference*.

+

E/C

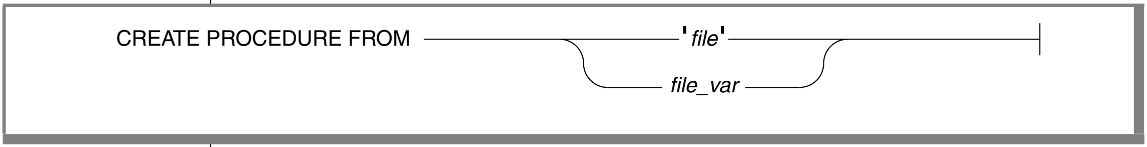
CREATE PROCEDURE FROM

Use the CREATE PROCEDURE FROM statement to access a user-defined procedure. The actual text of the CREATE PROCEDURE statement resides in a separate file. Use this statement with ESQL/C.

XPS

In Extended Parallel Server, use this statement to access any SPL routine. Extended Parallel Server does not support the CREATE FUNCTION FROM statement. ♦

Syntax



Element	Purpose	Restrictions	Syntax
file	Pathname and filename of file that contains full text of a CREATE PROCEDURE statement. Default pathname is the current directory.	Must exist, and can contain only one CREATE PROCEDURE statement. Se also "Default Directory That Holds the File" on page 2-193.	Must conform to the rules of the operating system.
file_var	Name of a program variable that contains *file* specification	Character data type; contents have same restrictions as *file*.	Language specific

Usage

You cannot create a user-defined procedure directly in an ESQL/C program. That is, the program cannot contain the CREATE PROCEDURE statement.

To use a user-defined procedure in an ESQL/C program

1. Create a source file with the CREATE PROCEDURE statement.
2. Use the CREATE PROCEDURE FROM statement to send the contents of this source file to the database server for execution.

   The file can contain only one CREATE PROCEDURE statement.

For example, suppose that the following CREATE PROCEDURE statement is in a separate file, called **raise_pr.sql**:

```
CREATE PROCEDURE raise_prices( per_cent int )
   UPDATE stock -- increase by percentage;
   SET unit_price = unit_price +
      ( unit_price * (per_cent / 100) );
END PROCEDURE;
```

In the ESQL/C program, you can access the **raise_prices()** SPL procedure with the following CREATE PROCEDURE FROM statement:

```
EXEC SQL create procedure from 'raise_pr.sql';
```

IDS

If you are not sure whether the UDR in the file is a user-defined function or a user-defined procedure, use the CREATE ROUTINE FROM statement. ♦

IDS

Procedures use the collating order that was in effect when they were created. See **SET COLLATION** for information about using non-default collation ♦

Default Directory That Holds the File

The filename (and any pathname) that you specify is relative.

UNIX

On UNIX, if you specify a simple filename instead of a full pathname in the *file* parameter, the client application looks for the file in your home directory on the computer where the database resides. If you do not have a home directory on this computer, the default directory is the root directory. ♦

Windows

On Windows, if you specify a filename but not a directory in the *file* parameter, the client application looks for the file in your current working directory if the database is on the local computer. Otherwise, the default directory is **%INFORMIXDIR%\bin**. ♦

Important: *The ESQL/C preprocessor does not process the contents of the file that you specify. It just sends the contents to the database server for execution. Therefore, there is no syntactic check that the file that you specify in CREATE PROCEDURE FROM actually contains a CREATE PROCEDURE statement. To improve readability of the code, however, it is recommended that you match these two statements.*

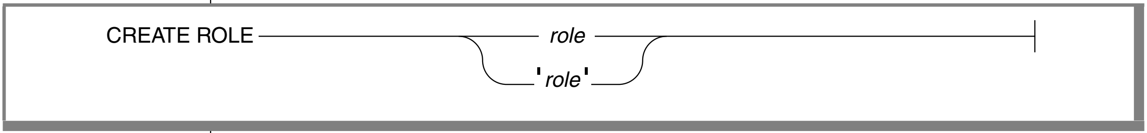
Related Information

Related statements: CREATE PROCEDURE, CREATE FUNCTION FROM, and CREATE ROUTINE FROM

CREATE ROLE

Use the CREATE ROLE statement to create a new role.

Syntax Usage



Element	Purpose	Restrictions	Syntax
role	Name declared here for a role that the DBA created	Must be unique among *role* names in the database. The maximum number of bytes in *role* is 32.	Identifier, p. 4-189

The database administrator (DBA) can use the CREATE ROLE statement to create a new role. A role can be considered as a classification, with privileges on database objects granted to the role. The DBA can assign the privileges of a related work task, such as **engineer**, to a role and then grant that role to users, instead of granting the same set of privileges to every user.

The *role* name is an *authorization identifier*. It cannot be a user name that is known to the database server or to the operating system of the database server. The *role* name cannot already be listed in the **username** column of the **sysusers** system catalog table, nor in the **grantor** or **grantee** columns of the **systabauth**, **syscolauth**, **sysprocauth**, and **sysroleauth** system catalog tables.

IDS

Also, the role name cannot already be listed in the **grantor** or **grantee** columns of the **sysfragauth** system catalog table. ♦

After a role is created, the DBA can use the GRANT statement to grant the role to users or to other roles. When a role is granted to a user, the user must use the SET ROLE statement to enable the role. Only then can the user use the privileges of the role.

The CREATE ROLE statement, when used with the GRANT and SET ROLE statements, allows a DBA to create one set of privileges for a role and then grant the role to many users, instead of granting the same set of privileges to many users.

A role exists until either the DBA or a user to whom the role was granted with the WITH GRANT OPTION uses the DROP ROLE statement to drop the role.

To create the role **engineer**, enter the following statement:

```
CREATE ROLE engineer
```

Related Information

Related statements: DROP ROLE, GRANT, REVOKE, and SET ROLE

For a discussion on how to use roles, see the *IBM Informix Database Design and Implementation Guide*.

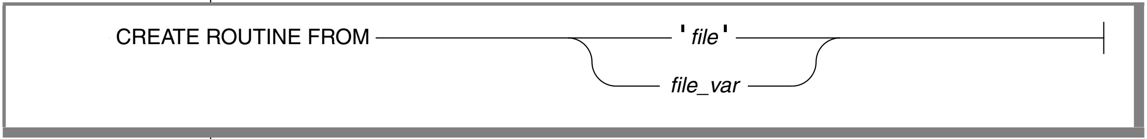
+

IDS

CREATE ROUTINE FROM

Use the CREATE ROUTINE FROM statement to access a user-defined routine (UDR). The actual text of the CREATE FUNCTION or CREATE PROCEDURE statement resides in a separate file.

Syntax



Element	Purpose	Restrictions	Syntax
file	Pathname and filename of file that contains the text of a CREATE PROCEDURE or CREATE FUNCTION statement Default path is current directory.	Must exist and can contain only one CREATE FUNCTION or CREATE PROCEDURE statement.	Operating-system dependent
file_var	Name of a program variable that contains *file* specification	Must be a character data type; contents must satisfy *file* restrictions.	Language specific

Usage

An IBM Informix ESQL/C program cannot directly define a UDR. That is, it cannot contain the CREATE FUNCTION or CREATE PROCEDURE statement.

To use a UDR in an ESQL/C program

1. Create a source file with the CREATE FUNCTION or CREATE PROCEDURE statement.
2. Use the CREATE ROUTINE FROM statement to send the contents of this source file to the database server for execution.

   The file that you specify can contain only one CREATE FUNCTION or CREATE PROCEDURE statement.

The filename that you provide is relative. If you provide no pathname, the client application looks for the file in the current directory.

If you do not know at compile time whether the UDR in the file is a function or a procedure, use the CREATE ROUTINE FROM statement in the ESQL/C program. If you do know if the UDR is a function or a procedure, it is recommended that you use the matching statement to access the source file:

- To access user-defined functions, use CREATE FUNCTION FROM.
- To access user-defined procedures, use CREATE PROCEDURE FROM.

Use of the matching statements improves the readability of the code.

IDS

Routines use the collating order that was in effect when they were created. See **SET COLLATION** for information about using non-default collation ♦

Related Information

Related statements: CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE, and CREATE PROCEDURE FROM

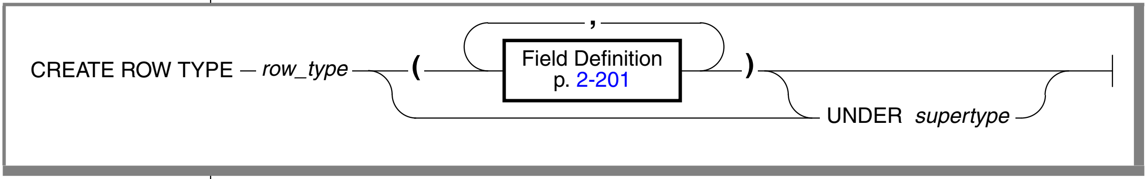
+

IDS

CREATE ROW TYPE

Use the CREATE ROW TYPE statement to create a named row type.

Syntax



Element	Purpose	Restrictions	Syntax
row_type	Name that you declare here for a new named row data type	See "Procedure for Creating a Subtype" on page 2-200.	Identifier, p. 4-189
supertype	Name of the supertype within an inheritance hierarchy	Must already exist as a named row type.	Data type, p. 4-49

Usage

The CREATE ROW TYPE statement creates a named ROW data type. You can assign a named ROW type to a table or view to create a *typed table* or *typed view*. You can also define a column as a named ROW type. Although you can assign a ROW type to a table to define the schema of the table, ROW types are not the same as table rows. Table rows consist of one or more columns; ROW types consist of one or more fields, defined using the Field Definition syntax.

A named ROW type is valid in most contexts where you can specify a data type. Named ROW types are strongly typed. No two named ROW types are equivalent, even if they are structurally equivalent.

ROW types without names are called *unnamed ROW types*. Any two unnamed ROW types are considered equivalent if they are structurally equivalent. For more information, see "Row Data Types" on page 4-62.

Privileges on named ROW type columns are the same as privileges on any column. For more information, see "Table-Level Privileges" on page 2-463. (To see what privileges you have on a column, check the **syscolauth** system catalog table, which is described in the *IBM Informix Guide to SQL: Reference*.)

Privileges on Named Row Data Types

This table indicates which privileges you must have to create a ROW type.

Task	Privileges Required
Create a named ROW type	Resource privilege on the database
Create a named ROW type as a subtype under a supertype	Under privilege on the supertype, as well as the Resource privilege

For information about Resource and Under privileges, and the ALL keyword in the context of privileges, see the GRANT statement.

To find out what privileges exist on a ROW type, check the **sysxtdtypes** system catalog table for the *owner* name and the **sysxtdtypeauth** system catalog table for privileges that might have been granted.

Privileges on a typed table (a table that is assigned a named ROW type) are the same as privileges on any table. For more information, see "Table-Level Privileges" on page 2-463.

To find out what privileges you have on a given table, check the **systabauth** system catalog table. For more information on system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

Inheritance and Named ROW Types

A named ROW type can belong to an inheritance hierarchy, as either a subtype or a supertype. Use the UNDER clause in the CREATE ROW TYPE statement to create a named ROW type as a subtype.

The supertype must also be a named ROW type. If you create a named ROW type under an existing supertype, then the new type name *row_type* becomes the name of the subtype.

When you create a named ROW type as a subtype, the subtype inherits all fields of the supertype. In addition, you can add new fields to the subtype that you create. The new fields are specific to the subtype alone.

You cannot substitute a ROW type in an inheritance hierarchy for its supertype or for its subtype.

For example, consider a type hierarchy in which **person_t** is the supertype and **employee_t** is the subtype. If a column is of type **person_t**, the column can only contain **person_t** data. It cannot contain **employee_t** data. Likewise, if a column is of type **employee_t**, the column can only contain **employee_t** data. It cannot contain **person_t** data.

Creating a Subtype

In most cases, you add new fields when you create a named ROW type as a subtype of another named ROW type (its supertype). To create the fields of a named ROW type, use the field definition clause, as described in "Field Definition" on page 2-201. When you create a subtype, you must use the UNDER keyword to associate the supertype with the named ROW type that you want to create. The following statement creates the **employee_t** type under the **person_t** type:

```
CREATE ROW TYPE employee_t (salary NUMERIC(10,2),
   bonus NUMERIC(10,2)) UNDER person_t;
```

The **employee_t** type inherits all the fields of **person_t** and has two additional fields: **salary** and **bonus**; but the **person_t** type is not altered.

Type Hierarchies

When you create a subtype, you create a *type hierarchy*. In a type hierarchy, each subtype that you create inherits its properties from a single supertype. If you create a named ROW type **customer_t** under **person_t**, **customer_t** inherits all the fields of **person_t**. If you create another named ROW type, **salesrep_t** under **customer_t**, **salesrep_t** inherits all the fields of **customer_t**.

Thus, **salesrep_t** inherits all the fields that **customer_t** inherited from **person_t** as well as all the fields defined specifically for **customer_t**. For a discussion of type inheritance, refer to the *IBM Informix Guide to SQL: Tutorial*.

Procedure for Creating a Subtype

Before you create a named ROW type as a subtype in an inheritance hierarchy, check the following information:

- Verify that you are authorized to create new data types. You must have the Resource privilege on the database. You can find this information in the **sysusers** system catalog table.

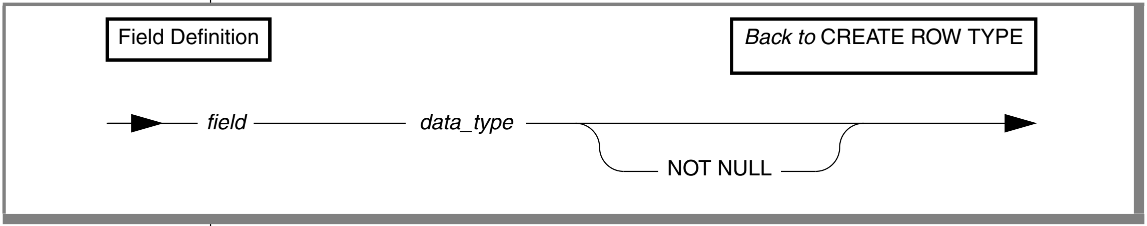
- Verify that the supertype exists. You can find this information in the **sysxtdtypes** system catalog table.
- Verify that you are authorized to create subtypes to that supertype. You must have the Under privilege on the supertype. You can find this information in the **sysusers** system catalog table.
- Verify that the name that you assign to the named ROW type is unique within the database. In an ANSI-compliant database, the *owner.type* combination must be unique within the database. In a database that is not ANSI-compliant, the name must be unique among data type names in the database. To verify whether the name you want to assign to a new data type is unique within the schema, check the **sysxtdtypes** system catalog table. The name must not be the name of an existing data type.
- If you are defining fields for the ROW type, check that no duplicate field names exist in both new and inherited fields.

Important: *When you create a subtype, do not redefine fields that it inherited for its supertype. If you attempt to redefine these fields, the database server returns an error.*

You cannot apply constraints to named ROW types, but you can specify constraints when you create or alter a table that uses named ROW types.

Field Definition

Use the field definition portion of CREATE ROW TYPE to define a new field in a named ROW type.



Element	Purpose	Restrictions	Syntax
data_type	Data type of the field	See "Restrictions on Serial and Simple-Large-Object Data Types" on page 2-202.	Identifier, p. 4-189
field	Name of a field in *data_type*	Must be unique among field names of this row type and of its supertype.	Identifier, p. 4-189

The NOT NULL constraint on named ROW type field applies to corresponding columns when the named ROW type is used to create a typed table.

Restrictions on Serial and Simple-Large-Object Data Types

Serial and simple-large-object data types cannot be nested within a table. Therefore, if a ROW type contains a BYTE, TEXT, SERIAL, or SERIAL8 field, you cannot use the ROW type to define a column in a table that is not based on a ROW type. For example, the following code example produces an error:

```
CREATE ROW TYPE serialtype (s serial, s8 serial8);
CREATE TABLE tab1 (col1 serialtype) --INVALID CODE
```

You cannot create a ROW type that has a BYTE or TEXT value that is stored in a separate storage space. That is, you cannot use the IN clause to specify the storage location. For example, the following example produces an error:

```
CREATE ROW TYPE row1 (field1 byte IN blobspace1) --INVALID CODE
```

Across a table hierarchy, you can use only one SERIAL and one SERIAL8. That is, if a supertable table contains a SERIAL column, no subtable can contain a SERIAL column. However, a subtable can have a SERIAL8 column (as long as no other subtables contain a SERIAL8 column). Consequently, when you create the named ROW types on which the table hierarchy is to be based, they can contain at most one SERIAL and one SERIAL8 field among them.

You cannot set the starting SERIAL or SERIAL8 value with CREATE ROW TYPE. To modify the value for a serial field, you must use either the MODIFY clause of the ALTER TABLE statement or the INSERT statement to insert a value that is larger than the current maximum (or default) serial value.

Serial fields in ROW types havte performance implications across a table hierarchy. To insert data into a subtable whose supertable (or its supertable) contains the serial counter, the database server must also open the supertable, update the serial value, and close the supertable, thus adding extra overhead.

Related Information

Related statements: DROP ROW TYPE, CREATE TABLE, CREATE CAST, GRANT, and REVOKE

For a discussion of named ROW types, see the *IBM Informix Database Design and Implementation Guide* and the *IBM Informix Guide to SQL: Reference*.

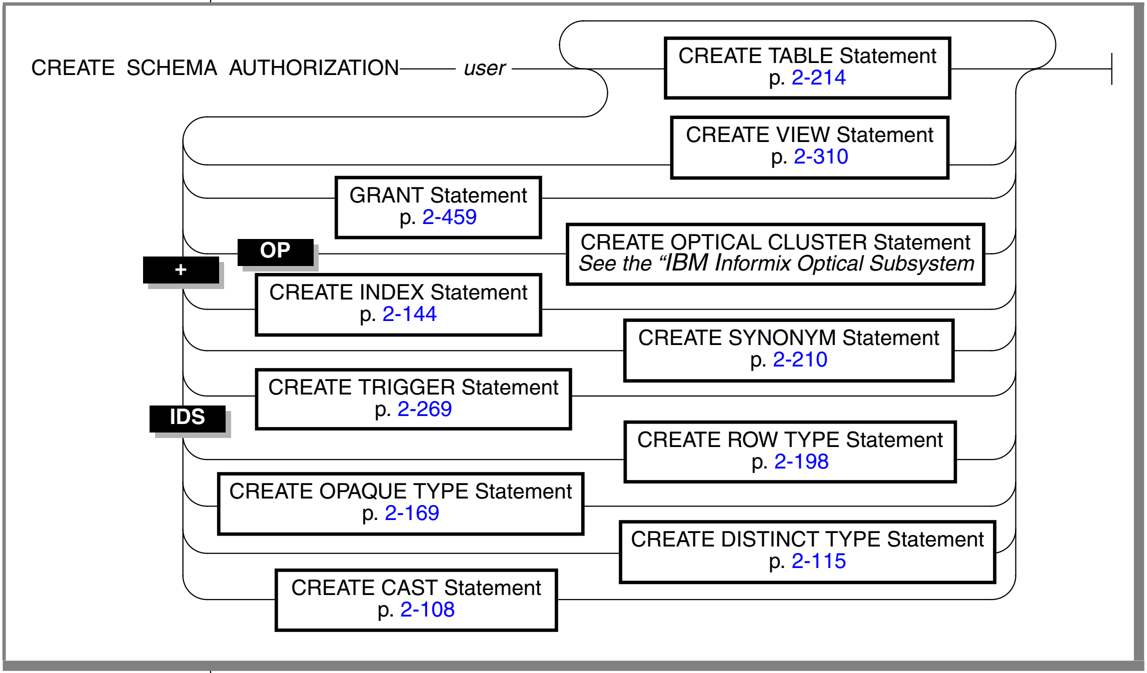
DB

SQLE

CREATE SCHEMA

Use the CREATE SCHEMA statement to issue a block of data definition language (DDL) and GRANT statements as a logical unit. Use this statement with DB-Access.

Syntax


CREATE SCHEMA AUTHORIZATION
user
CREATE TABLE Statement p. 2-214
CREATE VIEW Statement p. 2-310
GRANT Statement p. 2-459
OP
CREATE OPTICAL CLUSTER Statement See the "IBM Informix Optical Subsystem
+
CREATE INDEX Statement p. 2-144
CREATE SYNONYM Statement p. 2-210
CREATE TRIGGER Statement p. 2-269
IDS
CREATE ROW TYPE Statement p. 2-198
CREATE OPAQUE TYPE Statement p. 2-169
CREATE DISTINCT TYPE Statement p. 2-115
CREATE CAST Statement p. 2-108

Element	Purpose	Restrictions	Syntax
user	User who owns the database objects that this statement creates	If you have DBA privileges, you can specify the name of any user. Otherwise, you must have the Resource privilege and you must specify your own user name.	Identifier, p. 4-189

Usage

The CREATE SCHEMA statement allows the DBA to specify an owner for all database objects that the CREATE SCHEMA statement creates. You cannot issue CREATE SCHEMA until you create the database that stores the objects.

Users with the Resource privilege can create a schema for themselves. In this case, *user* must be the name of the person with the Resource privilege who is running the CREATE SCHEMA statement. Anyone with the DBA privilege can also create a schema for someone else. In this case, *user* can identify a user other than the person who is running the CREATE SCHEMA statement.

You can put CREATE and GRANT statements in any logical order, as the following example shows. Statements are considered part of the CREATE SCHEMA statement until a semicolon or an end-of-file symbol is reached.

```
CREATE SCHEMA AUTHORIZATION sarah
   CREATE TABLE mytable (mytime DATE, mytext TEXT)
   GRANT SELECT, UPDATE, DELETE ON mytable TO rick
   CREATE VIEW myview AS
      SELECT * FROM mytable WHERE mytime > '12/31/1997'
   CREATE INDEX idxtime ON mytable (mytime);
```

Creating Database Objects Within CREATE SCHEMA

All database objects that a CREATE SCHEMA statement creates are owned by *user*, even if you do not explicitly name each database object. If you are the DBA, you can create database objects for another user. If you are not the DBA, specifying an owner other than yourself results in an error message.

You can only grant privileges with the CREATE SCHEMA statement; you cannot revoke or drop privileges.

If you create a database object or use the GRANT statement outside a CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**.

Related Information

Related statements: CREATE INDEX, CREATE SYNONYM, CREATE TABLE, CREATE VIEW, and GRANT

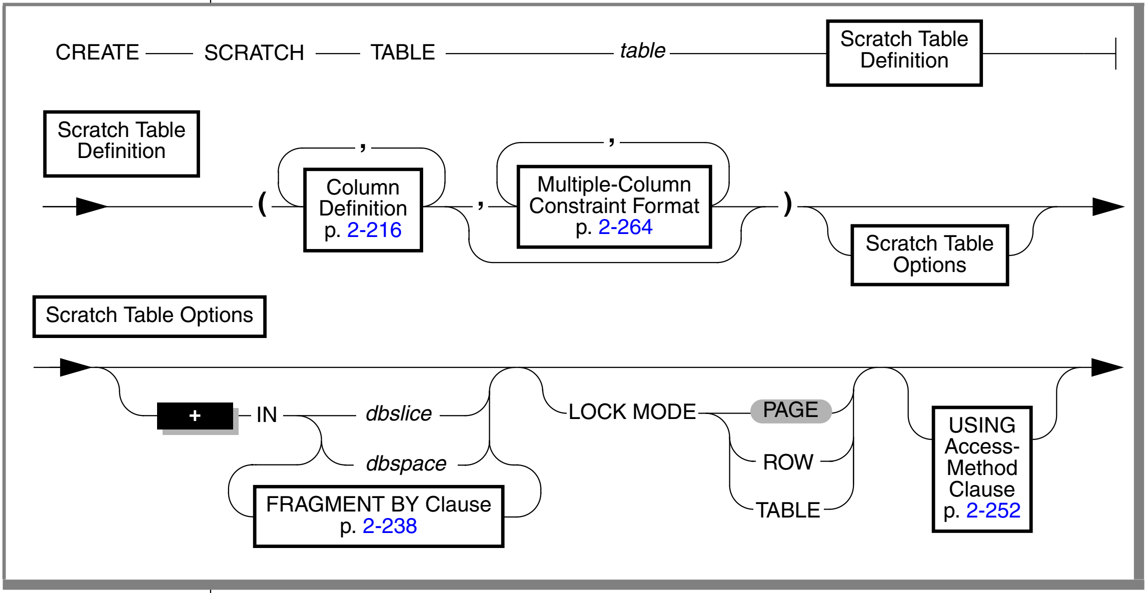
For a discussion of how to create a database, see the *IBM Informix Database Design and Implementation Guide*.

XPS

CREATE SCRATCH TABLE

Use the CREATE SCRATCH TABLE statement to create a non-logging temporary table in the current Extended Parallel Server database.

Syntax



Element	Purpose	Restrictions	Syntax
dbslice	Name of dbslice to store *table*	Must already exist.	Identifier, p. 4-189
dbspace	Name of dbspace to store *table*. Default is the dbspace that stores the current database.	Must already exist.	Identifier, p. 4-189
table	Name that you declare here for a nonlogging temporary table	Must be unique in the current session.	Database Object Name, p. 4-46

Usage

CREATE SCRATCH TABLE is a special case of the CREATE Temporary TABLE statement. See "CREATE Temporary TABLE" on page 2-261.

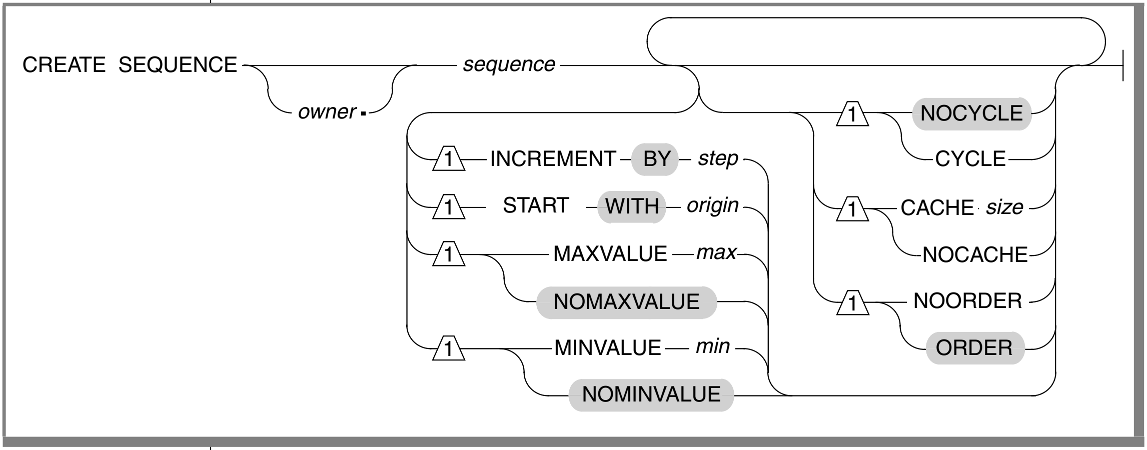
+

IDS

CREATE SEQUENCE

Use the CREATE SEQUENCE statement to create a new sequence. A *sequence* is a database object from which multiple users can generate unique integers.

Syntax



Element	Purpose	Restrictions	Syntax
max	Upper limit of values	Must be an integer > *origin*	Literal number, p. 4-216
min	Lower limit of values	Must be an integer less than *origin*	Literal number, p. 4-216
origin	First number in the sequence	Must be an integer in INT8 range	Literal number, p. 4-216
owner	Owner of *sequence*	Must be aauthorization identifier	Owner Name, p. 4-234
sequence	Name that you declare here for the new sequence	Must be unique among sequence, table, view, and synonym names	Identifier, p. 4-189
size	Number of values that are preallocated in memory	Integer > 1, but < cardinality of a cycle (= |(*max* - *min*)/*step*|)	Literal number, p. 4-216
step	Interval between successive values	Nonzero integer in INT range	Literal number, p. 4-216

Usage

A sequence (sometimes called a *sequence generator*) returns a monotonically ascending or descending series of unique integers, one at a time. The CREATE SEQUENCE statement defines a new sequence and declares its identifier.

Authorized users of a sequence can request a new value by including the ***sequence*.NEXTVAL** expression in SQL statements. The ***sequence*.CURRVAL** expression returns the current value of the specified *sequence*.

Generated values logically resemble the SERIAL8 data type, but are unique within the sequence. Because the database server generates the values, sequences support a much higher level of concurrency than a serial column can. The values are independent of transactions; a generated value cannot be rolled back, even if the transaction in which it was generated fails.

You can use a sequence to generate primary key values automatically, using one sequence for many tables, or each table can have its own sequence.

CREATE SEQUENCE can specify the following characteristics of a sequence:

- Initial value
- Size and sign of the increment between values.
- Maximum and minimum values
- Whether the sequence recycles values after reaching its limit
- How many values are preallocated in memory for rapid access

A database can support multiple sequences concurrently, but the name of a sequence must be unique within the current database among the names of tables, temporary tables, views, synonyms, and sequences.

ANSI

In an ANSI-compliant database, the *owner.sequence* combination must be unique among tables, temporary tables, views, synonyms, and sequences. ♦

An error occurs if you include contradictory options, such as specifying both the MINVALUE and NOMINVALUE options, or both CACHE and NOCACHE.

INCREMENT BY Option

Use the INCREMENT BY option to specify the interval between successive numbers in the sequence. The interval, or *step* value, can be a positive whole number (for an *ascending* sequence) or a negative whole number (for a *descending* sequence) in the INT8 range. The BY keyword is optional.

If you do not specify any *step* value, the default interval between successive generated values is `1`, and the sequence is an ascending sequence.

START WITH Option

Use the START WITH option to specify the first number of the sequence. This *origin* value must be an integer within the INT8 range that is greater than or equal to the *min* value (for an ascending sequence) or that is less than or equal to the *max* value (for a descending sequence), if *min* or *max* is specified in the CREATE SEQUENCE statement. The WITH keyword is optional.

If you do not specify an *origin* value, the default initial value is *min* for an ascending sequence or *max* for a descending sequence. (The "MAXVALUE or NOMAXVALUE Option" and "MINVALUE or NOMINVALUE Option" sections that follow describe the *max* and *min* specifications respectively.)

MAXVALUE or NOMAXVALUE Option

Use the MAXVALUE option to specify the upper limit of values in a sequence. The maximum value, or *max*, must be an integer in the INT8 range that is greater than the value of the *origin*.

If you do not specify a *max* value, the default is NOMAXVALUE. This default setting supports values that are less than or equal to `2e64` for ascending sequences, or less than or equal to `-1` for descending sequences.

MINVALUE or NOMINVALUE Option

Use the MINVALUE option to specify the lower limit of values in a sequence. The minimum value, or *min*, must be an integer in the INT8 range that is less than the value of the *origin*.

If you do not specify a *min* value, the default is NOMINVALUE. This default setting supports values that are greater than or equal to `1` for ascending sequences, or greater than or equal to `-(2e64)` for descending sequences.

CYCLE or NOCYCLE Option

Use the CYCLE option to continue generating sequence values after the sequence reaches the maximum (ascending) or minimum (descending) limit. After an ascending sequence reaches the *max* value, it generates the *min* value for the next sequence value. After a descending sequence reaches the *min* value, it generates the *max* value for the next sequence value.

The default is NOCYCLE. At this default setting, the sequence cannot generate more values after reaching the declared limit. Once the sequence reaches the limit, the next reference to *sequence*.NEXTVAL returns an error.

CACHE or NOCACHE Option

Use the CACHE option to specify the number of sequence values that are preallocated in memory for rapid access. This feature can enhance the performance of a heavily used sequence. The cache *size* must be a positive whole number in the INT range. If you specify the CYCLE option, then *size* must be less than the number of values in a cycle (or less than |(*max* - *min*)/*step*|). The minimum is `2` preallocated values. The default is `20` preallocated values.

The NOCACHE keyword specifies that no generated values (that is, zero) are preallocated in memory for this sequence object.

The configuration parameter SEQ_CACHE_SIZE specifies the maximum number of sequence objects that can have preallocated values in the sequence cache. If this configuration parameter is not set, then by default no more than 10 different sequence objects can be defined with the CACHE option.

ORDER or NOORDER Option

These keywords have no effect on the behavior of the sequence. The sequence always issues values to users in the order of their requests, as if the ORDER keyword were always specified. The ORDER and NOORDER keywords are accepted by the CREATE SEQUENCE statement for compatibility with implementations of sequence objects in other dialects of SQL.

Related Information

Related statements: ALTER SEQUENCE, DROP SEQUENCE, RENAME SEQUENCE, CREATE SYNONYM, DROP SYNONYM, GRANT, REVOKE, INSERT, UPDATE, and SELECT

For information about the **syssequences** system catalog table in which sequence objects are registered, see the *IBM Informix Guide to SQL: Reference*.

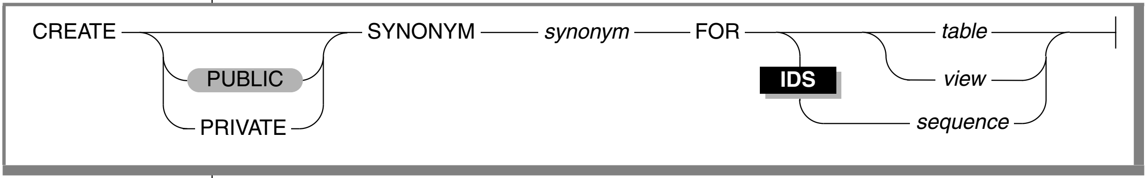
For information about initializing a sequence and generating or reading values from a sequence, see "NEXTVAL and CURRVAL Operators" on page 4-102.

+

CREATE SYNONYM

Use the CREATE SYNONYM statement to declare and register an alternative name for an existing table, view, or sequence object.

Syntax



Element	Purpose	Restrictions	Syntax
sequence	Name of a local sequence	Must exist in current database	Identifier, p. 4-189
table, *view*	Name of table or view for which *synonym* is being created	Must exist in current database, or in a database specified in a qualifier	Database Object Name, p. 4-46
synonym	Synonym declared here for the name of a table, view, or sequence	Must be unique among table object names; see also **Usage** notes.	Database Object Name, p. 4-46

Usage

Users have the same privileges for a synonym that they have for the database object that the synonym references. The **syssynonyms**, **syssyntable**, and **systables** system catalog tables maintain information about synonyms.

You cannot create a synonym for a synonym in the same database.

The identifier of the synonym must be unique among the names of tables, temporary tables, views, and sequence objects in the same database. (See, however, the section "Synonyms with the Same Name" on page 2-212.)

Once a synonym is created, it persists until the owner executes the DROP SYNONYM statement. (This persistence distinguishes a synonym from an alias that you can declare in the FROM clause of a SELECT statement; the alias is in scope only during execution of that SELECT statement.) If a synonym refers to a table, view, or sequence in the same database, the synonym is automatically dropped if the referenced table, view, or sequence is dropped.

Synonyms for Remote and External Tables and Views

A synonym can be created for any table or view in any database on your database server. This example declares a synonym for a table outside your current database, in the **payables** database of your current database server.

```
CREATE SYNONYM mysum FOR payables:jean.summary
```

You can also create a synonym for a table or view that exists in a database of a database server that is not your current database server. Both database servers must be online when you create the synonym. In a network, the remote database server verifies that the table or view referenced by the synonym exists when you create the synonym. The next example reates a synonym for a table supported by a remote database server:

```
CREATE SYNONYM mysum FOR payables@phoenix:jean.summary
```

The identifier **mysum** now refers to the table **jean.summary**, which is in the **payables** database on the **phoenix** database server. If the **summary** table is dropped from the **payables** database, the **mysum** synonym is left intact. Subsequent attempts to use **mysum** return the error `Table not found`.

IDS

You cannot create synonyms, however, for these external objects:

- Typed tables (including any table that is part of a table hierarchy)
- Tables or views that contain any extended data types.
- Sequence objects outside the local database ♦

PUBLIC and PRIVATE Synonyms

If you use the PUBLIC keyword (or no keyword at all), anyone who has access to the database can use your synonym. If the database is not ANSI-compliant, a user does not need to know the name of the owner of a public synonym. Any synonym in a database that is not ANSI compliant *and* was created in an Informix database server earlier than Version 5.0 is a public synonym.

ANSI

In an ANSI-compliant database, all synonyms are private. If you use the PUBLIC or PRIVATE keywords, the databasde server issues a syntax error. ♦

If you use the PRIVATE keyword to declare a synonym in a database that is not ANSI-compliant, the unqualified synonym can be used by its owner. Other users must qualify the synonym with the name of the owner.

Synonyms with the Same Name

ANSI

In an ANSI-compliant database, the *owner*.*synonym* combination must be unique among all synonyms, tables, views. and sequences . You must specify *owner* when you refer to a synonym that you do not own, as in this example:

```
CREATE SYNONYM emp FOR accting.employee  ♦
```

In a database that is not ANSI-compliant, no two public synonyms can have the same identifier, and the identifier of a synonym must also be unique among the names of tables, views, and sequences in the same database.

The *owner*.*synonym* combination of a private synonym must be unique among all the synonyms in the database. That is, more than one private synonym with the same name can exist in the same database, but a different user must own each of these synonyms. The same user cannot create both a private and a public synonym that have the same name. For example, the following code generates an error:

```
CREATE SYNONYM our_custs FOR customer;
CREATE PRIVATE SYNONYM our_custs FOR cust_calls;-- ERROR!!!
```

A private synonym can be declared with the same name as a public synonym only if the two synonyms have different owners. If you own a private synonym, and a public synonym exists with the same name, the database server resolves the unqualified name as the private synonym. (In this case, you must specify *owner*.*synonym* to reference the public synonym.) If you use DROP SYNONYM with the unqualified synonym identifier when your private synonym and the public synonym of another user both have the same identifier, only your private synonym is dropped. If you repeat the same DROP SYNONYM statement, the database server drops the public synonym.

Chaining Synonyms

If you create a synonym for a table or view that is not in the current database, and this table or view is dropped, the synonym stays in place. You can create a new synonym for the dropped table or view with the name of the dropped table or view as the synonym, which points to another external or remote table or view. (Synonyms for external sequence objects are not supported.)

In this way, you can move a table or view to a new location and chain synonyms together so that the original synonyms remain valid. (You can chain up to 16 synonyms in this manner.)

The following steps chain two synonyms together for the **customer** table, which will ultimately reside on the **zoo** database server (the CREATE TABLE statements are not complete):

1. In the **stores_demo** database on the database server that is called **training**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...)
   ```

2. On the database server called **accntg**, issue the following statement:

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer
   ```

3. On the database server called **zoo**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...)
   ```

4. On the database server called **training**, issue the following statement:

   ```
   DROP TABLE customer
   CREATE SYNONYM customer FOR stores_demo@zoo:customer
   ```

The synonym **cust** on the **accntg** database server now points to the **customer** table on the **zoo** database server.

The following steps show an example of chaining two synonyms together and changing the table to which a synonym points:

1. On the database server called **training**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...)
   ```

2. On the database server called **accntg**, issue the following statement:

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer
   ```

3. On the database server called **training**, issue the following statement:

   ```
   DROP TABLE customer
   CREATE TABLE customer (lastname CHAR(20)...)
   ```

The synonym **cust** on the **accntg** database server now points to a new version of the **customer** table on the **training** database server.

Related Information

Related statement: DROP SYNONYM

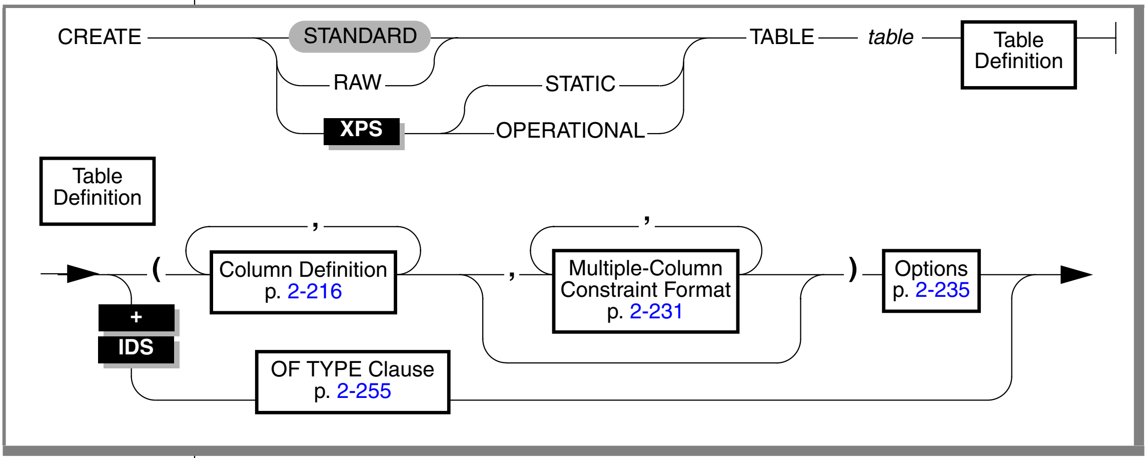
For a discussion of concepts related to synonyms, see the *IBM Informix Database Design and Implementation Guide*.

CREATE TABLE

Use the CREATE TABLE statement to create a new table in the current database, to place data-integrity constraints on columns, to designate where the table should be stored, to indicate the size of its initial and subsequent extents, and to specify how to lock the new table.

You can use the CREATE TABLE statement to create relational-database tables or typed tables (object-relational tables). For information on how to create temporary tables, see "CREATE Temporary TABLE" on page 2-261.

Syntax



Element	Purpose	Restrictions	Syntax
table	Name that you declare here for the new table	Must be unique among names of tables, synonyms, views, and sequences within the current database	Database Object Name, p. 4-46

Usage

When you create a new table, every column must have a data type associated with it. The *table* name must be unique among all the names of tables, views, sequences, and synonyms within the same database, but the names of columns need only be unique among the column names of the same table.

ANSI

In an ANSI-compliant database, the combination *owner.table* must be unique within the database. ♦

DB

In DB-Access, using the CREATE TABLE statement outside the CREATE SCHEMA statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

E/C

In ESQL/C, using the CREATE TABLE statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

For information about the **DBANSIWARN** environment variable, refer to the *IBM Informix Guide to SQL: Reference*.

Logging Options

Use the Logging Type options to specify characteristics that can improve performance in various bulk operations on the table. Other than the default option (STANDARD) that is used for OLTP databases, these logging options are used primarily to improve performance in data warehousing databases.

A table can have either of the following logging characteristics.

Logging Type	Description
STANDARD	Logging table that allows rollback, recovery, and restoration from archives. This type is the default. Use this type of table for all the recovery and constraints functionality that OLTP databases require.
RAW	Nonlogging table that cannot have indexes or referential constraints but can be updated. Use this type of table for quickly loading data. .

XPS

By using raw tables with Extended Parallel Server, you can take advantage of light appends and avoid the overhead of logging, checking constraints, and building indexes. ♦

Warning: *Use raw tables for fast loading of data, but set the logging type to STANDARD and perform a level-0 backup before you use the table in a transaction or modify the data within the table. If you must use a raw table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.*

XPS

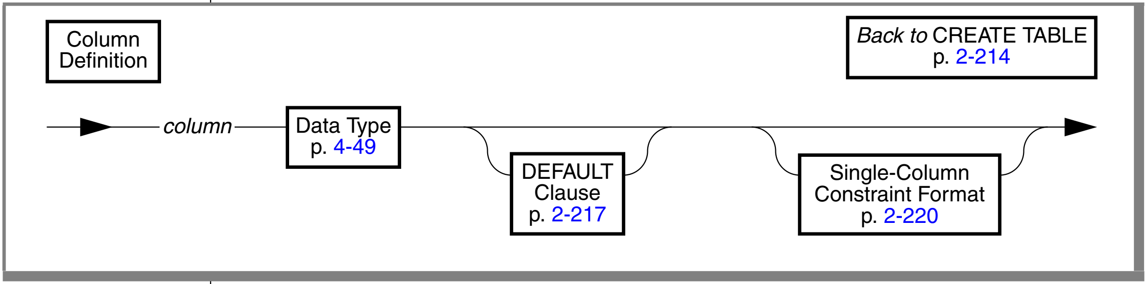
Extended Parallel Server supports two additional logging type options.

Option	Effect
OPERATIONAL	Logging table that uses light appends; it cannot be restored from archive. Use this type on tables that are refreshed frequently, because light appends allow the quick addition of many rows.
STATIC	Nonlogging table that can contain index and referential constraints but cannot be updated. Use this type for read-only operations, because no logging or locking overhead occurs. ♦

For more information on these logging types of tables, refer to your *Administrator's Guide*.

Column Definition

Use the column definition portion of CREATE TABLE to list the name, data type, default values, and constraints of a *single column*.


Column Definition
Back to CREATE TABLE p. 2-214
column
Data Type p. 4-49
DEFAULT Clause p. 2-217
Single-Column Constraint Format p. 2-220

Element	Purpose	Restrictions	Syntax
column	Name of a column in the table	Must be unique in this table.	Identifier, p. 4-189

Because of the maximum row size limit of 32,767 bytes, no more than 195 columns in the table can be of the data types BYTE, TEXT, ROW, LVARCHAR, NVARCHAR, VARCHAR, and varying-length UDTs. Similarly, no more than 97 columns can be of COLLECTION data types (SET, LIST, and MULTISET).

As with any SQL identifier, syntactic ambiguities can occur if the *column* name is a keyword. For information on reserved words for Dynamic Server, see Appendix A, "Reserved Words for IBM Informix Dynamic Server."

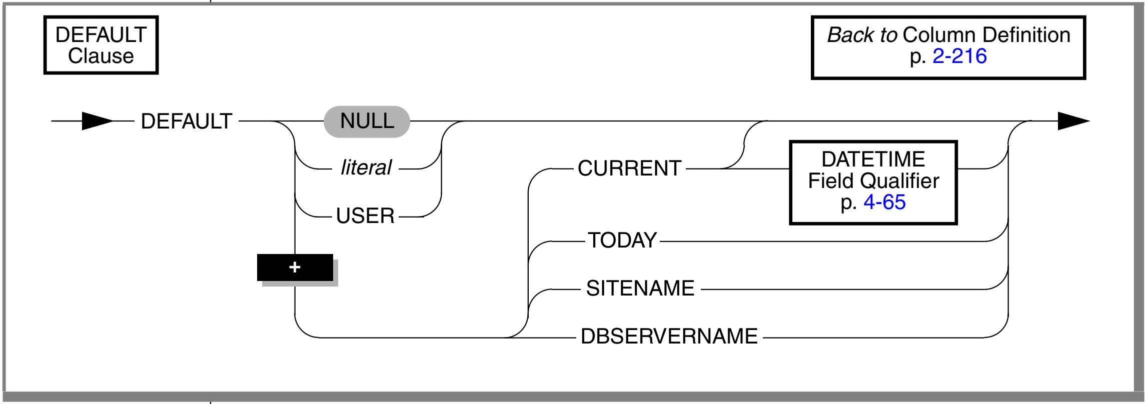
For more information on reserved words for Extended Parallel Server, see Appendix B, "Reserved Words for IBM Informix Extended Parallel Server." For more information on the ambiguities that can occur, see "Using Keywords as Column Names" on page 4-195.

IDS

If you define a column of a table to be of a named ROW type, the table does not adopt any constraints of the named ROW. ♦

DEFAULT Clause

Use the DEFAULT clause to specify the default value for the database server to insert into a column when no explicit value for the column is specified.



Element	Purpose	Restrictions	Syntax
literal	String of alphabetic or numeric characters	Must be an appropriate data type for the column. See "Using a Literal as a Default Value" on page 2-218.	Expression, p. 4-67

You cannot specify default values for SERIAL or SERIAL8 columns.

Using NULL as a Default Value

If you specify no default value for a column, the default is NULL unless you place a NOT NULL constraint on the column. In this case, no default exists.

If you specify NULL as the default value for a column, you cannot specify a NOT NULL constraint as part of the column definition. (For details of NOT NULL constraints, see"Using the NOT NULL Constraint" on page 2-221.)

NULL is not a valid default value for a column that is part of a primary key.

If the column is BYTE or TEXT data type, NULL is the only valid default value.

IDS

If the column is BLOB or CLOB data type, NULL is the only valid default value. ♦

Using a Literal as a Default Value

You can designate a *literal* value as a default value. A literal value is a string of alphabetic or numeric characters. To use a literal value as a default value, you must adhere to the syntax restrictions in the following table.

For Columns of Data Type	Format of Default Value
BOOLEAN	Use `'t'` or `'f'` (respectively for *true* or *false*) as a Quoted String, p. 4-243
CHAR, CHARACTER VARYING, DATE, VARCHAR, NCHAR, NVARCHAR	Quoted String, p. 4-243. See note that follows for DATE.
DATETIME	Literal DATETIME, p. 4-212
DECIMAL, MONEY, FLOAT, SMALLFLOAT	Literal Number, p. 4-216 (DECIMAL)
INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, SMALLFLOAT, INT8	Literal Number, p. 4-216 (INTEGER)
INTERVAL	Literal INTERVAL, p. 4-214
Opaque data types (IDS only)	Quoted String, p. 4-243 in Single-Column Constraint format (p. 2-220)

DATE literals must be of the format that the **DBDATE** (or else **GL_DATE**) environment variable specifies. In the default locale, if neither **DBDATE** nor **GL_DATE** is set, DATE literals must be of the *mm/dd/yyyy* format.

Using a Built-in Function as a Default Value

You can specify a built-in function as the default column value. The following table lists built-in functions that you can specify, the data type requirements, and the recommended size (in bytes) for their corresponding columns.

Built-In Function	Data Type Requirement	Recommended Size
CURRENT	DATETIME column with matching qualifier	Enough bytes to accommodate the longest DATETIME value for locale
DBSERVERNAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column	128 bytes
SITENAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column	128 bytes
TODAY	DATE column	Enough bytes to accommodate the longest DATE value for locale
USER	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column	32 bytes

These column sizes are recommended because, if the column length is too small to store the default value during INSERT or ALTER TABLE operations, the database server returns an error.

IDS

You cannot designate a built-in function (that is, CURRENT, USER, TODAY, SITENAME, or DBSERVERNAME) as the default value for a column that holds opaque or distinct data types. ♦

For descriptions of these functions, see "Constant Expressions" on page 4-95.

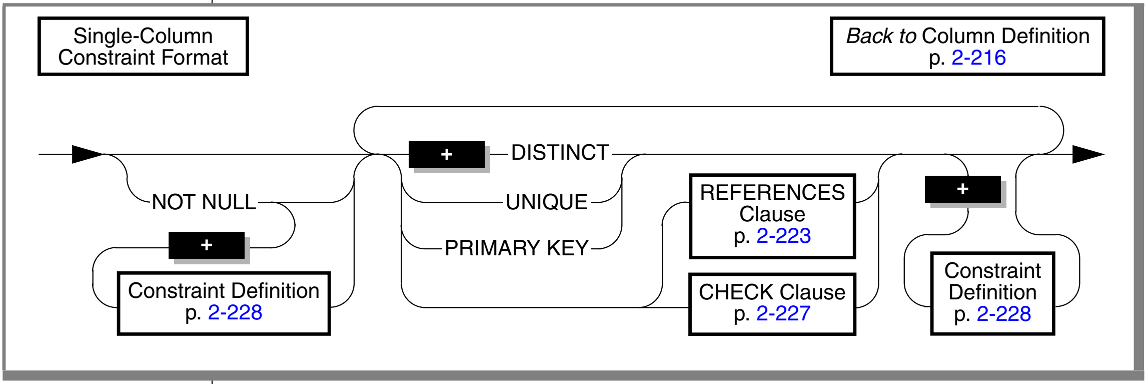
The following example creates a table called **accounts**. In **accounts**, the **acc_num, acc_type**, and **acc_descr** columns have literal default values. The **acc_id** column defaults to the login name of the user.

```
CREATE TABLE accounts (
   acc_num INTEGER DEFAULT 1,
   acc_type CHAR(1) DEFAULT 'A',
   acc_descr CHAR(20) DEFAULT 'New Account',
   acc_id CHAR(32) DEFAULT USER)
```

Single-Column Constraint Format

Use the Single-Column Constraint format to associate one or more constraints with a column, in order to perform any of the following tasks:

- Create one or more data-integrity constraints for a column.
- Specify a meaningful name for a constraint.
- Specify the constraint-mode that controls the behavior of a constraint during INSERT, DELETE, and UPDATE operations.


Single-Column Constraint Format
Back to Column Definition p. 2-216
NOT NULL
+
Constraint Definition p. 2-228
+
DISTINCT
UNIQUE
PRIMARY KEY
REFERENCES Clause p. 2-223
CHECK Clause p. 2-227
+
Constraint Definition p. 2-228

The following example creates a standard table with two constraints: **num**, a primary-key constraint on the **acc_num** column; and **code,** a unique constraint on the **acc_code** column:

```
CREATE TABLE accounts (
   acc_num   INTEGER PRIMARY KEY CONSTRAINT num,
   acc_code  INTEGER UNIQUE CONSTRAINT code,
   acc_descr CHAR(30))
```

The constraints used in this example are defined in the following sections.

Restrictions on Using the Single-Column Constraint Format

The single-column constraint format cannot specify a constraint that involves more than one column. Thus, you cannot use the single-column constraint format to define a composite key. For information on multiple-column constraints, see "Multiple-Column Constraint Format" on page 2-231.

You cannot place unique, primary-key, or referential constraints on BYTE or TEXT columns. You can, however, check for NULL or non-NULL values with a check constraint.

IDS

You cannot place unique constraints, primary-key constraints, or referential constraints on BLOB or CLOB columns. ♦

Using the NOT NULL Constraint

Use the NOT NULL keywords to require that a column receive a value during insert or update operations. If you place a NOT NULL constraint on a column (and no default value is specified), you *must* enter a value into this column when you insert a row or update that column in a row. If you do not enter a value, the database server returns an error, because no default value exists.

The following example creates the **newitems** table. In **newitems**, the column **manucode** does not have a default value nor does it allow NULLs.

```
CREATE TABLE newitems (
   newitem_num INTEGER,
   manucode CHAR(3) NOT NULL,
   promotype INTEGER,
   descrip CHAR(20))
```

You cannot specify NULL as the explicit default value for a column and also specify the NOT NULL constraint.

Using the UNIQUE or DISTINCT Constraints

Use the UNIQUE or DISTINCT keyword to require that a column or set of columns accepts only unique data values. You cannot insert values that duplicate the values of some other row into a column that has a unique constraint. When you create a UNIQUE or DISTINCT constraint, the database server automatically creates an internal index on the constrained column or columns. (In this context, the keyword DISTINCT is a synonym for UNIQUE.)

You cannot place a unique constraint on a column that already has a primary-key constraint.

You cannot place a unique constraint on a BYTE or TEXT column.

IDS

You cannot place a unique or primary-key constraint on a BLOB or CLOB column.

Opaque data types support a unique constraint only where a secondary-access method supports uniqueness for that type. The default secondary-access method is a generic B-tree, which supports the **equal( )** operator function. Therefore, if the definition of the opaque type includes the **equal( )** function, a column of that opaque type can have a unique constraint. ♦

The following example creates a simple table that has a unique constraint on one of its columns:

```
CREATE TABLE accounts
   (acc_name  CHAR(12),
    acc_num   SERIAL UNIQUE CONSTRAINT acc_num)
```

For an explanation of the constraint name, refer to "Declaring a Constraint Name" on page 2-229.

Using the PRIMARY KEY Constraint

A *primary key* is a column (or a set of columns, when you use the multiple-column constraint format) that contains a non-NULL, unique value for each row in a table. When you create a PRIMARY KEY constraint, the database server automatically creates an internal index on the column or columns that make up the primary key.

You can designate only one primary key for a table. If you define a single column as the primary key, then it is unique by definition. You cannot explicitly give the same column a unique constraint.

IDS

You cannot place a unique or primary-key constraint on a BLOB or CLOB column.

Opaque types support a primary key constraint only where a secondary-access method supports the uniqueness for that type. The default secondary-access method is a generic B-tree, which supports the **equal()** function. Therefore, if the definition of the opaque type includes the **equal()** function, a column of that opaque type can have a primary-key constraint. ♦

You cannot place a primary-key constraint on a BYTE or TEXT column.

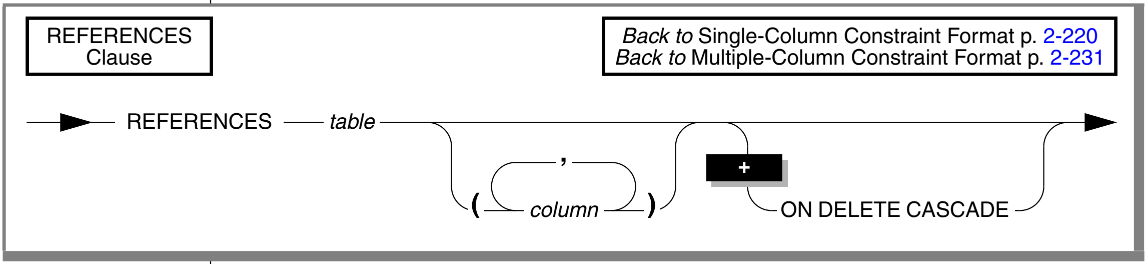
In the previous two examples, a unique constraint was placed on the column **acc_num**. The following example creates this column as the primary key for the **accounts** table:

```
CREATE TABLE accounts
   (acc_name  CHAR(12),
    acc_num   SERIAL PRIMARY KEY CONSTRAINT acc_num)
```

REFERENCES Clause

Use the REFERENCES clause to establish a referential relationship:

- Within a table (that is, between two columns of the same table)
- Between two tables (in other words, create a foreign key)


REFERENCES Clause
Back to Single-Column Constraint Format p. 2-220
Back to Multiple-Column Constraint Format p. 2-231
REFERENCES
table
,
(
column
)
+
ON DELETE CASCADE

Element	Purpose	Restrictions	Syntax
column	Name of the referenced column or columns	See "Restrictions on Referential Constraints" on page 2-224.	Identifier, p.4-189
table	Name of the referenced table	Must reside in the same database as the referencing table.	Database Object Name, p. 4-46

The *referencing* column (the column being defined) is the column or set of columns that refers to the referenced column or set of columns. The referencing column(s) can contain NULL and duplicate values, but values in the referenced column (or set of columns) must be unique.

The relationship between referenced and referencing columns is called a *parent-child* relationship, where the parent is the referenced column (primary key) and the child is the referencing column (foreign key). The referential constraint establishes this parent-child relationship.

When you create a referential constraint, the database server automatically creates an internal index on the constrained column or columns.

Restrictions on Referential Constraints

You must have the References privilege to create a referential constraint.

When you use the REFERENCES clause, you must observe the following restrictions:

- The referenced and referencing tables must be in the same database.
- The referenced column (or set of columns when you use the multiple-column constraint format) must have a unique or primary-key constraint.
- The data types of the referencing and referenced columns must be identical.

  The only exception is that a referencing column must be an integer data type if the referenced column is a serial.
- You cannot place a referential constraint on a BYTE or TEXT column.
- When you use the single-column constraint format, you can reference only one column.

XPS

- When you use the multiple-column constraint format, the maximum number of columns in the REFERENCES clause is 16, and the total length of the columns cannot exceed 380 bytes. ♦

IDS

- When you use the multiple-column constraint format, the maximum number of columns in the REFERENCES clause is 16, and the total length of the columns cannot exceed 390 bytes.
- You cannot place a referential constraint on a BLOB or CLOB column. ♦

Default Values for the Referenced Column

If the referenced table is different from the referencing table, you do not need to specify the referenced column; the default column is the primary-key column (or columns) of the referenced table. If the referenced table is the same as the referencing table, you must specify the referenced column.

Referential Relationships Within a Table

You can establish a referential relationship between two columns of the same table. In the following example, the **emp_num** column in the **employee** table uniquely identifies every employee through an employee number. The **mgr_num** column in that table contains the employee number of the manager who manages that employee. In this case, **mgr_num** references **emp_num**. Duplicate values appear in the **mgr_num** column because managers manage more than one employee.

```
CREATE TABLE employee
   (
   emp_num INTEGER PRIMARY KEY,
   mgr_num INTEGER REFERENCES employee (emp_num)
   )
```

Locking Implications of Creating a Referential Constraint

When you create a referential constraint, an exclusive lock is placed on the referenced table. The lock is released when the CREATE TABLE statement is finished. If you are creating a table in a database with transactions, and you are using transactions, the lock is released at the end of the transaction.

Example That Uses the Single-Column Constraint Format

The following example uses the single-column constraint format to create a referential relationship between the **sub_accounts** and **accounts** tables. The **ref_num** column in the **sub_accounts** table references the **acc_num** column (the primary key) in the **accounts** table.

```
CREATE TABLE accounts (
   acc_num INTEGER PRIMARY KEY,
   acc_type INTEGER,
   acc_descr CHAR(20))
CREATE TABLE sub_accounts (
   sub_acc INTEGER PRIMARY KEY,
   ref_num INTEGER REFERENCES accounts (acc_num),
   sub_descr CHAR(20))
```

When you use the single-column constraint format, you do not explicitly specify the **ref_num** column as a foreign key. To use the FOREIGN KEY keyword, use the "Multiple-Column Constraint Format" on page 2-231.

Using the ON DELETE CASCADE Option

Use the ON DELETE CASCADE option to specify whether you want rows deleted in a child table when corresponding rows are deleted in the parent table. If you do not specify cascading deletes, the default behavior of the database server prevents you from deleting data in a table if other tables reference it.

If you specify this option, later when you delete a row in the parent table, the database server also deletes any rows associated with that row (foreign keys) in a child table. The principal advantage to the cascading-deletes feature is that it allows you to reduce the quantity of SQL statements you need to perform delete actions.

For example, the **all_candy** table contains the **candy_num** column as a primary key. The **hard_candy** table refers to the **candy_num** column as a foreign key. The following CREATE TABLE statement creates the **hard_candy** table with the cascading-delete option on the foreign key:

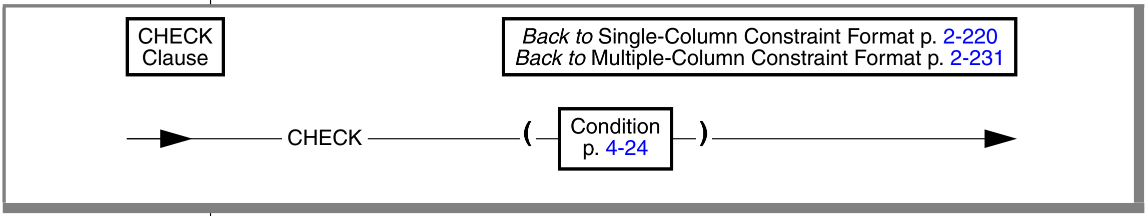
```
CREATE TABLE all_candy
   (candy_num SERIAL PRIMARY KEY,
    candy_maker CHAR(25));

CREATE TABLE hard_candy
   (candy_num INT,
    candy_flavor CHAR(20),
    FOREIGN KEY (candy_num) REFERENCES all_candy
    ON DELETE CASCADE)
```

Because the ON DELETE CASCADE option is specified for the child table, when an item from the **all_candy** table is deleted, the delete cascades to the corresponding rows of the **hard_candy** table.

For information about syntax restrictions and locking implications when you delete rows from tables that have cascading deletes, see “Considerations When Tables Have Cascading Deletes” on page 2-346.

CHECK Clause

Use the CHECK clause to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement.


CHECK Clause
Back to Single-Column Constraint Format p. 2-220
Back to Multiple-Column Constraint Format p. 2-231
CHECK
(
Condition p. 4-24
)

IDS

The *condition* cannot include a user-defined function or procedure. ♦

During an insert or update, if the check constraint of a row evaluates to *false*, the database server returns an error. The database server does not return an error if a row evaluates to NULL for a check constraint. In some cases, you might want to use both a check constraint and a NOT NULL constraint.

Using a Search Condition

You use *search conditions* to define check constraints. The search condition cannot contain the following items: user-defined routines, subqueries, aggregates, host variables, or rowids. In addition, the search condition cannot contain the following built-in functions: CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY.

Warning: *When you specify a date value in a search condition, make sure you specify 4 digits for the year, so that the* ***DBCENTURY*** *environment variable has no effect on the condition. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can produce unpredictable results if the condition depends on an abbreviated year value. For more information on the* ***DBCENTURY*** *environment variable, see the "IBM Informix Guide to SQL: Reference." More generally, the database server saves the settings of environment variables from the time of creation of check constraints. If any of these settings are subsequently changed in a way that can affect the evaluation of a condition in a check constraint, the new settings are disregarded, and the original environment variable settings are used when the condition is evaluated.*

With a BYTE or TEXT column, you can check for NULL or not-NULL values. This constraint is the only constraint allowed on a BYTE or TEXT column.

Restrictions When Using the Single-Column Constraint Format

When you use the single-column constraint format to define a check constraint, the check constraint cannot depend on values in other columns of the table. The following example creates the **my_accounts** table that has two columns with check constraints, each in the single-column constraint format:

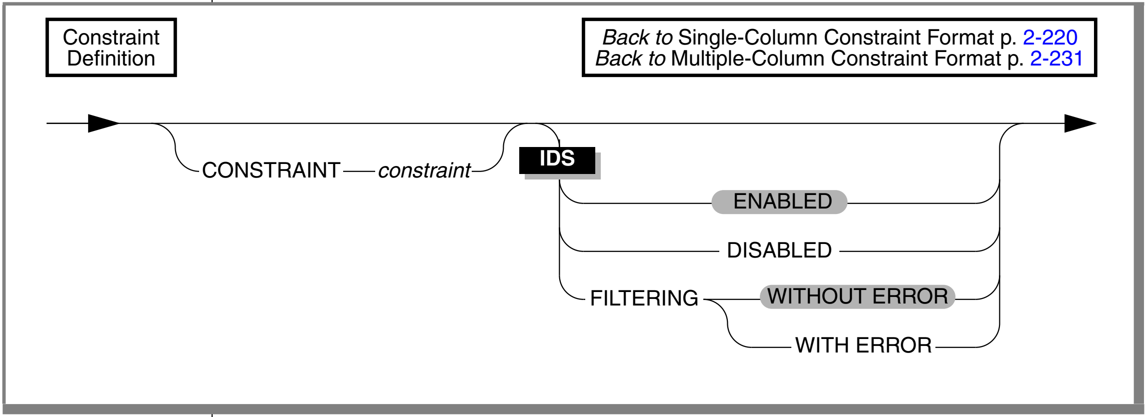
```
CREATE TABLE my_accounts (
   chk_id   SERIAL PRIMARY KEY,
   acct1    MONEY CHECK (acct1 BETWEEN 0 AND 99999),
   acct2    MONEY CHECK (acct2 BETWEEN 0 AND 99999))
```

Both **acct1** and **acct2** are columns of MONEY data type whose values must be between `0` and `99999`. If, however, you want to test that **acct1** has a larger balance than **acct2**, you cannot use the single-column constraint format. To create a constraint that checks values in more than one column, you must use the "Multiple-Column Constraint Format" on page 2-231.

Constraint Definition

Use the constraint definition portion of CREATE TABLE for these purposes:

- To declare a name for the constraint
- IDS To set a constraint to disabled, enabled, or filtering mode ♦


Constraint Definition
Back to Single-Column Constraint Format p. 2-220
Back to Multiple-Column Constraint Format p. 2-231
CONSTRAINT
constraint
IDS
ENABLED
DISABLED
FILTERING
WITHOUT ERROR
WITH ERROR

Element	Purpose	Restrictions	Syntax
constraint	Name of constraint	Must be unique among index names	Database Object Name, p. 4-46

Declaring a Constraint Name

The database server implements the constraint as an index. Whenever you use the single- or multiple-column constraint format to place a data restriction on a column, but without declaring a *constraint* name, the database server creates a constraint and adds a row for that constraint in the **sysconstraints** system catalog table. The database server also generates an identifier and adds a row to the **sysindexes** system catalog table for each new primary-key, unique, or referential constraint that does not share an index with an existing constraint. Even if you declare a name for a constraint, the database server generates the name that appears in the **sysindexes** table.

If you want, you can specify a meaningful name for the constraint. The name must be unique among the names of constraints and indexes in the database.

Constraint names appear in error messages having to do with constraint violations. You can use this name when you use the DROP CONSTRAINT clause of the ALTER TABLE statement.

IDS

In addition, you specify a constraint name when you change the mode of constraint with the SET Database Object Mode statement or the SET Transaction Mode statement. ♦

ANSI

When you create a constraint of any type, the combination of the *owner* name and *constraint* name must be unique within the database. ♦

IDS

The system catalog table that holds information about indexes is the **sysindices** table. ♦

Constraint Names That the Database Server Generates

If you do not specify a constraint name, the database server generates a constraint name using the following template:

```
<constraint_type><tabid>_<constraintid>
```

In this template, *constraint_type* is the letter **u** for unique or primary-key constraints, **r** for referential constraints, **c** for check constraints, and **n** for NOT NULL constraints. In the template, *tabid* and *constraintid* are values from the **tabid** and **constrid** columns of the **systables** and **sysconstraints** system catalog tables, respectively. For example, the constraint name for a unique constraint might look like ” **u111_14**” (with a leading blank space).

If the generated name conflicts with an existing identifier, the database server returns an error, and you must then supply an explicit constraint name.

The generated index name in **sysindexes** (or **sysindices**) has this format:

```
[blankspace]<tabid>_<constraintid>
```

For example, the index name might be something like " **111_14** " (with quotation marks used here to show the blank space).

IDS

Choosing a Constraint-Mode Option

Use the constraint-mode options to control the behavior of constraints in INSERT, DELETE, and UPDATE operations. These are the options.

Mode	Purpose
DISABLED	Does not enforce the constraint during INSERT, DELETE, and UPDATE operations.
ENABLED	Enforces the constraint during INSERT, DELETE, and UPDATE operations. If a target row causes a violation of the constraint, the statement fails. This mode is the default.
FILTERING	Enforces the constraint during INSERT, DELETE, and UPDATE operations. If a target row causes a violation of the constraint, the statement continues processing. The database server writes the row in question to the violations table associated with the target table and writes diagnostic information to the associated diagnostics table.

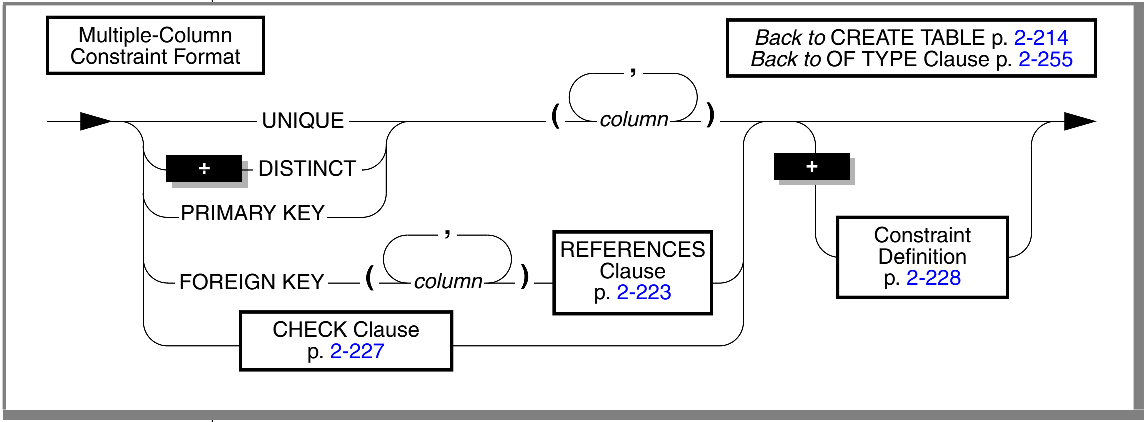
If you choose filtering mode, you can specify the WITHOUT ERROR or WITH ERROR options. The following list explains these options.

Error Option	Purpose
WITHOUT ERROR	Does not return an integrity-violation error when a filtering-mode constraint is violated during an insert, delete, or update operation. This is the default error option.
WITH ERROR	Returns an integrity-violation error when a filtering-mode constraint is violated during an insert, delete, or update operation

To set the constraint mode after the table exists, see "SET Database Object Mode" on page 2-652. For information about where the database server stores rows that violate a constraint set to FILTERING, see "START VIOLATIONS TABLE" on page 2-729.

Multiple-Column Constraint Format

Use the multiple-column constraint format to associate one or more columns with a constraint. This alternative to the single-column constraint format allows you to associate multiple columns with a constraint.


Multiple-Column Constraint Format
Back to CREATE TABLE p. 2-214
Back to OF TYPE Clause p. 2-255
UNIQUE
+
DISTINCT
PRIMARY KEY
FOREIGN KEY
(
,
column
)
REFERENCES Clause p. 2-223
CHECK Clause p. 2-227
+
Constraint Definition p. 2-228

Element	Purpose	Restrictions	Syntax
column	Columns on which to place constraint	Not BYTE, TEXT, BLOB, nor CLOB	Identifier, p.4-189

You can include a maximum of 16 columns in a constraint list. The total length of all columns cannot exceed 380 bytes.

When you define a unique constraint (by using the UNIQUE or DISTINCT keyword), a column cannot appear in the constraint list more than once.

Using the multiple-column constraint format, you can perform these tasks:

- Create data-integrity constraints for a set of one or more columns
- Specify a mnemonic name for a constraint
- Specify the constraint-mode option that controls the behavior of a constraint during insert, delete, and update operations

When you use this format, you can create composite primary and foreign keys, or define check constraints that compare data in different columns.

Restrictions with the Multiple-Column Constraint Format

When you use the multiple-column constraint format, you cannot define any default values for columns. In addition, you cannot establish a referential relationship between two columns of the same table.

To define a default value for a column or establish a referential relationship between two columns of the same table, refer to "Single-Column Constraint Format" on page 2-220 and "Referential Relationships Within a Table" on page 2-225 respectively.

Using Large-Object Types in Constraints

You cannot place unique, primary-key, or referential (FOREIGN KEY) constraints on BYTE or TEXT columns. You can, however, check for NULL or non-NULL values with a check constraint.

IDS

You cannot place unique or primary-key constraints on BLOB or CLOB columns. ♦

You can find detailed discussions of specific constraints in these sections.

Constraint	For more information, see	For an example, see
CHECK	"CHECK Clause" on page 2-227	"Defining Check Constraints Across Columns" on page 2-233
DISTINCT	"Using the UNIQUE or DISTINCT Constraints" on page 2-221	"Examples of the Multiple-Column Constraint Format" on page 2-233
FOREIGN KEY	"Using the FOREIGN KEY Constraint" on page 2-233	"Defining Composite Primary and Foreign Keys" on page 2-234
PRIMARY KEY	"Using the PRIMARY KEY Constraint" on page 2-222	"Defining Composite Primary and Foreign Keys" on page 2-234
UNIQUE	"Using the UNIQUE or DISTINCT Constraints" on page 2-221	"Examples of the Multiple-Column Constraint Format" on page 2-233

Using the FOREIGN KEY Constraint

A foreign key *joins* and establishes dependencies between tables. That is, it creates a referential constraint. (For more information on referential constraints, see the "REFERENCES Clause" on page 2-223.)

A foreign key references a unique or primary key in a table. For every entry in the foreign-key columns, a matching entry must exist in the unique or primary-key columns if all foreign-key columns contain non-NULL values.

You cannot specify BYTE or TEXT columns as foreign keys.

IDS

You cannot specify BLOB or CLOB columns as foreign keys. ♦

Examples of the Multiple-Column Constraint Format

The following example creates a standard table, called **accounts**, with a unique constraint, called **acc_num**, using the multiple-column constraint format. (Nothing in this example, however, would prevent you from using the single-column constraint format to define this constraint.)

```
CREATE TABLE accounts
   (acc_name CHAR(12),
    acc_num  SERIAL,
    UNIQUE  (acc_num) CONSTRAINT acc_num)
```

For constraint names, see "Declaring a Constraint Name" on page 2-229.

Defining Check Constraints Across Columns

When you use the multiple-column constraint format to define check constraints, a check constraint can apply to more than one column in the same table. (You cannot, however, create a check constraint whose *condition* uses a value from a column in another table.)

This example compares two columns, **acct1** and **acct2**, in the new table:

```
CREATE TABLE my_accounts
   (
   chk_id   SERIAL PRIMARY KEY,
   acct1    MONEY,
   acct2    MONEY,
   CHECK (0 < acct1 AND acct1 < 99999),
   CHECK (0 < acct2 AND acct2 < 99999),
   CHECK (acct1 > acct2)
   )
```

In this example, the **acct1** column must be greater than the **acct2** column, or the insert or update fails.

Defining Composite Primary and Foreign Keys

When you use the multiple-column constraint format, you can create a composite key. A *composite key* specifies multiple columns for a primary-key or foreign-key constraint.

The next example creates two tables. The first table has a composite key that acts as a primary key, and the second table has a composite key that acts as a foreign key.

```
CREATE TABLE accounts (
   acc_num INTEGER,
   acc_type INTEGER,
   acc_descr CHAR(20),
   PRIMARY KEY (acc_num, acc_type))

CREATE TABLE sub_accounts (
   sub_acc INTEGER PRIMARY KEY,
   ref_num INTEGER NOT NULL,
   ref_type INTEGER NOT NULL,
   sub_descr CHAR(20),
   FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
      (acc_num, acc_type))
```

In this example, the foreign key of the **sub_accounts** table, **ref_num** and **ref_type**, references the composite key, **acc_num** and **acc_type**, in the **accounts** table. If, during an insert or update, you tried to insert a row into the **sub_accounts** table whose value for **ref_num** and **ref_type** did not exactly correspond to the values for **acc_num** and **acc_type** in an existing row in the **accounts** table, the database server would return an error.

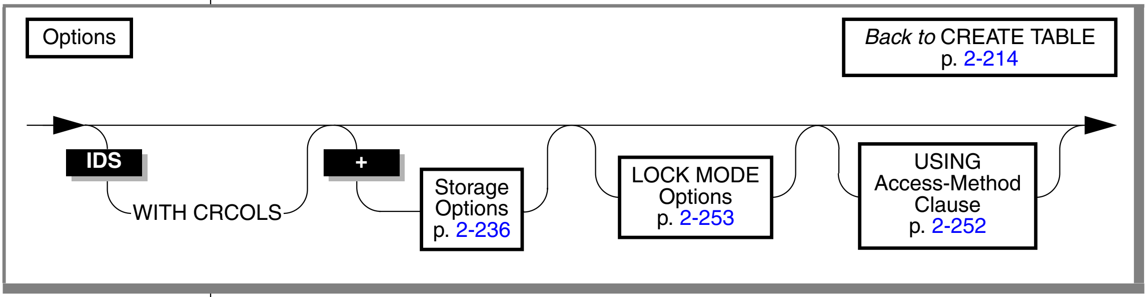
A referential constraint must have a one-to-one relationship between referencing and referenced columns. In other words, if the primary key is a set of columns (a composite key), then the foreign key also must be a set of columns that corresponds to the composite key.

Because of the default behavior of the database server, when you create the foreign-key reference, you do not have to reference the composite-key columns (**acc_num** and **acc_type**) explicitly. You can rewrite the references section of the previous example as follows:

```
   FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
```

Options

The CREATE TABLE options let you specify storage locations, extent size, locking modes, and user-defined access methods.



IDS

Using the WITH CRCOLS Option

Use the WITH CRCOLS keywords to create two shadow columns that Enterprise Replication uses for conflict resolution. The first column, **cdrserver**, contains the identity of the database server where the last modification occurred. The second column, **cdrtime**, contains the time stamp of the last modification. You must add these columns before you can use time-stamp or user-defined routine conflict resolution.

For most database operations, the **cdrserver** and **cdrtime** columns are hidden. For example, if you include the WITH CRCOLS keywords when you create a table, the **cdrserver** and **cdrtime** columns:

- Do not appear when you issue the statement

  ```
  SELECT * from tablename
  ```
- Do not appear in DB-Access when you ask for information about the columns of the table
- Are not included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*

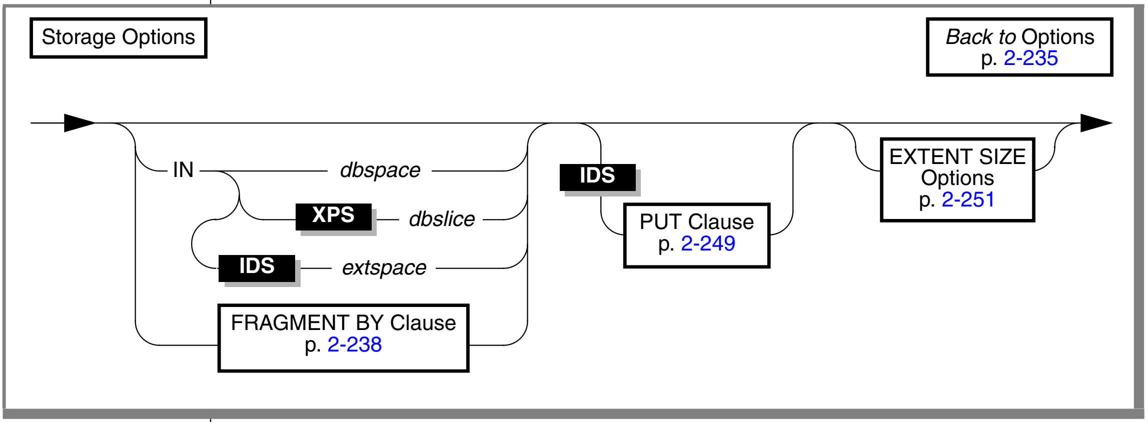
To view the contents of **cdrserver** and **cdrtime**, explicitly name the columns in a SELECT statement, as the following example shows:

```
SELECT cdrserver, cdrtime from tablename
```

For more information about how to use this option, refer to the *IBM Informix Dynamic Server Enterprise Replication Guide*.

Storage Options

Use the storage-option portion of CREATE TABLE to specify the storage space and the size of the extents for the table.



Element	Purpose	Restrictions	Syntax
dbslice	Dbslice to store the table	Must already exist.	Identifier, p. 4-189
dbspace	Dbspace to store the table	Must already exist.	Identifier, p. 4-189
extspace	Name declared in the **onspaces** command to a storage area outside the database server	Must already exist.	See documentation for your access method.

If you use the "USING Access-Method Clause" on page 2-252 to specify an access method, that method must support the storage space.

You can specify a dbspace for the table that is different from the storage location for the database, or you can fragment the table into several dbspaces. If you do not specify the IN clause or a fragmentation scheme, the database server stores the table in the dbspace where the current database resides.

IDS

You can use the PUT clause to specify storage options for smart large objects. For more information, see "PUT Clause" on page 2-249.

Tip: *If your table has columns that contain simple large objects (TEXT or BYTE), you can specify a separate blobspace for each object. For information on storing simple large objects, refer to "Large-Object Data Types" on page 4-57.* ♦

Using the IN Clause

Use the IN clause to specify a storage space for a table. The storage space that you specify must already exist.

Storing Data in a dbspace

You can use the IN clause to isolate a table. For example, if the **history** database is in the dbs1 dbspace, but you want the **family** data placed in a separate dbspace called famdata, use the following statements:

```
CREATE DATABASE history IN dbs1

CREATE TABLE family
   (
   id_num       SERIAL(101) UNIQUE,
   name         CHAR(40),
   nickname     CHAR(20),
   mother       CHAR(40),
   father       CHAR(40)
   )
   IN famdata
```

For more information about how to store and manage your tables in separate dbspaces, see your *Administrator's Guide*.

XPS

Storing Data in a dbslice

If you are using Extended Parallel Server, the IN *dbslice* clause allows you to fragment a table across a group of dbspaces that share the same naming convention. The database server fragments the table by round-robin in the dbspaces that make up the dbslice at the time the table is created.

To fragment a table across a dbslice, you can use either the IN *dbslice* syntax or the FRAGMENT BY ROUND ROBIN IN *dbslice* syntax.

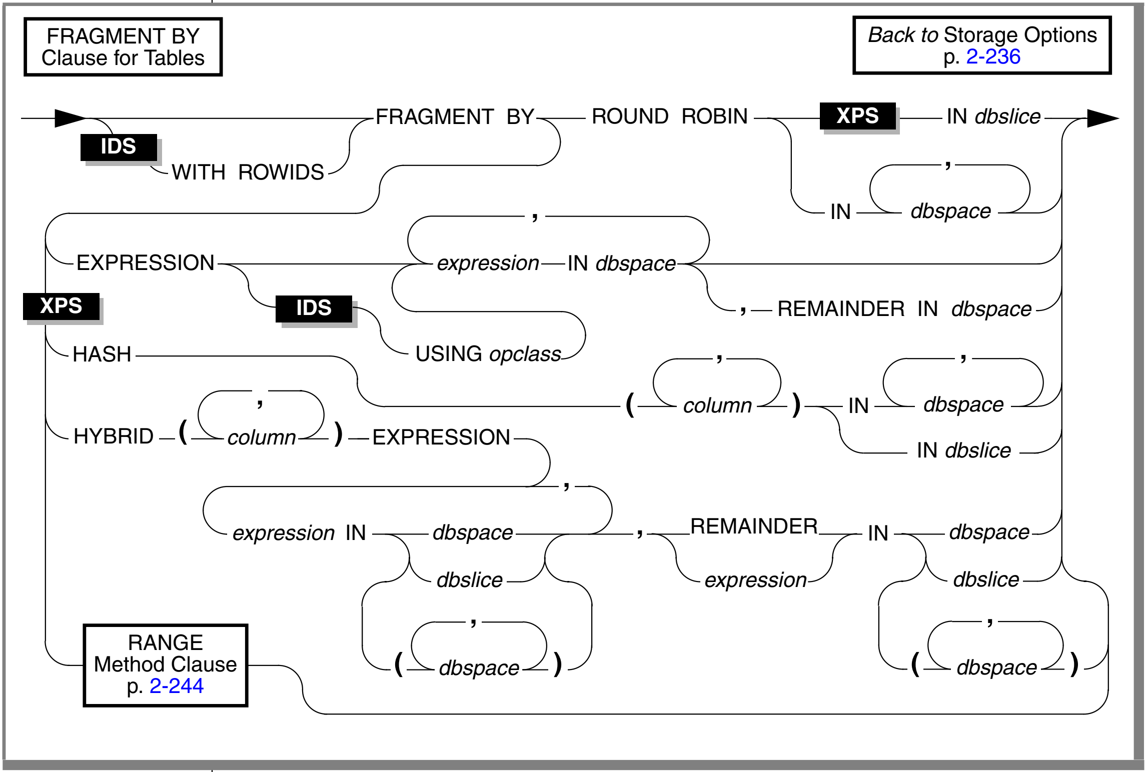
IDS

Storing Data in an extspace

In general, use the extspace storage option in conjunction with the "USING Access-Method Clause" on page 2-252. For more information, refer to the user documentation for your custom-access method.

FRAGMENT BY Clause

Use the FRAGMENT BY clause to create fragmented tables and specify their distribution scheme .



Element	Purpose	Restrictions	Syntax
column	Column to which to apply the fragmentation strategy	Must be a column within the table.	Identifier, p. 4-189
dbslice, *dbspace*	Dbslice or dbspace to store the table fragment	The *dbslice* must be defined. You can specify no more than 2,048 *dbspaces* (but at least 2).	Identifier, p. 4-189
expression	Expression that defines a table fragment using a range, hash, or arbitrary rule	Columns can be from the current table only, and data values can be from only a single row. Value returned must be Boolean (true or false).	Expression, p. 4-67
opclass	No default operator class	Must be defined and must be associated with a B-tree index.	Identifier, p. 4-189

When you fragment a table, the IN keyword introduces the storage space where a table fragment is to be stored.

IDS

Using the WITH ROWIDS Option

Nonfragmented tables contain a hidden column called **rowid**, but by default, fragmented tables have no **rowid** column. You can use the WITH ROWIDS keywords to add the **rowid** column to a fragmented table. Each row is automatically assigned a unique **rowid** value that remains stable for the life of the row, and that the database server can use to find the physical location of the row. Each row requires an additional 4 bytes to store the **rowid**.

Important: *This is a deprecated feature. Use primary keys as an access method rather than the* ***rowid*** *column.*

You cannot use the WITH ROWIDS clause with typed tables.

Fragmenting by ROUND ROBIN

In a round-robin distribution scheme, specify at least two dbspaces where you want the fragments to be placed. As records are inserted into the table, they are placed in the first available dbspace. The database server balances the load between the specified dbspaces as you insert records and distributes the rows in such a way that the fragments always maintain approximately the same number of rows. In this distribution scheme, the database server must scan all fragments when it searches for a row.

XPS

With Extended Parallel Server, you can specify a dbslice to fragment a table across a group of dbspaces that share the same naming convention. For a syntax alternative to `FRAGMENT BY ROUND ROBIN IN` *dbslice* that achieves the same results, see "Storing Data in a dbslice" on page 2-237. ♦

IDS

Use the PUT clause to specify round-robin fragmentation for smart large objects. For more information, see the "PUT Clause" on page 2-249. ♦

Fragmenting by EXPRESSION

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a storage space. Each fragment expression in the rule isolates data and aids the database server in searching for rows.

To fragment a table by expression, specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which rows are placed in a fragment, as the next example shows:

  ```
  FRAGMENT BY EXPRESSION c1 < 100 IN dbsp1,
  c1 >= 100 AND c1 < 200 IN dbsp2, c1 >= 200 IN dbsp3
  ```

- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically uses OR clauses to group data, as the following example shows:

  ```
  FRAGMENT BY EXPRESSION
  zip_num = 95228 OR zip_num = 95443 IN dbsp2,
  zip_num = 91120 OR zip_num = 92310 IN dbsp4,
  REMAINDER IN dbsp5
  ```

Warning: *See the note about the* **DBCENTURY** *environment variable and date values in fragment expressions in the section "Logging Options" on page 2-215.*

IDS

The USING Opclass Option

With the USING option, you can specify a nondefault operator class for the fragmentation strategy. The secondary-access method of the chosen operator class must have a B-tree index structure.

In the following example, the **abs_btree_ops** operator class specifies several user-defined strategy functions that order integers based on their absolute values:

```
CREATE OPCLASS abs_btree_ops FOR btree
   STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte, abs_gt)
   SUPPORT (abs_cmp)
```

For the fragmentation strategy, you can specify the **abs_btree_ops** operator class in the USING clause and use its strategy functions to fragment the table, as follows:

```
FRAGMENT BY EXPRESSION USING abs_btree_ops
   (abs_lt(x,345)) IN dbsp1,
   (abs_gte(x,345) AND abs_lte(x,500)) IN dbsp2,
   (abs_gt(x,500)) IN dbsp3
```

For information on how to create and extend an operator class, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

IDS

User-Defined Functions in Fragment Expressions

For rows that include user-defined data types, you can use comparison conditions or user-defined functions to define the range rules. In the following example, comparison conditions define the range rules for the **long1** column, which contains an opaque data type:

```
FRAGMENT BY EXPRESSION
long1 < '3001' IN dbsp1,
long1 BETWEEN '3001' AND '6000' IN dbsp2,
long1 > '6000' IN dbsp3
```

An implicit, user-defined cast converts `3001` and `6000` to the opaque type.

Alternatively, you can use user-defined functions to define the range rules for the opaque data type of the **long1** column:

```
FRAGMENT BY EXPRESSION
(lessthan(long1,'3001')) IN dbsp1,
(greaterthanorequal(long1,'3001') AND
lessthanorequal(long,'6000')) IN dbsp2,
(greaterthan(long1,'6000')) IN dbsp3
```

Explicit user-defined functions require parentheses around the entire fragment expression before the IN clause, as the previous example shows.

User-defined functions in a fragment expression can be written in SPL or in the C or Java language. These functions must satisfy four requirements:

- They must evaluate to a Boolean value.
- They must be nonvariant.
- They must reside within the same database as the table.
- They must not generate OUT parameters.

For information on how to create user-defined functions for fragment expressions, refer to *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Using the REMAINDER Keyword

Use the REMAINDER keyword to specify the storage space in which to store valid values that fall outside the specified expression or expressions.

If you do not specify a remainder, and a row is inserted or updated such that it no longer belongs to any dbspace, the database server returns an error.

XPS

Fragmenting by HASH

A hash-distribution scheme distributes the rows as you insert them, so that the fragments maintain approximately the same number of rows. In this distribution scheme, the database server can eliminate fragments when it searches for a row because the hash is known internally. For example, if you have a large database, as in a data-warehousing environment, you can fragment your tables across disks that belong to different coservers. If you expect to perform many queries that scan most of the data, a system-defined hash-distribution scheme can balance the I/O processing. The following example uses eight coservers with one dbspace defined on each coserver.

```
CREATE TABLE customer
   (
    cust_id integer,
    descr char(45),
    level char(15),
    sale_type char(10),
    channel char(30),
    corp char(45),
    cust char(45),
    vert_mkt char(30),
    state_prov char(20),
    country char(15),
    org_cust_id char(20)
)
FRAGMENT BY HASH (cust_id) IN
      customer1_spc,
      customer2_spc,
      customer3_spc,
      customer4_spc,
      customer5_spc,
      customer6_spc,
      customer7_spc,
      customer8_spc
EXTENT SIZE 20 NEXT SIZE 16
```

You can also specify a *dbslice*. When you specify a dbslice, the database server fragments the table across the dbspaces that make up the dbslice.

Serial Columns in HASH-Distribution Schemes

If you base table fragmentation on a SERIAL or SERIAL8 column, only a hash-distribution scheme is valid. In addition, the serial column must be the only column in the hashing key. (These restrictions apply only to *table* distributions. Fragmentation schemes for *indexes* that are based on SERIAL or SERIAL8 columns are not subject to these restrictions.)

The following excerpt is from a CREATE TABLE statement:

```
CREATE TABLE customer
   (
    cust_id serial,
   . . .
)
FRAGMENT BY HASH (cust_id) IN customer1_spc, customer2_spc
```

You might notice a difference between serial-column values in fragmented and nonfragmented tables. The database server assigns serial values round-robin across fragments, so a fragment might contain values from noncontiguous ranges. For example, if there are two fragments, the first serial value is placed in the first fragment, the second serial value is placed in the second fragment, the third value is placed in the first fragment, and so on.

XPS

Fragmenting by HYBRID

The HYBRID clause allows you to apply two distribution schemes to the same table. You can use a combination of hash- and expression-distribution schemes or a combination of range-distribution schemes on a table. This section discusses the hash and expression form of hybrid fragmentation. For details of range fragmentation, see "RANGE Method Clause" on page 2-244.

In hybrid fragmentation, the EXPRESSION clause determines the base fragmentation strategy of the table, associating an expression with a set of dbspaces (dbspace, dbslice, or dbspacelist format) for data storage. The hash column(s) determines the dbspace within the specified set of dbspaces.

When you specify a dbslice, the database server fragments the table across the dbspaces that make up the dbslice. Similarly, if you specify a dbspace list, the database server fragments the table across the dbspaces in that list. In the next example, **my_hybrid**, distributes rows based on two columns of the table. The value of **col1** determines in which dbslice the row belongs.

The hash value of **col2** determines in which dbspace (within the previously determined dbslice) to insert into.

```
CREATE TABLE my_hybrid
      (col1 INT, col2 DATE, col3 CHAR(10))
   HYBRID (col2) EXPRESSION col1 < 100 IN dbslice1,
      col1 >= 100 and col1 < 200 IN dbslice2,REMAINDER IN dbslice3
```

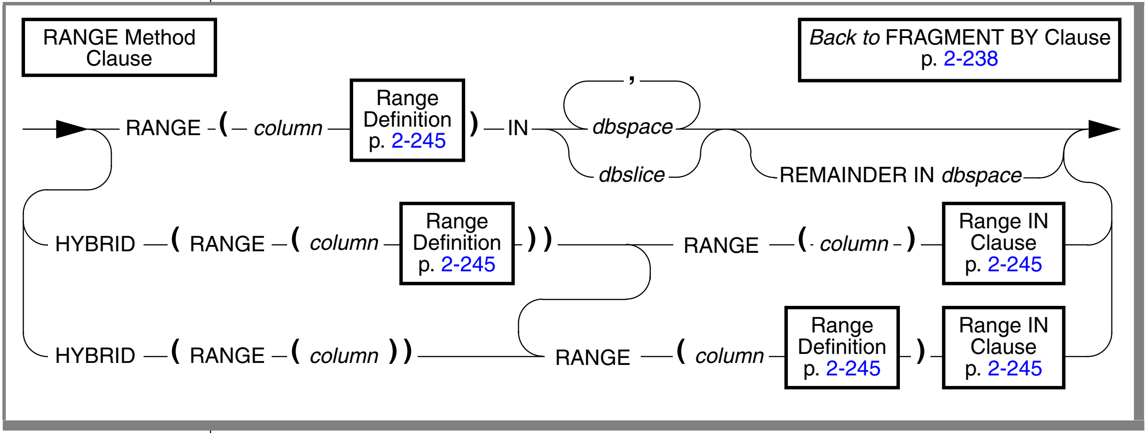
For more information on an expression-based distribution scheme, see "Fragmenting by EXPRESSION" on page 2-239.

XPS

RANGE Method Clause

You can use a range-fragmentation method as a convenient alternative to fragmenting by the EXPRESSION or HYBRID clauses. This provides a method to implicitly and uniformly distribute data whose fragmentation column values are dense or naturally uniform.

In a range-fragmented table, each dbspace stores a contiguous, completely bound and non-overlapping range of integer values over one or two columns. In other words, the database server implicitly clusters rows within the fragments, based on the range of the values in the fragmentation column.



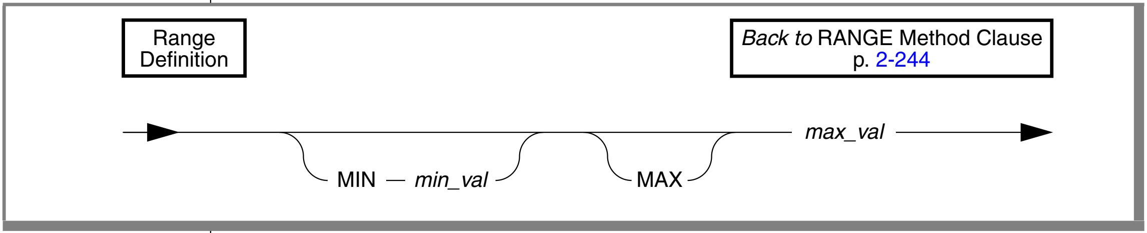
Element	Purpose	Restrictions	Syntax
column	Column on which to apply the fragmentation strategy	Must be in the current table and must be of data type INT or SMALL INT.	Identifier, p. 4-189
dbslice	Dbslice that contains the dbspaces where the table fragments reside	Must exist when you execute the statement.	Identifier, p.4-189
dbspace	Dbspace that contains the table fragment	Must exist when you execute the statement. The maximum number of dbspaces is 2048.	Identifier, p. 4-189

For hybrid strategies with two range definitions, the second *column* must be different column name from the first. For hybrid strategies with exactly one range definition, both occurrences of *column* must specify the same column.

If you list more than one *dbslice*, including a remainder *dbslice*, each dbslice must contain the same number of dbspaces. Unless you are specifying the dbspace in the REMAINDER option, you must specify at least two dbspaces.

Range Definition

Use the range definition to specify the minimum and maximum values of the entire range.

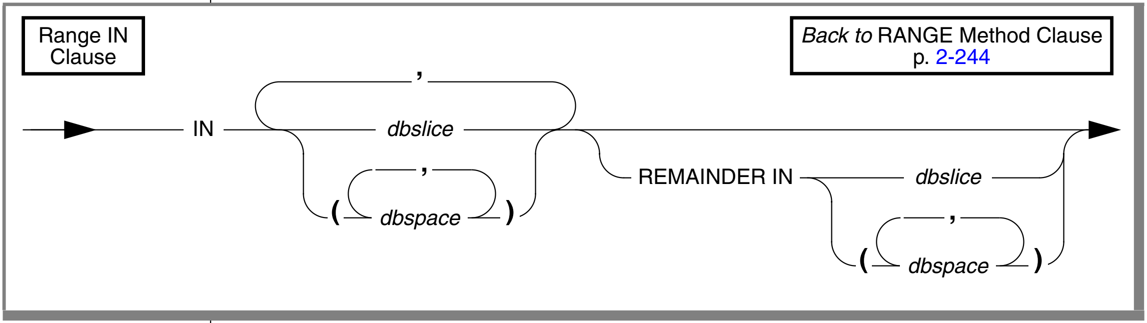


Element	Purpose	Restrictions	Syntax
max_val	Maximum value in the range	Must be an INT or SMALLINT greater than or equal to the *min_val* if *min_val* is supplied.	Literal Number, p. 4-216
min_val	Minimum value in the range; the default is `0`.	Must be an INT or SMALLINT less than or equal to *max_val*.	Literal Number, p. 4-216

You do not need to specify a minimum value. The minimum and maximum values define the exact range of values to allocate for each storage space.

Range IN Clause

Use the IN clause to specify the storage spaces in which to distribute the data.



Element	Purpose	Restrictions	Syntax
dbslice	Dbslice that contains the dbspaces to store table fragments	Must exist.	Identifier, p.4-189
dbspace	Dbspace to store the table fragment	Must exist.	Identifier, p. 4-189

If you specify more than one dbslice, including a remainder dbslice, each dbslice must contain the same number of dbspaces.

Unless you are specifying the dbspace in the REMAINDER option, the minimum number of dbspaces that you can specify is two. The maximum number of dbspaces that you can specify is 2,048.

When you use a range-fragmentation method, the number of integer values between the minimum and maximum specified values must be equal to or greater than the number of storage spaces specified so that the database server can allocate non-overlapping contiguous ranges across the dbspaces. For example, the following code returns an error, because the allocations for the range cannot be distributed across all specified dbspaces:

```
CREATE TABLE Tab1 (Col1 INT...)
   FRAGMENT BY RANGE (Col1 MIN 5 MAX 7)
      IN db1, db2, db3, db4, db5, db6 -- returns an error
```

The error for this example occurs because the specified range contains three values (5, 6, and 7), but six dbspaces are specified; three values cannot be distributed across six dbspaces.

Using the REMAINDER Keyword

Use the REMAINDER keyword to specify the storage space in which to store valid values that fall outside the specified expression or expressions.

If you do not specify a remainder and a row is inserted or updated such that it no longer belongs to any storage space, the database server returns an error.

Restrictions

If you fragment a table with range fragmentation, you cannot perform the following operations on the table after it is created:

- You cannot change the fragmentation strategy (ALTER FRAGMENT).
- You cannot rename the columns of the table (RENAME COLUMN).
- You cannot alter the table in any way except to change the table type or to change the lock mode.

That is, the Usage-TYPE options and the Lock Mode clause are the only valid options of ALTER TABLE for a table that has range fragmentation.

Examples

The following examples illustrate range fragmentation in its simple and hybrid forms.

Simple Range-Fragmentation Strategy

The following example shows a simple range-fragmentation strategy:

```
CREATE TABLE Tab1 (Col1 INT...)
   FRAGMENT BY RANGE (Col1 MIN 100 MAX 200)
      IN db1, db2, db3, db4
```

In this example, the database server fragments the table according to the following allocations.

Storage Space	Holds Values	Storage Space	Holds Values
db1	100 <= Col1 < 125	db3	150 <= Col1 < 175
db2	125 <= Col1 < 150	db4	175 <= Col1 < 200

The previous table shows allocations that can also be made with an expression-based fragmentation scheme:

```
... FRAGMENT BY EXPRESSION
   Col1 >= 100 AND Col1 < 125 IN db1
   Col1 >= 125 AND Col1 < 150 IN db2
   Col1 >= 150 AND Col1 < 175 IN db3
   Col1 >= 175 AND Col1 < 200 IN db4
```

As the examples show, the range-fragmentation example requires much less coding to achieve the same results. The same is true for the hybrid-range fragmentation compared to hybrid-expression fragmentation methods.

Column-Major-Range Allocation

The following example demonstrates the syntax for column-major-range allocation, a hybrid-range fragmentation strategy:

```
CREATE TABLE tab2 (col2 INT, colx char (5))
   FRAGMENT BY HYBRID
      ( RANGE (col2 MIN 100 MAX 220))
      RANGE (col2)
      IN dbsl1, dbsl2, dbsl3
```

This type of fragmentation creates a distribution across dbslices and provides a further subdivision within each dbslice (across the dbspaces in the dbslice) such that when a query specifies a value for col1 (for example, `WHERE col1 = 127`), the query uniquely identifies a dbspace. To take advantage of the additional subdivision, you must specify more than one dbslice.

Row-Major-Range Allocation

The following example demonstrates the syntax for row-major-range allocation, a hybrid-range fragmentation strategy:

```
CREATE TABLE tab3 (col3 INT, colx char (5))
   FRAGMENT BY HYBRID
      ( RANGE (col3) )
      RANGE (col3 MIN 100 MAX 220)
      IN dbsl1, dbsl2, dbsl3
```

This fragmentation strategy is the counterpart to the column-major-range allocation. The advantages and restrictions are equivalent.

Independent-Range Allocation

The following example demonstrates the syntax for an independent-range allocation, a hybrid-range fragmentation strategy:

```
CREATE TABLE tab4 (col4 INT, colx char (5), col5 INT)
   FRAGMENT BY HYBRID
      ( RANGE (col4 MIN 100 MAX 200) )
      RANGE (col5 MIN 500 MAX 800)
      IN dbsl1, dbsl2, dbsl3
```

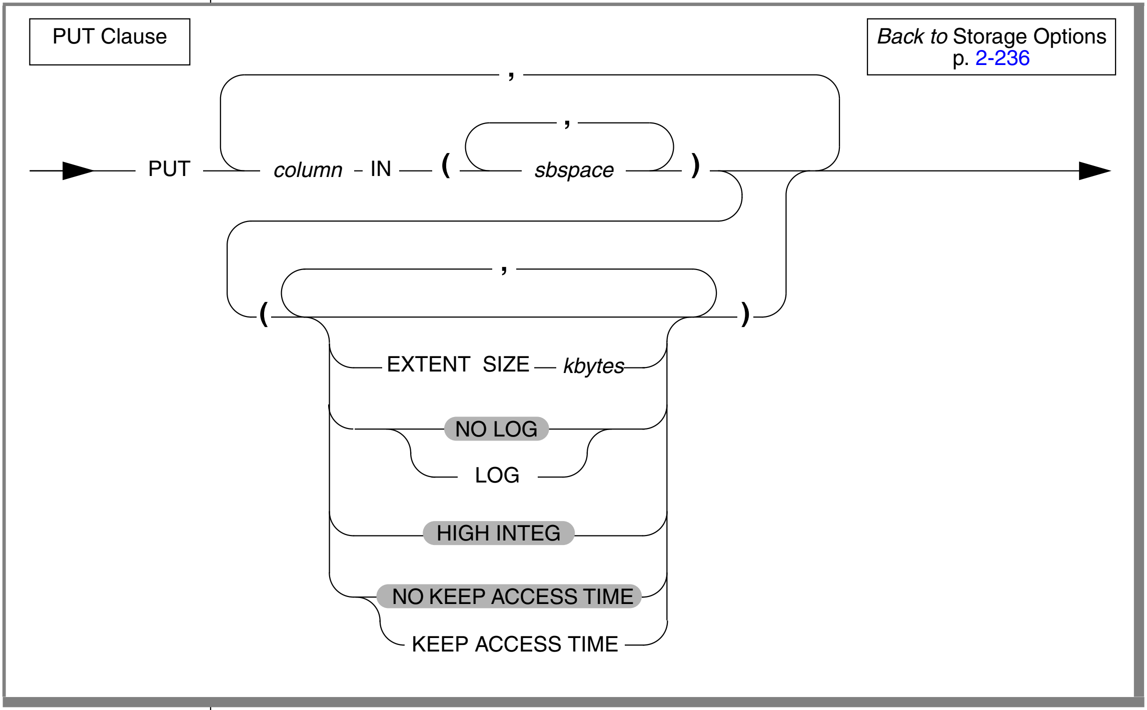
In this type of range fragmentation, the two columns are independent, and therefore the range allocations are independent. The range allocation for a dbspace on both columns is the conjunctive combination of the range allocation on each of the two independent columns.

This type of fragmentation does not provide subdivisions within either column. With this type of fragmentation, a query that specifies values for both columns (such as, `WHERE col4 = 128 and col5 = 650`) uniquely identifies the dbspace at the intersection of the two dbslices identified by the columns independently.

IDS

PUT Clause

Use the PUT clause to specify the storage spaces and characteristics for each column that will contain smart large objects.



Element	Purpose	Restrictions	Syntax
column	Column to store in *sbspace*	Must contain a user-defined, complex, BLOB, or CLOB data type.	Identifier, p. 4-189
kbytes	Number of kilobytes to allocate for the extent size	Must be an integer value.	Literal Number, p. 4-216
sbspace	Name of an area of storage	Must exist.	Identifier, p. 4-189

The column cannot be in the form *column.field*. That is, the smart large object that you are storing cannot be one field of a ROW type.

A smart large object is contained in a single sbspace. The SBSPACENAME configuration parameter specifies the system default in which smart large objects are created unless you specify another area.

Specifying more than one sbspace distributes the smart large objects in a round-robin distribution scheme, so that the number of smart large objects in each space is approximately equal. The **syscolattribs** system catalog table contains one row for each sbspace that you specify in the PUT clause.

When you fragment smart large objects across different sbspaces you can work with smaller sbspaces. If you limit the size of an sbspace, backup and archive operations can perform more quickly. For an example that uses the PUT clause, see "Alternative to Full Logging" on page 2-251.

Six storage options are available to store BLOB and CLOB data:

Option	Purpose
EXTENT SIZE	Specifies how many kilobytes in a smart-large-object extent. The database server might round the EXTENT SIZE up so that the extents are multiples of the sbspace page size.
HIGH INTEG	Produces user-data pages that contain a page header and a page trailer to detect incomplete writes and data corruption. This is the default data-integrity behavior.
KEEP ACCESS TIME	Records, in the smart-large-object metadata, the system time when the smart large object was last read or written.
LOG	Follows the logging procedure used with the current database log for the corresponding smart large object. This option can generate large amounts of log traffic and increase the risk of filling the logical log. (See also "Alternative to Full Logging" on page 2-251.)
NO KEEP ACCESS TIME	Does not record the system time when the smart large object was last read or written. This provides better performance than the KEEP ACCESS TIME option, and is the default tracking behavior.
NO LOG	Turns off logging. This option is the default behavior.

If a user-defined or complex data type contains more than one large object, the specified large-object storage options apply to all large objects in the type unless the storage options are overridden when the large object is created.

Important: *The PUT clause does not affect the storage of simple-large-object data types (BYTE and TEXT). For information on how to store BYTE and TEXT data, see "Large-Object Data Types" on page 4-57.*

Alternative to Full Logging

Instead of full logging, you can turn off logging when you load the smart large object initially and then turn logging back on once the object is loaded.

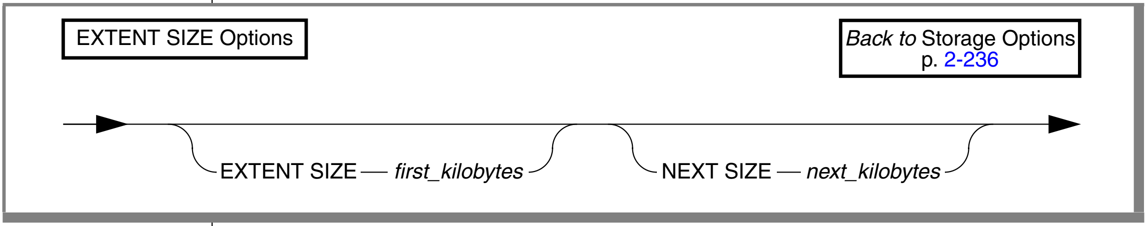
Use the NO LOG option to turn off logging. If you use NO LOG, you can restore the smart-large-object metadata later to a state in which no structural inconsistencies exist. In most cases, no transaction inconsistencies will exist either, but that result is not guaranteed.

The following statement creates the **greek** table. Data values for the table are fragmented into the **dbs1** and **dbs2** dbspaces. The PUT clause assigns the smart-large-object data in the **gamma** and **delta** columns to the **sb1** and **sb2** sbspaces, respectively. The TEXT data values in the **eps** column are assigned to the **blb1** blobspace.

```
CREATE TABLE greek
(alpha INTEGER,
 beta  VARCHAR(150),
 gamma CLOB,
 delta BLOB,
 eps   TEXT IN blb1)
   FRAGMENT BY EXPRESSION
   alpha <= 5 IN dbs1, alpha > 5 IN dbs2
   PUT gamma IN (sb1), delta IN (sb2)
```

EXTENT SIZE Options

The EXTENT SIZE options can define the size of extents assigned to the table.


EXTENT SIZE Options
Back to Storage Options
p. 2-236
EXTENT SIZE — *first_kilobytes*
NEXT SIZE — *next_kilobytes*

Element	**Purpose**	**Restrictions**	**Syntax**
first_kilobytes	Length in kilobytes of the first extent for the table; default is 16.	Must return a positive number; maximum is the chunk size.	Expression, p.4-67
next_kilobytes	Length in kilobytes of each subsequent extent; default is 16.	Must return a positive number; maximum is the chunk size.	Expression, p.4-67

The minimum length of *first_kilobytes* (and of *next_kilobytes*) is four times the disk-page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes.

The next example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

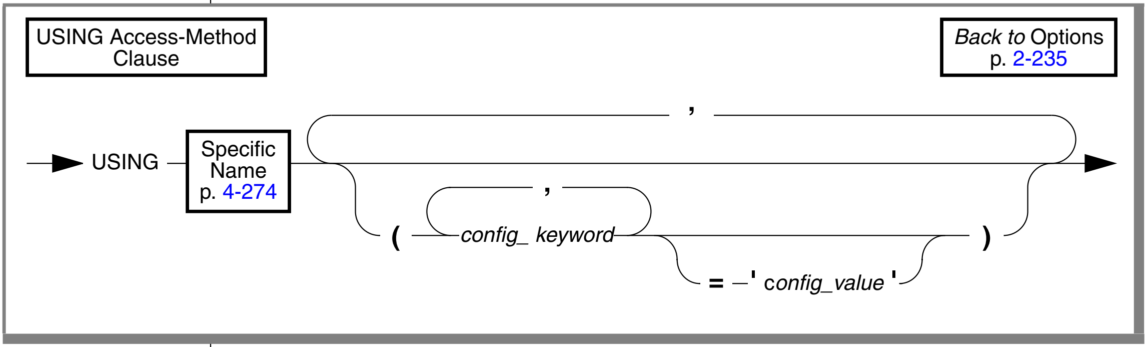
```
CREATE TABLE emp_info
   (
   f_name     CHAR(20),
   l_name     CHAR(20),
   position   CHAR(20),
   start_date DATETIME YEAR TO DAY,
   comments   VARCHAR(255)
   )
EXTENT SIZE 20
```

If you need to revise the extent sizes of a table, you can modify the extent and next-extent sizes in the generated schema files of an unloaded table. For example, to make a database more efficient, you might unload a table, modify the extent sizes in the schema files, and then create and load a new table. For information about how to optimize extents, see your *Administrator's Guide*.

IDS

USING Access-Method Clause

The USING Access Method clause can specify an access method.



Element	Purpose	Restrictions	Syntax
config_keyword	Configuration keyword associated with the specified access method	No more than 18 bytes. The access method must exist.	Literal keyword
config_value	Value of the specified configuration keyword	No more than 236 bytes. Must be defined by the access method.	Quoted String, p. 4-243

A primary-access method is a set of routines that perform all of the operations you need to make a table available to a database server, such as create, drop, insert, delete, update, and scan. The database server provides a built-in primary-access method.

You store and manage a virtual table either outside of the database server in an extspace or inside the database server in an sbspace. (See "Storage Options" on page 2-236.) You can access a virtual table with SQL statements. Access to a virtual table requires a user-defined primary-access method.

DataBlade modules can provide other primary-access methods to access virtual tables. When you access a virtual table, the database server calls the routines associated with that access method rather than the built-in table routines. For more information on these other primary-access methods, refer to your access-method documentation.

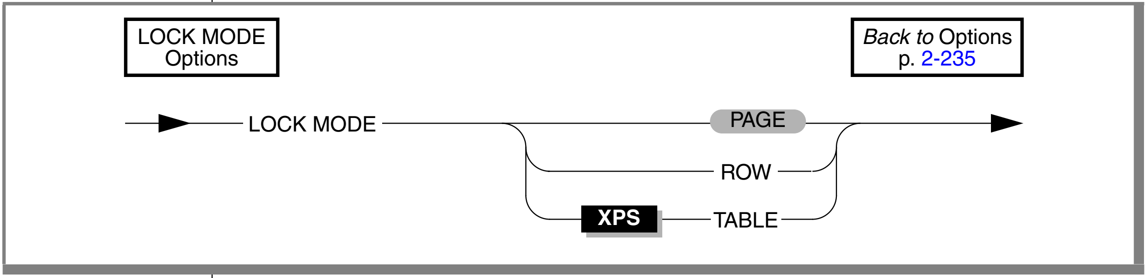
You can retrieve a list of configuration values for an access method from a table descriptor (**mi_am_table_desc**) using the MI_TAB_AMPARAM macro. Not all keywords require configuration values.

The access method must already exist. For example, if an access method called **textfile** exists, you can specify it with the following syntax:

```
CREATE TABLE mybook
   (... )
   IN myextspace
   USING textfile (DELIMITER=':')
```

LOCK MODE Options

Use the LOCK MODE options to specify the locking granularity of the table.



You can subsequently change the lock mode of the table with the ALTER TABLE statement.

The following table describes the locking-granularity options available.

Granularity	Effect
PAGE	Obtains and releases one lock on a whole page of rows This is the default locking granularity. Page-level locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all the rows. For example, if you are processing the contents of a table in the same order as its cluster index, page locking is appropriate.
ROW	Obtains and releases one lock per row Row-level locking provides the highest level of concurrency. If you are using many rows at one time, however, the lock-management overhead can become significant. You can also exceed the maximum number of locks available, depending on the configuration of your database server.
TABLE (XPS only)	Places a lock on the entire table This type of lock reduces update concurrency compared to row and page locks. A table lock reduces the lock-management overhead for the table With table locking, multiple read-only transactions can still access the table.

IDS

Precedence and Default Behavior

In Dynamic Server, you do not have to specify the lock mode each time you create a new table. You can globally set the locking granularity of all new tables in the following environments:

- Database session of an individual user

  You can set the **IFX_DEF_TABLE_LOCKMODE** environment variable to specify the lock mode of new tables during your current session.

- Database server (all sessions on the database server)

  If you are a DBA, you can set the DEF_TABLE_LOCKMODE configuration parameter in the ONCONFIG file to determine the lock mode of all new tables in the database server.

  If you are not a DBA, you can set the **IFX_DEF_TABLE_LOCKMODE** environment variable for the database server, before you run **oninit**, to determine the lock mode of all new tables in the database server.

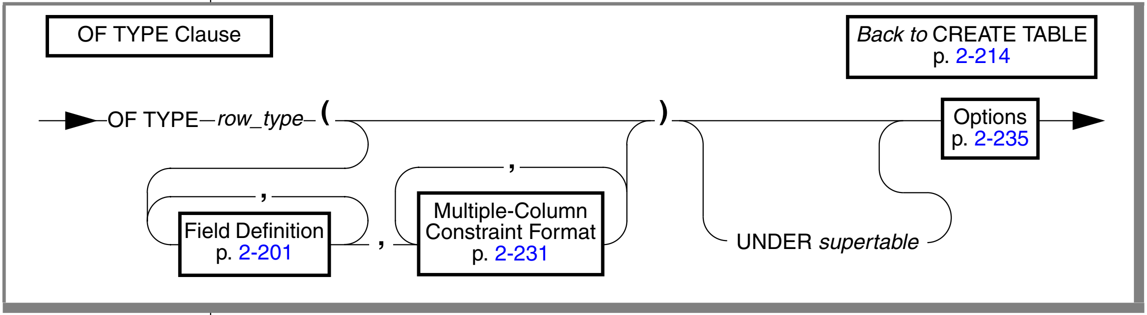
The LOCK MODE setting in a CREATE TABLE statement takes precedence over the settings of the **IFX_DEF_TABLE_LOCKMODE** environment variable and the DEF_TABLE_LOCKMODE configuration parameter.

If CREATE TABLE specifies no LOCK MODE setting, the default mode depends on the setting of the **IFX_DEF_TABLE_LOCKMODE** environment variable or the DEF_TABLE_LOCKMODE configuration parameter. For information about **IFX_DEF_TABLE_LOCKMODE**, refer to the *IBM Informix Guide to SQL: Reference*. For information about the DEF_TABLES_LOCKMODE configuration parameter, refer to the *Administrator's Reference*.

IDS

OF TYPE Clause

Use the OF TYPE clause to create a *typed table* for an object-relational database. A typed table is a table that has a named-row type assigned to it.



Element	Purpose	Restrictions	Syntax
row_type	Name of the row type on which this table is based	Must be a named-row data type that exists in the database.	Data Type, p. 4-49; Identifier, p. 4-189
supertable	Name of the table from which this table inherits its properties	Must already exist as a typed table.	Database Object Name, p. 4-46

If you use the UNDER clause, the *row_type* must be derived from the row type of the *supertable*. A type hierarchy must already exist in which the named-row type of the new table is a subtype of the named-row type of the *supertable*.

Jagged rows are any set rows from a table hierarchy in which the number of columns is not fixed among the typed tables within the hierarchy. Some APIs, such as ESQL/C and JDBC, do not support queries that return jagged rows.

When you create a typed table, the columns of the table are not named in the CREATE TABLE statement. Instead, the columns are specified when you create the row type. The columns of a typed table correspond to the fields of the named-row type. You cannot add additional columns to a typed table.

For example, suppose you create a named-row type, **student_t,** as follows:

```
CREATE ROW TYPE student_t
   (namee       VARCHAR(30),
    average     REAL,
    birthdate   DATETIME YEAR TO DAY)
```

If a table is assigned the type **student_t**, the table is a typed table whose columns are of the same name and data type (and in the same order) as the fields of the type **student_t**. For example, the following CREATE TABLE statement creates a typed table named **students** whose type is **student_t**:

```
CREATE TABLE students OF TYPE student_t
```

The **students** table has the following columns:

```
name      VARCHAR(30)
average   REAL
birthdate DATETIME
```

For more information about ROW types, refer to the CREATE ROW TYPE statement on page 2-198.

Using Large-Object Data in Typed Tables

Use the BLOB or CLOB instead of BYTE or TEXT data types when you create a typed table that contains columns for large objects. For backward compatibility, you can create a named-row type that contains BYTE or TEXT fields and use that type to re-create an existing (untyped) table as a typed table. Although you can use a row type that contains BYTE or TEXT fields to create a typed table, you cannot use such a row type as a column. You can use a row type that contains BLOB or CLOB fields in both typed tables and columns.

Using the UNDER Clause

Use the UNDER clause to specify inheritance (that is, define the table as a subtable). The subtable inherits properties from the supertable which it is under. In addition, you can define new properties specific to the subtable.

Continuing the example shown in "OF TYPE Clause" on page 2-255, the following statements create+ a typed table, **grad_students**, that inherits all of the columns of the **students** table, but also has columns for **adviser** and **field_of_study** that correspond to fields in the **grad_student_t** row type.

```
CREATE ROW TYPE grad_student_t
   (adviser        CHAR(25),
    field_of_study CHAR(40)) UNDER student_t;

CREATE TABLE grad_students OF TYPE grad_student_t  UNDER students;
```

When you use the UNDER clause, the subtable inherits these properties:

- All columns in the supertable
- All constraints defined on the supertable
- All indexes defined on the supertable
- Referential integrity
- The access method
- The storage option (including fragmentation strategy)

  If a subtable defines no fragments, but if its supertable has fragments defined, then the subtable inherits the fragments of the supertable.
- All triggers defined on the supertable

Tip: *Any heritable attributes that are added to a supertable after subtables have been created will automatically be inherited by existing subtables. You do not need to add all heritable attributes to a supertable before you create its subtables.*

Restrictions on Table Hierarchies

Inheritance occurs in one direction only, namely from supertable to subtable. Properties of subtables are *not* inherited by supertables. The section "System Catalog Information" on page 2-259 lists the inherited database objects for which the system catalog maintains no information regarding subtables.

No two tables in a table hierarchy can have the same data type. For example, the final line of the next code example is invalid, because the tables **tab2** and **tab3** cannot have the same row type (**rowtype2**):

```
         create row type rowtype1 (...);
         create row type rowtype2 (...) under rowtype1;
         create table tab1 of type rowtype1;
         create table tab2 of type rowtype2 under tab1;
--Invalid -->create table tab3 of type rowtype2 under tab1;
```

Privileges on Tables

The privileges on a table describe both who can access the information in the table and who can create new tables. For more information about privileges, see "GRANT" on page 2-459.

ANSI

In an ANSI-compliant database, no default table-level privileges exist. You must grant these privileges explicitly. ♦

When set to `yes`, the environment variable **NODEFDAC** prevents default privileges from being granted to PUBLIC on a new table in a database that is not ANSI compliant.

For information about how to prevent privileges from being granted to PUBLIC, see the **NODEFDAC** environment variable in the *IBM Informix Guide to SQL: Reference*. For additional information about privileges, see the *IBM Informix Guide to SQL: Tutorial*.

Default Index Creation Strategy for Constraints

When you create a table with unique or primary-key constraints, the database server creates an internal, unique, ascending index for each constraint.

When you create a table with a referential constraint, the database server creates an internal, nonunique, ascending index for each column specified in the referential constraint.

The database server stores this internal index in the same location that the table uses. If you fragment the table, the database server stores the index fragments in the same dbspaces as the table fragments or in some cases, the database dbspace.

If you require an index fragmentation strategy that is independent of the underlying table fragmentation, do not include the constraint when you create the table. Instead, use the CREATE INDEX statement to create a unique index with the desired fragmentation strategy. Then use the ALTER TABLE statement to add the constraint. The new constraint uses the previously defined index.

Important: *In a database without logging, detached checking is the only kind of constraint checking available. Detached checking means that constraint checking is performed on a row-by-row basis.*

System Catalog Information

When you create a table, the database server adds basic information about the table to the **systables** system catalog table and column information to **syscolumns** system catalog table. The **sysblobs** system catalog table contains information about the location of dbspaces and simple large objects. The **syschunks** table in the **sysmaster** database contains information about the location of smart large objects.

The **systabauth**, **syscolauth**, **sysfragauth**, **sysprocauth**, **sysusers**, and **sysxtdtypeauth** tables contain information about the privileges that various CREATE TABLE options require. The **systables**, **sysxtdtypes**, and **sysinherits** system catalog tables provide information about table types.

Constraints, indexes, and triggers are recorded in the system catalog for the supertable, but not for subtables that inherit them. Fragmentation information, however, is recorded for both supertables and subtables. For more information about inheritance, refer to the *IBM Informix Guide to SQL: Tutorial*.

Related Information

Related statements: ALTER TABLE, CREATE INDEX, CREATE DATABASE, CREATE EXTERNAL TABLE, CREATE ROW TYPE, CREATE Temporary TABLE, DROP TABLE, SET Database Object Mode, and SET Transaction Mode

XPS

See also SET Default Table Type and SET Default Table Space. ♦

For discussions of database and table creation, including discussions on data types, data-integrity constraints, and tables in hierarchies, see the *IBM Informix Database Design and Implementation Guide*.

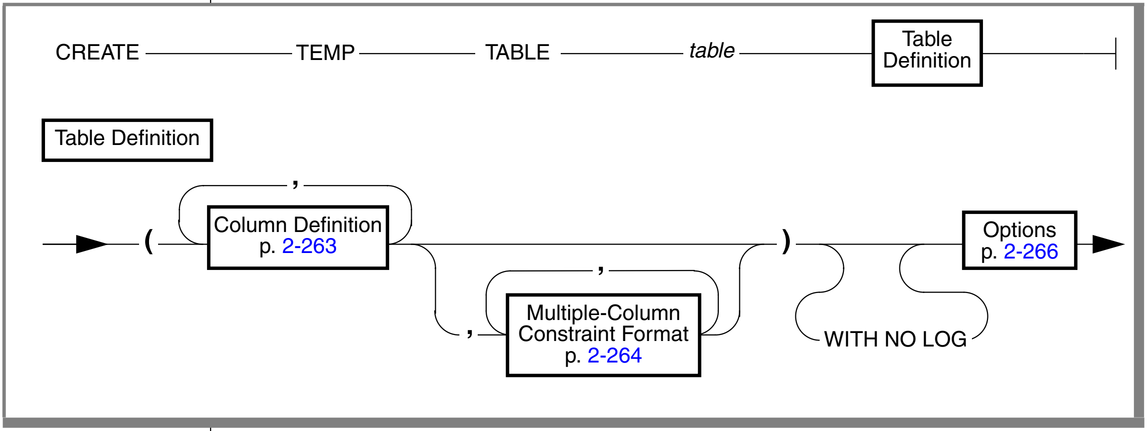
For information about the system catalog tables that store information about objects in the database, see the *IBM Informix Guide to SQL: Reference*.

For information about the **syschunks** table (in the **sysmaster** database) that contains information about the location of smart large objects, see your *Administrator's Reference*.

CREATE TEMP TABLE

Use the CREATE TEMP TABLE statement to create a temporary table in the current database.

Syntax



Element	Purpose	Restrictions	Syntax
table	Name declared here for a table	Must be unique in database.	Database Object Name, p. 4-46

Usage

The CREATE TEMP TABLE statement is a special case of the CREATE Temporary TABLE statement. The CREATE Temporary TABLE statement can also create a SCRATCH table in an Extended Parallel Server database.

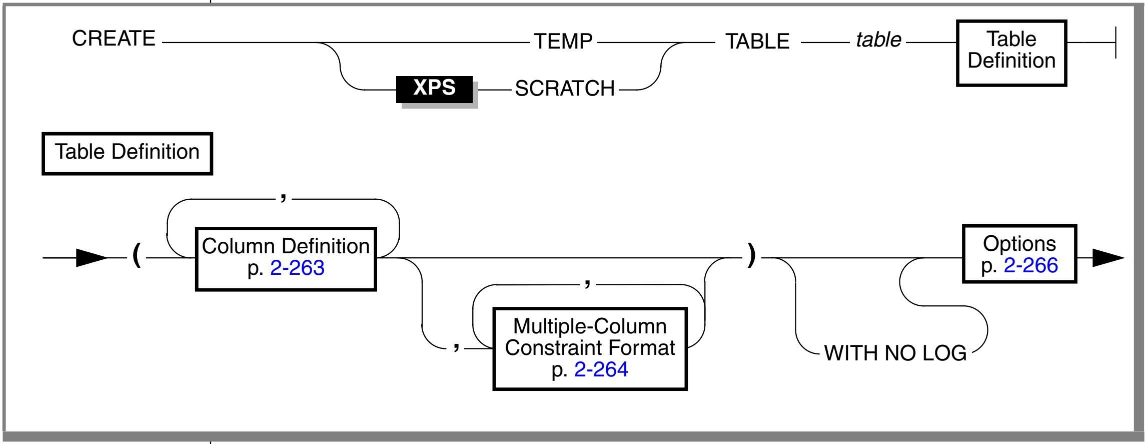
The syntax of the CREATE TEMP TABLE statement is a subset of the syntax that the CREATE TABLE statement supports.

For the complete syntax and semantics of the CREATE TEMP TABLE statement, see "CREATE Temporary TABLE" on page 2-261.

CREATE Temporary TABLE

Use the CREATE Temporary TABLE statement to create a temporary table in the current database.

Syntax



Element	Purpose	Restrictions	Syntax
table	Name declared here for a table	Must be unique in session.	Database Object Name, p. 4-46

Usage

You must have the Connect privilege on the database to create a temporary table. The temporary table is visible only to the user who created it.

DB

In DB-Access, using the CREATE Temporary TABLE statement outside the CREATE SCHEMA statement generates warnings if you set **DBANSIWARN**. ♦

E/C

The CREATE Temporary TABLE statement generates warnings if you use the **-ansi** flag or set the **DBANSIWARN** environment variable. ♦

Using the TEMP Option

Once you create a TEMP table, you can build indexes on the table.

XPS

Using the SCRATCH Option

Use the SCRATCH keyword to reduce the overhead of transaction logging. A scratch table is a nonlogging temporary table that does not support indexes or referential constraints. A scratch table is identical to a TEMP table created with the WITH NO LOG option. Operations on scratch tables are not included in transaction-log operations.

Naming a Temporary Table

A temporary table is associated with a session, not with a database. When you create a temporary table, you cannot create another temporary table with the same name (even for another database) until you drop the first temporary table or end the session. The name must be different from the name of any other table, view, or synonym in the current database, but it need not be different from the temporary table names that are used by other users.

ANSI

In an ANSI-compliant database, the combination *owner.table* must be unique in the database. ♦

Using the WITH NO LOG Option

You should use a SCRATCH table rather than a TEMP…WITH NO LOG table. The behavior of a temporary table that you create with the WITH NO LOG option is the same as that of a SCRATCH table. ♦

Use the WITH NO LOG option to reduce the overhead of transaction logging. If you specify WITH NO LOG, operations on the temporary table are not included in the transaction-log operations. The WITH NO LOG option is required on all temporary tables that you create in temporary dbspaces.

IDS

If you use the WITH NO LOG option in a database that does not use logging, the WITH NO LOG keywords are ignored. If your database does not have logging, any table behaves as if the WITH NO LOG option were specified. ♦

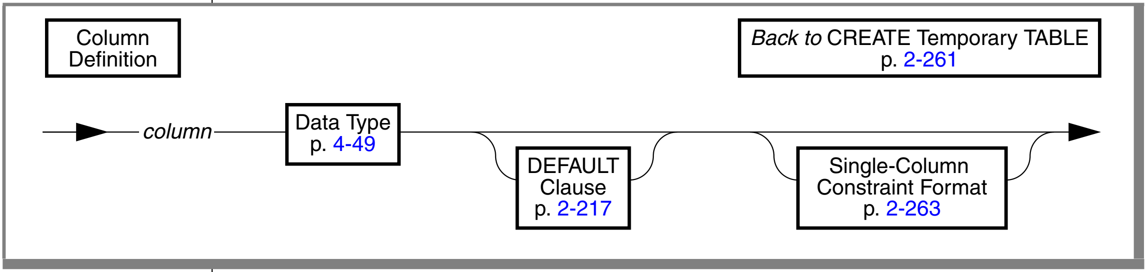
Once you turn off logging on a temporary table, you cannot turn it back on; a temporary table is, therefore, always logged or never logged.

The following temporary table is not logged in a database that uses logging:

```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
   WITH NO LOG
```

Column Definition

Use the column definition portion of CREATE Temporary TABLE to list the name, data type, default value, and constraints of a *single column*.

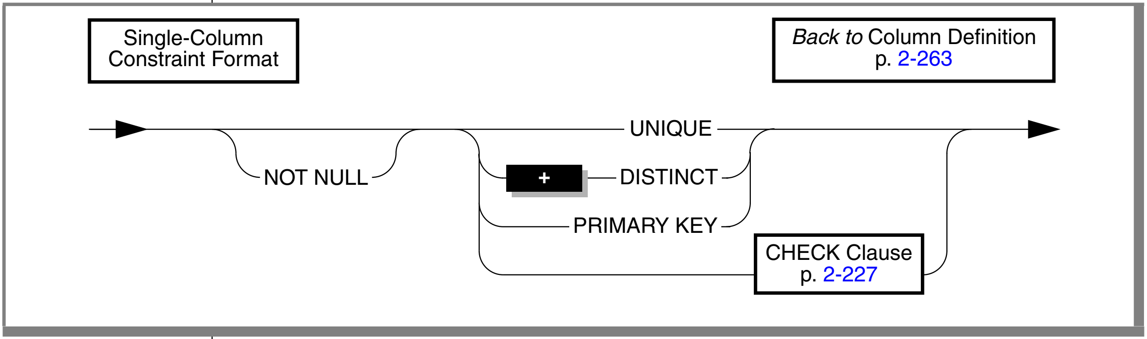

Column Definition
Back to CREATE Temporary TABLE p. 2-261
column
Data Type p. 4-49
DEFAULT Clause p. 2-217
Single-Column Constraint Format p. 2-263

Element	Purpose	Restrictions	Syntax
column	Name of a column in the table	Must be unique in its table.	Identifier, p. 4-189

This portion of the CREATE Temporary TABLE statement is almost identical to the corresponding section in the CREATE TABLE statement. The difference is that fewer types of constraints are allowed in a temporary table.

Single-Column Constraint Format

Use the single-column constraint format to create one or more data-integrity constraints for a single column in a temporary table.


Single-Column Constraint Format
Back to Column Definition p. 2-263
NOT NULL
UNIQUE
+
DISTINCT
PRIMARY KEY
CHECK Clause p. 2-227

This is a subset of the syntax of "Single-Column Constraint Format" on page 2-220 that the CREATE TABLE statement supports.

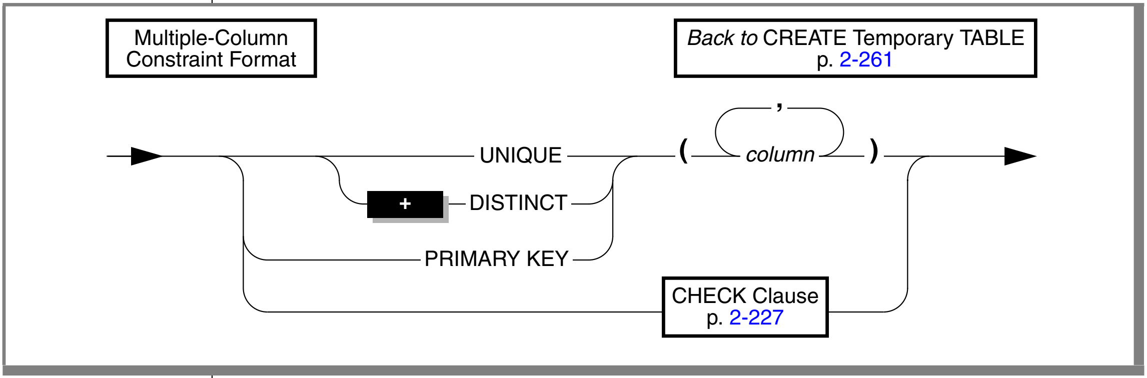
You can find detailed discussions of specific constraints in these sections.

Constraint	For more information, see
CHECK	"CHECK Clause" on page 2-227
DISTINCT	"Using the UNIQUE or DISTINCT Constraints" on page 2-221
NOT NULL	"Using the NOT NULL Constraint" on page 2-221
PRIMARY KEY	"Using the PRIMARY KEY Constraint" on page 2-222
UNIQUE	"Using the UNIQUE or DISTINCT Constraints" on page 2-221

Constraints that you define on temporary tables are always enabled.

Multiple-Column Constraint Format

Use the multiple-column constraint format to associate one or more columns with a constraint. This alternative to the single-column constraint format allows you to associate multiple columns with a constraint.


Multiple-Column Constraint Format
Back to CREATE Temporary TABLE p. 2-261
UNIQUE
+
DISTINCT
PRIMARY KEY
(
,
column
)
CHECK Clause p. 2-227

Element	Purpose	Restrictions	Syntax
column	Name of the column or columns on which the constraint is placed	Must be unique in a table, but the same name can be in different tables of the same database.	Identifier, p. 4-189

This is a subset of the syntax of "Multiple-Column Constraint Format" on page 2-231 that the CREATE TABLE statement supports.

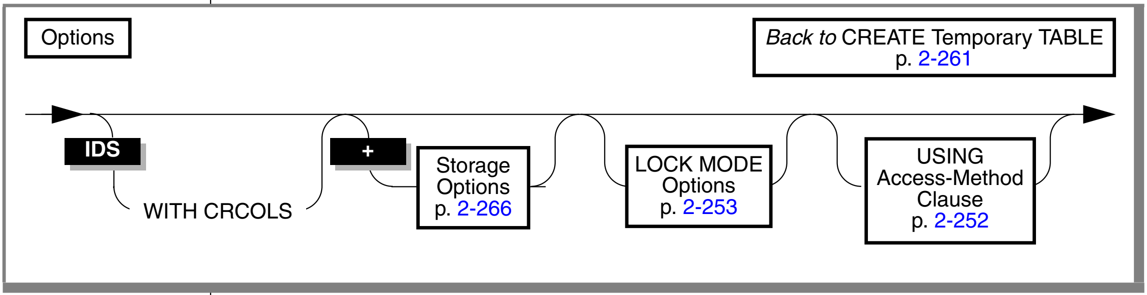
This alternative to the single-column constraint segment of CREATE Temporary TABLE can associate multiple columns with a constraint. Constraints that you define on temporary tables are always enabled.

The following table indicates where you can find detailed discussions of specific constraints.

Constraint	For more information, see	For an example, see
CHECK	"CHECK Clause" on page 2-227	"Defining Check Constraints Across Columns" on page 2-233
DISTINCT	"Using the UNIQUE or DISTINCT Constraints" on page 2-221	"Examples of the Multiple-Column Constraint Format" on page 2-233
PRIMARY KEY	"Using the PRIMARY KEY Constraint" on page 2-222	"Defining Composite Primary and Foreign Keys" on page 2-234
UNIQUE	"Using the UNIQUE or DISTINCT Constraints" on page 2-221	"Examples of the Multiple-Column Constraint Format" on page 2-233

Options

The CREATE Temporary TABLE Options let you specify storage locations, locking modes, and user-defined access methods. You cannot specify initial and next extents for a temporary table. Extents for a temporary table are always eight pages.


Options
Back to CREATE Temporary TABLE p. 2-261
IDS
WITH CRCOLS
+
Storage Options p. 2-266
LOCK MODE Options p. 2-253
USING Access-Method Clause p. 2-252

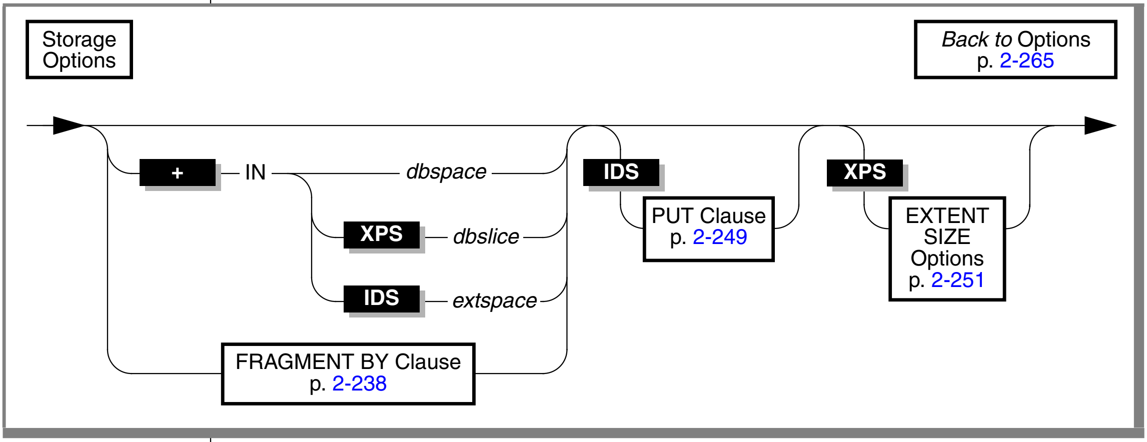
This is a subset of the syntax of "Options" on page 2-235 that the CREATE TABLE statement supports.

Storage Options

Use the storage-option portion of the CREATE Temporary Table statement to specify the distribution scheme for the table.

XPS

If you are using Extended Parallel Server, you can fragment a temporary table across multiple dbspaces that different coservers manage. ♦



Element	Purpose	Restrictions	Syntax
dbspace	Dbspace in which to store the table. Default is the dbspace that stores the current database	Must already exist	Identifier, p. 4-189
dbslice	Name of the dbslice in which to store the table	Must already exist	Identifier, p. 4-189
extspace	Name that **onspaces** assigned to a storage area outside the database server	Must already exist	See documentation for access method.

To create a fragmented, unique index on a temporary table, you must specify an explicit expression-based distribution scheme for a temporary table in the CREATE Temporary TABLE statement.

Where Temporary Tables are Stored

The distribution scheme that you specify with the CREATE Temporary TABLE statement (either with the IN clause or the FRAGMENT BY clause) takes precedence over the information that the **DBSPACETEMP** environment variable and the DBSPACETEMP configuration parameter specify.

For temporary tables for which you do not specify an explicit distribution scheme, each temporary table that you create round-robins to a dbspace that the **DBSPACETEMP** environment variable or the DBSPACETEMP configuration parameter specifies if the environment variable is not set.

For example, if you create three temporary tables, the first one goes into the dbspace called **tempspc1**, the second one goes into **tempspc2**, and the third one goes into **tempspc3**.

This behavior also applies to temporary tables that you create with SELECT...INTO TEMP or SELECT...INTO SCRATCH.

For more information on the **DBSPACETEMP** environment variable, see the *IBM Informix Guide to SQL: Reference*. For more information on the DBSPACETEMP configuration parameter, see your *Administrator's Reference*.

The following example shows how to insert data into a temporary table called **result_tmp** to output to a file the results of a user-defined function (**f_one**) that returns multiple rows:

```
CREATE TEMP TABLE result_tmp( ... );
INSERT INTO result_tmp EXECUTE FUNCTION f_one();
UNLOAD TO 'file' SELECT * FROM temp1;
```

XPS

In Extended Parallel Server, to re-create this example, use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

Differences between Temporary and Permanent Tables

Compared to permanent tables, temporary tables differ in these ways:

- They have fewer types of constraints available.
- They have fewer options that you can specify.
- They are not preserved.

  For more information, see "Duration of Temporary Tables" on page 2-268.
- They are not visible to other users or sessions.
- They do not appear in the system catalogs.

XPS

You can use the following data definition statements on a temporary table from a secondary coserver: CREATE Temporary TABLE, CREATE INDEX, CREATE SCHEMA, DROP TABLE, and DROP INDEX. ♦

DB

The INFO statement and the **Info Menu** option of DB-Access cannot reference temporary tables. ♦

Duration of Temporary Tables

The duration of a temporary table depends on whether or not it is logged.

A logged temporary table exists until one of the following situations occurs:

- The application disconnects.
- A DROP TABLE statement is issued on the temporary table.
- The database is closed.

When any of these events occur, the temporary table is deleted.

Nonlogging temporary tables include tables that were created using the WITH NO LOG option of CREATE TEMP TABLE and all SCRATCH tables.

A nonlogging temporary table exists until one of the following events occurs:

- The application disconnects.
- A DROP TABLE statement is issued on the temporary table.

Because these tables do not disappear when the database is closed, you can use a nonlogging temporary table to transfer data from one database to another while the application remains connected.

Related Information

Related statements: ALTER TABLE, CREATE TABLE, CREATE DATABASE, DROP TABLE, and SELECT

XPS

See also SET Default Table Type and SET Default Table Space. ♦

For additional information about the **DBANSIWARN** and **DBSPACETEMP** environment variables, refer to the *IBM Informix Guide to SQL: Reference*.

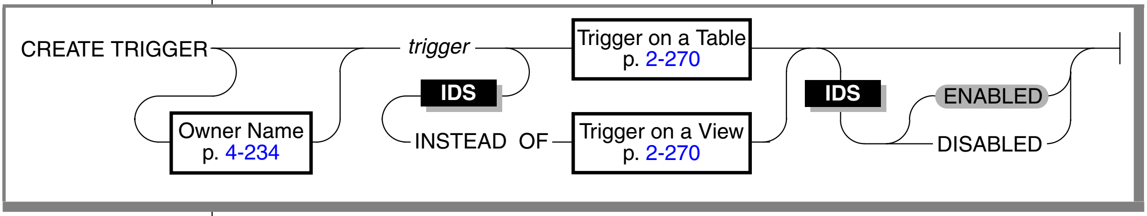
For additional information about the ONCONFIG parameter DBSPACETEMP, see your *Administrator's Guide*.

+

CREATE TRIGGER

Use the CREATE TRIGGER statement to define a trigger on a table or on a view.

Syntax



Element	Purpose	Restrictions	Syntax
trigger	Name that you declare here for a new trigger	Must be unique among the names of triggers in the current database	Identifier, p. 4-189

Usage

A *trigger*, unless disabled, automatically executes a specified set of SQL statements, called the *trigger action*, when a specified *trigger event* occurs.

The trigger event that initiates the trigger action can be an INSERT, DELETE, UPDATE, or (for triggers on IDS tables only) a SELECT statement. The event must specify the table or view on which the trigger is defined. (SELECT or UPDATE events for triggers on tables can also specify one or more columns.)

You can use the CREATE TRIGGER statement in two distinct ways:

- Define a trigger on a table in the current database.

IDS

- Define an INSTEAD OF trigger on a view in the current database. ♦

Any SQL statement that is an instance of the trigger event is called a *triggering statement*. When the event occurs, triggers defined on tables and triggers defined on views differ in whether the triggering statement is executed:

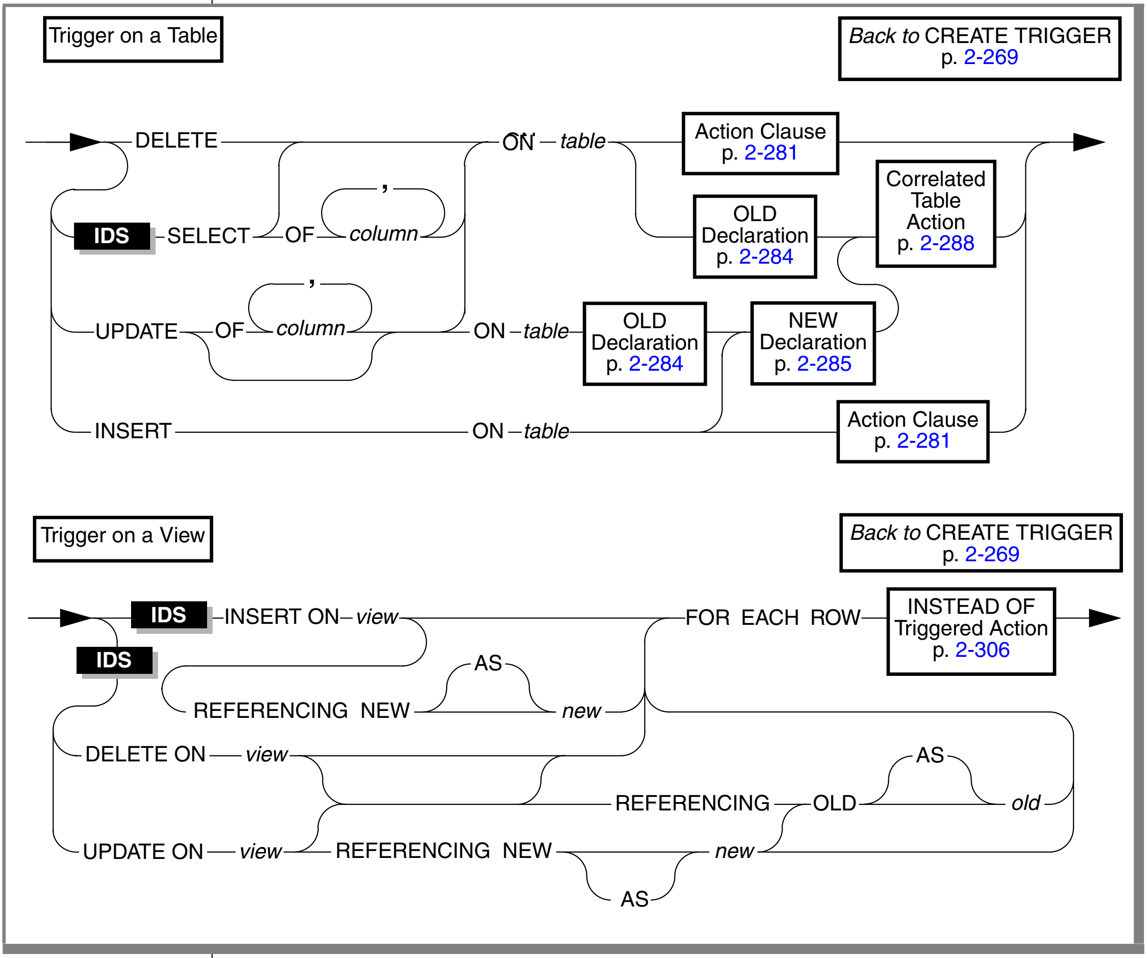
- For tables, the trigger event and the trigger action both execute.

IDS

- For views, only the trigger action executes, instead of the event. ♦

Defining a Trigger Event and Action

This syntax defines the event and action of a trigger on a table or on a view:



Element	Purpose	Restrictions	Syntax
column	The name of a column in the triggering *table*.	Must exist	Identifier, p. 4-189
new, old	Old or new correlation name that you declare here.	Unique in this trigger	Identifier, p. 4-189
table, *view*	Name or synonym of the triggering table or view. The *table* or *view* can include an *owner*. qualifier..	Must exist in the current database	Database Object Name, p. 4-46

The left-hand portion of this diagram (including the *table* or *view*) defines the *trigger event* (sometimes called the *triggering event*). The rest of the diagram declares correlation names and defines the *trigger action* (sometimes called the *triggered action*). (For triggers on tables, see "Action Clause" on page 2-281 and "Correlated Table Action" on page 2-288. For INSTEAD OF triggers on views, see "The Action Clause of INSTEAD OF Triggers" on page 2-306.)

This diagram simplifies the syntax of correlation names in UPDATE events. You can declare either the *old* or the *new* name first, and the REFERENCING keyword is not repeated if both an *old* and a *new* correlation are declared.

Rules for Triggers

To create a trigger on a table or a view, you must own the table or view, or have DBA status. For the relationship between privileges of the trigger owner and of other users, see "Privileges to Execute Trigger Actions" on page 2-298.

The table on which you create a trigger must exist in the current database. You cannot create a trigger on any of the following types of tables:

- a diagnostics table, a violations table, or a table in another database
- a raw table, a temporary table, or a system catalog table

XPS

You cannot create a trigger on a static table nor on a scratch table. When you create a trigger on an operational table, the table cannot use light appends. For more information on light appends, see the *Administrator's Guide*. ♦

IDS

You must observe these rules when you define an INSTEAD OF trigger:

- You can define an INSTEAD OF trigger only on a view, not on a table.
- The view must be local to the current database.
- The view cannot be an updatable view WITH CHECK OPTION.
- No WHEN clause nor SELECT event is valid in an INSTEAD OF trigger,
- No BEFORE or AFTER action is valid in an INSTEAD OF trigger.
- No OF *column* clause is valid in an INSTEAD OF UPDATE trigger.
- Any INSTEAD OF triggered action must specify FOR EACH ROW.
- The triggered action cannot include EXECUTE PROCEDURE INTO.

If multiple tables underly the view, only its owner can create the trigger, but that owner can grant DML privileges on the view to other users. ♦

DB

In DB-Access, if you want to define a trigger as part of a schema, place the CREATE TRIGGER statement inside a CREATE SCHEMA statement. ♦

E/C

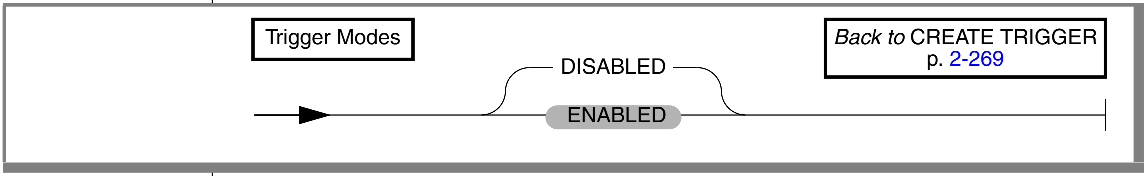
If you are embedding the CREATE TRIGGER statement in an ESQL/C program, you cannot use a host variable in the trigger definition. ♦

You can use the DROP TRIGGER statement to remove an existing trigger. If you use DROP TABLE or DROP VIEW to remove triggering tables or views from the database, all triggers on those tables or views are also dropped.

IDS

Trigger Modes

You can set a trigger mode to enable or disable a trigger when you create it.



You can create triggers on tables or on views in ENABLED or DISABLED mode.

- When a trigger is created in ENABLED mode, the database server executes the trigger action when the trigger event is encountered. (If you specify no mode, ENABLED is the default mode.)
- When a trigger is created in DISABLED mode, the trigger event does not cause execution of the trigger action. In effect, the database server ignores the trigger and its action, even though the **systriggers** system catalog table maintains information about the disabled trigger.

You can use the SET TRIGGERS option of the Database Object Mode statement to set an existing trigger to the ENABLED or DISABLED mode.

After a DISABLED trigger is enabled by the SET TRIGGERS statement, the database server can execute the trigger action when the trigger event is encountered, but the trigger does not perform retroactively. The database server does not attempt to execute the trigger for rows that were inserted, deleted, or updated while the trigger was disabled and before it was enabled.

Warning: *Because the behavior of a trigger varies according to its ENABLED or DISABLED mode, be cautious about disabling a trigger. If disabling a trigger will eventually destroy the semantic integrity of the database, do not disable the trigger.*

Triggers and SPL Routines

You cannot define a trigger in an SPL routine that is called inside a DML (data manipulation language) statement. Thus, the following statement returns an error if the **sp_items** procedure includes the CREATE TRIGGER statement:

```
INSERT INTO items EXECUTE PROCEDURE sp_items
```

(The DML statements are listed in "Data Manipulation Statements" on page 1-10.) SPL variables are not valid in CREATE TRIGGER statements. An SPL routine called by a trigger cannot perform INSERT, DELETE, or UPDATE operations on any table or view that is not local to the current database. See also "Rules for SPL Routines" on page 2-298 for additional restrictions on SPL routines that are invoked in triggered actions.

Trigger Events

The *trigger event* specifies what DML statements can initiate the trigger. The event can be an INSERT, DELETE, or UPDATE operation on the *table* or *view*, or (for IDS tables only) a SELECT operation that manipulates the *table*. You must specify exactly one trigger event. Any SQL statement that is an instance of the trigger event is called a *triggering statement*.

For each *table*, you can define only one trigger that is activated by an INSERT statement and only one trigger that is activated by a DELETE statement. The same table, however, can have multiple triggers that are activated by UPDATE or SELECT statements, provided that each trigger specifies a disjunct set of columns in defining the UPDATE or SELECT event on the table.

IDS

The INSTEAD OF trigger replaces the trigger event with a triggered action. A *view* can have no more than one INSTEAD OF trigger defined for each type of event (INSERT, DELETE, or UPDATE). You can, however, define a trigger on one or more other views, each with its own INSTEAD OF trigger. ♦

You cannot specify a DELETE event if the triggering table has a referential constraint that specifies ON DELETE CASCADE.

You are responsible for guaranteeing that the triggering statement returns the same result with and without the trigger action. See also the sections "Action Clause" on page 2-281 and "Triggered-Action List" on page 2-288.

A triggering statement from an external database server can activate the trigger.

As the following example shows, an insert trigger on **newtab**, managed by **dbserver1**, is activated by an INSERT statement from **dbserver2**. The trigger executes as if the insert originated on **dbserver1**.

```
-- Trigger on stores_demo@dbserver1:newtab
CREATE TRIGGER ins_tr INSERT ON newtab
   REFERENCING new AS post_ins
   FOR EACH ROW(EXECUTE PROCEDURE nt_pct (post_ins.mc));
-- Triggering statement from dbserver2
INSERT INTO stores_demo@dbserver1:newtab
   SELECT item_num, order_num, quantity, stock_num, manu_code,
   total_price FROM items;
```

Trigger Events with Cursors

For triggers on tables, if the triggering statement uses a cursor, each part of the trigger action (including BEFORE, FOR EACH ROW, and AFTER, if these are specified for the trigger) is activated for each row that the cursor processes.

This behavior differs from what occurs when a triggering statement does not use a cursor and updates multiple rows. In this case, any BEFORE and AFTER triggered actions execute only once, but the FOR EACH ROW action list is executed for each row processed by the triggering statement. For additional information about trigger actions, see "Action Clause" on page 2-281.

Privileges on the Trigger Event

You must have appropriate Insert, Delete, Update, or Select privilege on the triggering table or view to execute a triggering INSERT, DELETE, UPDATE, or SELECT statement as the trigger event. The triggering statement might still fail, however, if you do not also have the privileges necessary to execute one of the SQL statements in the trigger action. When the trigger actions are executed, the database server checks your privileges for each SQL statement in the trigger definition, as if the statement were being executed independently of the trigger. For information on the privileges needed to execute the trigger actions, see "Privileges to Execute Trigger Actions" on page 2-298.

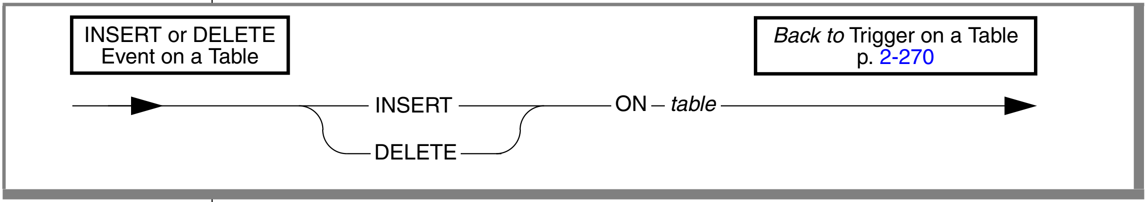
Performance Impact of Triggers

The INSERT, DELETE, UPDATE, and SELECT statements that initiate triggers might appear to execute slowly because they activate additional SQL statements, and the user might not know that other actions are occurring.

The execution time for a trigger event depends on the complexity of the trigger action and whether it initiates other triggers. The time increases as the number of cascading triggers increases. For more information on triggers that initiate other triggers, see "Cascading Triggers" on page 2-300.

INSERT Events and DELETE Events

INSERT and DELETE events on tables are defined by those keywords and by the ON *table* clause, using the following syntax.


INSERT or DELETE Event on a Table
Back to Trigger on a Table p. 2-270
INSERT
DELETE
ON
table

Element	Purpose	Restrictions	Syntax
table	Name of the triggering table	Must exist in the database	Identifier, p. 4-189

An insert trigger is activated when an INSERT statement includes the specified *table* (or a synonym for *table*) in its INTO clause. Similarly, a delete trigger is activated when a DELETE statement includes the specified *table* (or a synonym for *table*) in its FROM clause.

IDS

For triggers on views, the INSTEAD OF keywords must immediately precede the INSERT, DELETE, or UPDATE keyword that specifies the type of trigger event, and the name or synonym of a *view* (rather than of a table) must follow the ON keyword. The section "INSTEAD OF Triggers on Views" on page 2-305 describes s the syntax for defining INSTEAD OF trigger events. ♦

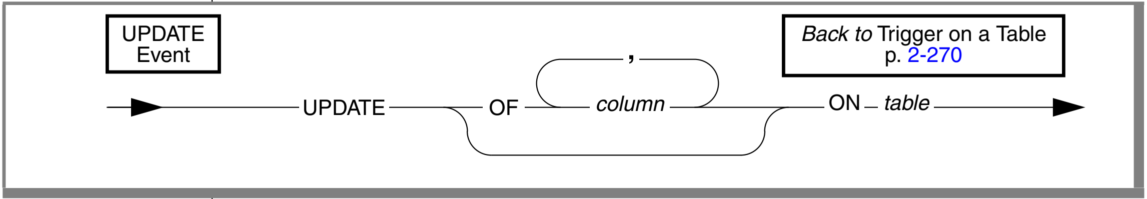
No more than one insert trigger, and no more than one delete trigger, can be defined on the same table.

If you define a trigger on a child table within a table hierarchy, and the child table supports cascading deletes, then a DELETE operation on the parent table activates the delete trigger on the child table.

See also the section "Re-Entrancy of Triggers" on page 2-294 for information about dependencies and restrictions on the actions of insert triggers and delete triggers.

UPDATE Event

UPDATE events (and SELECT events) can include an optional *column* list.



Element	Purpose	Restrictions	Syntax
column	Column that activates the trigger	Must exist in the triggering *table*	Identifier, p. 4-189
table	Name of the triggering table	Must exist in the database	Identifier, p. 4-189

If you define more than one update trigger on the same table, the *column* list is required, and the *column* lists for each trigger must be mutually exclusive. If you omit the OF *column* list, updating any column activates the trigger.

IDS

The OF *column* clause is not valid for an INSTEAD OF trigger on a view. ♦

An UPDATE on the triggering table can activate the trigger in two cases:

- The UPDATE statement references any column in the *column* list.
- The UPDATE event definition has no OF *column* list specification.

Whether it updates one column or more than one column from the *column* list, a triggering UPDATE statement activates the update trigger only once.

Defining Multiple Update Triggers

Multiple update triggers on the same table cannot include the same columns. In the following example, **trig3** is not valid on the **items** table because its column list includes **stock_num**, which is a triggering column in **trig1**.

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
   REFERENCING OLD AS pre NEW AS post
   FOR EACH ROW(EXECUTE PROCEDURE proc1());
CREATE TRIGGER trig2 UPDATE OF manu_code ON items
   BEFORE(EXECUTE PROCEDURE proc2());
-- Illegal trigger: stock_num occurs in trig1
CREATE TRIGGER trig3 UPDATE OF order_num, stock_num ON items
   BEFORE(EXECUTE PROCEDURE proc3());
```

When an UPDATE statement updates multiple columns that have different triggers, the column numbers of the triggering columns determine the order of trigger execution. Execution begins with the smallest triggering column number and proceeds in order to the largest triggering column number. The following example shows that table **taba** has four columns (**a**, **b**, **c**, **d**):

```
CREATE TABLE taba (a int, b int, c int, d int)
```

Define **trig1** as an update on columns **a** and **c**, and define **trig2** as an update on columns **b** and **d**, as the following example shows:

```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
   AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
   AFTER (UPDATE tabb SET z = z + 1);
```

The following example shows a triggering statement for the update trigger:

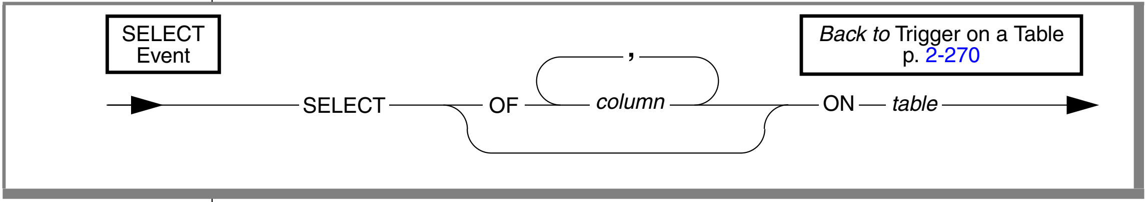
```
UPDATE taba SET (b, c) = (b + 1, c + 1)
```

Then **trig1** for columns **a** and **c** executes first, and **trig2** for columns **b** and **d** executes next. In this case, the smallest column number in the two triggers is column 1 (**a**), and the next is column 2 (**b**).

IDS

SELECT Event

DELETE and INSERT events are defined by those keywords (and the ON *table* clause), but SELECT and UPDATE events also support an optional *column* list.



Element	Purpose	Restrictions	Syntax
column	Column that activates the trigger	Must exist in the triggering *table*	Identifier, p. 4-189
table	Name of the triggering table	Must exist in the database	Identifier, p. 4-189

If you define more than one select trigger on the same table, the *column* list is required, and the *column* lists for each trigger must be mutually exclusive.

A SELECT on the triggering table can activate the trigger in two cases:

- The SELECT statement references any column in the *column* list.
- The SELECT event definition has no OF *column* list specification.

(Sections that follow, however, describe additional circumstances that can affect whether or not a SELECT statement activates a select trigger.)

Whether it specifies one column or more than one column from the *column* list, a triggering SELECT statement activates the select trigger only once.

The action of a select trigger cannot include an UPDATE, INSERT, or DELETE on the triggering table. The action of a select trigger can include UPDATE, INSERT, and DELETE actions on tables other than the triggering table. The following example defines a select trigger on one column of a table:

```
CREATE TRIGGER mytrig
   SELECT OF cola ON mytab REFERENCING OLD AS pre
   FOR EACH ROW (INSERT INTO newtab('for each action'))
```

You cannot specify a SELECT event for an INSTEAD OF trigger on a view.

Circumstances When a Select Trigger is Activated

A query on the triggering table activates a select trigger in these cases:

- The SELECT statement is a standalone SELECT statement.
- The SELECT statement occurs within a UDR called in a select list.
- The SELECT statement is a subquery in a select list.
- The SELECT statement occurs within a UDR called by EXECUTE PROCEDURE or EXECUTE FUNCTION.
- The SELECT statement selects data from a supertable in a table hierarchy. In this case the SELECT statement activates select triggers for the supertable and all the subtables in the hierarchy. ♦

For information on SELECT statements that do not activate a select trigger, see "Circumstances When a Select Trigger is Not Activated" on page 2-280.

Standalone SELECT Statements

A select trigger is activated if the triggering column appears in the select list of the projection clause of a standalone SELECT statement.

For example, if a select trigger is defined to execute whenever column **col1** of table **tab1** is selected, then both of the following standalone SELECT statements activate the select trigger:

```
SELECT * FROM tab1;
SELECT col1 FROM tab1;
```

SELECT Statements Within UDRs in the Select List

A select trigger is activated by a UDR if the UDR contains a SELECT statement within its statement block, and the UDR also appears in the select list of the projection clause of a SELECT statement. For example, assume that a UDR named **my_rtn** contains this SELECT statement in its statement block:

```
SELECT col1 FROM tab1
```

Now suppose that the following SELECT statement invokes the **my_rtn** UDR in its select list:

```
SELECT my_rtn() FROM tab2
```

This SELECT statement activates the select trigger defined on column **col1** of table **tab1** when the **my_rtn** UDR is executed.

UDRs That EXECUTE PROCEDURE and EXECUTE FUNCTION Call

A select trigger is activated by a UDR if the UDR contains a SELECT statement within its statement block and the UDR is called by an EXECUTE PROCEDURE or (for IDS triggers only) the EXECUTE FUNCTION statement. For example, assume that the user-defined procedure named **my_rtn** contains the following SELECT statement in its statement block:

```
SELECT col1 FROM tab1
```

Now suppose that the following statement invokes the **my_rtn** procedure:

```
EXECUTE PROCEDURE my_rtn()
```

This statement activates the select trigger defined on column **col1** of table **tab1** when the SELECT statement within the statement block is executed**.**

Subqueries in the Select List

A select trigger can be activated by a subquery that appears in the select list of the projection clause of a SELECT statement.

For example, if a select trigger was defined on **col1** of **tab1**, the subquery in the following SELECT statement activates that trigger:

```
SELECT (SELECT col1 FROM tab1 WHERE col1=1), colx, col y FROM tabz
```

IDS

Select Triggers in Table Hierarchies

A subtable inherits the select triggers that are defined on its supertable. When you select from a supertable, the SELECT statement activates the select triggers on the supertable and the inherited select triggers on the subtables in the table hierarchy.

For example, assume that table **tab1** is the supertable and table **tab2** is the subtable in a table hierarchy. If the select trigger **trig1** is defined on table **tab1**, a SELECT statement on table **tab1** activates the select trigger **trig1** for the rows in table **tab1** and the inherited select trigger **trig1** for the rows in table **tab2**.

If you add a select trigger to a subtable, this select trigger can override the select trigger that the subtable inherits from its supertable. For example, if the select trigger **trig1** is defined on column **col1** in supertable **tab1**, the subtable **tab2** inherits this trigger. But if you define a select trigger named **trig2** on column **col1** in subtable **tab2**, and a SELECT statement selects from **col1** in supertable **tab1**, this SELECT statement activates trigger **trig1** for the rows in table **tab1** and trigger **trig2** (not trigger **trig1**) for the rows in table **tab2**. In other words, the trigger that you add to the subtable overrides the trigger that the subtable inherits from the supertable.

IDS

Circumstances When a Select Trigger is Not Activated

A SELECT statement on the triggering table does not activate a select trigger in certain circumstances:

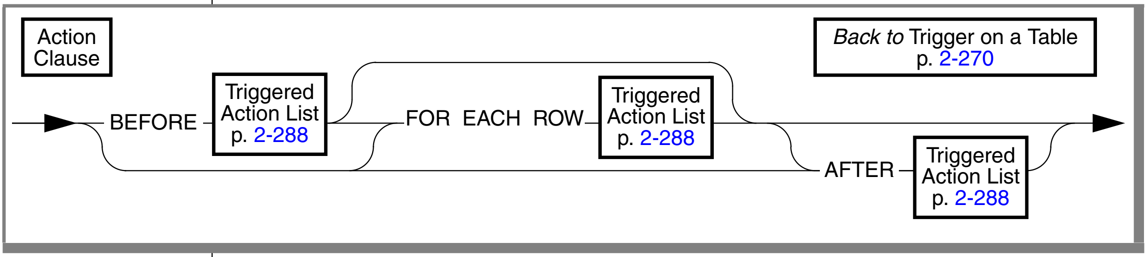
- If a subquery or UDR that contains the triggering SELECT statement appears in any clause of a SELECT statement other than the select list, the select trigger is not activated.

  For example, if the subquery or UDR appears in the WHERE clause or HAVING clause of a SELECT statement, the SELECT statement within the subquery or UDR does not activate the select trigger.
- If the trigger action of a select trigger calls a UDR that includes a triggering SELECT statement, the select trigger on the SELECT in the UDR is not activated. Cascading select triggers are not supported.

- If a SELECT statement contains a built-in aggregate or user-defined aggregate in its select list, the select trigger is not activated. For example, the following SELECT statement does not activate a select trigger defined on **col1** of **tab1**:

  ```
  SELECT MIN(col1) FROM tab1
  ```
- A SELECT statement that includes the UNION or UNION ALL operator does not activate a select trigger.
- The SELECT clause of INSERT does not activate a select trigger.
- If the select list of a SELECT includes the DISTINCT or UNIQUE keywords, the SELECT statement does not activate a select trigger.
- Select triggers are not supported on scroll cursors.
- If a SELECT statement refers to a remote triggering table, the select trigger is not activated on the remote database server.
- Columns in the ORDER BY list of a query activate no select triggers (nor any other triggers) unless also listed in the Projection clause. ♦

Action Clause


Action Clause
Back to Trigger on a Table p. 2-270
BEFORE
Triggered Action List p. 2-288
FOR EACH ROW
Triggered Action List p. 2-288
AFTER
Triggered Action List p. 2-288

The action clause defines trigger actions and can specify when they occur. You must define at least one trigger action, using the keywords BEFORE, FOR EACH ROW, or AFTER to indicate when the action occurs relative to execution of the triggering statement.

You can specify actions for any or all of these three options on a single trigger, but any BEFORE action list must be specified first, and any AFTER action list must be specified last. For more information on the action clause when a REFERENCING clause is also specified, see "Correlated Table Action" on page 2-288.

BEFORE Actions

The list of BEFORE trigger actions execute once before the triggering statement executes. Even if the triggering statement does not process any rows, the database server executes the BEFORE trigger actions.

FOR EACH ROW Actions

After a row of the triggering table is processed, the database server executes all of the statements of the FOR EACH ROW trigger action list; this cycle is repeated for every row that the triggering statement processes. (But if the triggering statement does not insert, delete, update, or select any rows, the database server does not execute the FOR EACH ROW trigger actions.)

XPS

You cannot define FOR EACH ROW actions on tables that have globally-detached indexes. ♦

IDS

The FOR EACH ROW action list of a select trigger is executed once for each instance of a row. For example, the same row can appear more than once in the result of a query joining two tables. For more information on FOR EACH ROW actions, see "Guaranteeing Row-Order Independence" on page 2-283. ♦

AFTER Actions

The specified set of AFTER trigger actions executes once after the action of the triggering statement is complete. If the triggering statement does not process any rows, the AFTER trigger actions still execute.

Actions of Multiple Triggers

When an UPDATE statement activates multiple triggers, the trigger actions merge. Assume that **taba** has columns **a**, **b**, **c**, and **d**, as this example shows:

```
CREATE TABLE taba (a int, b int, c int, d int)
```

Next, assume that you define **trig1** on columns **a** and **c**, and **trig2** on columns **b** and **d**. If both triggers specify BEFORE, FOR EACH ROW, and AFTER actions, then the trigger actions are executed in the following order:

1. BEFORE action list for trigger (**a**, **c**)
2. BEFORE action list for trigger (**b**, **d**)

3. FOR EACH ROW action list for trigger (**a**, **c**)
4. FOR EACH ROW action list for trigger (**b**, **d**)
5. AFTER action list for trigger (**a**, **c**)
6. AFTER action list for trigger (**b**, **d**)

The database server treats all the triggers that are activated by the same triggering statement as a single trigger, and the trigger action is the merged-action list. All the rules that govern a trigger action apply to the merged list as one list, and no distinction is made between the two original triggers.

Guaranteeing Row-Order Independence

In a FOR EACH ROW triggered-action list, the result might depend on the order of the rows being processed. You can ensure that the result is independent of row order by following these suggestions:

- Avoid selecting the triggering table in the FOR EACH ROW section.

  If the triggering statement affects multiple rows in the triggering table, the result of the SELECT statement in the FOR EACH ROW section varies as each row is processed. This condition also applies to any cascading triggers. See "Cascading Triggers" on page 2-300.
- In the FOR EACH ROW section, avoid updating a table with values derived from the current row of the triggering table.

  If the trigger actions modify any row in the table more than once, the final result for that row depends on the order in which rows from the triggering table are processed.
- Avoid modifying a table in the FOR EACH ROW section that is selected by another statement in the same FOR EACH ROW trigger action, including any cascading trigger actions.

If FOR EACH ROW actions modify a table, the changes might not be complete when a subsequent action of the trigger refers to the table. In this case, the result might differ, depending on the order in which rows are processed.

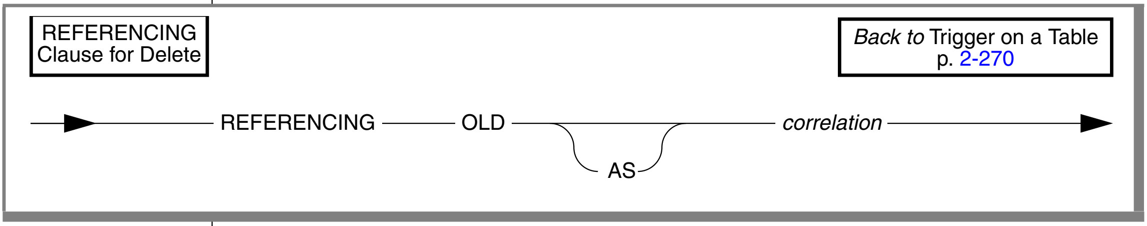
The database server does not enforce rules to prevent these situations because doing so would restrict the set of tables from which a trigger action can select. Furthermore, the result of most trigger actions is independent of row order. Consequently, you are responsible for ensuring that the results of the trigger actions are independent of row order.

REREFERENCING Clauses

The REFERENCING clause for any event declares a correlation name that can be used to qualify column values in the triggering table. They enable FOR EACH ROW actions to reference new values in the result of trigger events.

They also enable FOR EACH ROW actions to reference old column values that existed in the triggering table prior to modification by trigger events.

REFERENCING Clause for Delete


REFERENCING Clause for Delete
Back to Trigger on a Table p. 2-270
REFERENCING
OLD
AS
correlation

Element	Purpose	Restrictions	Syntax
correlation	Name that you declare here for old column value for use within the trigger action	Must be unique within this CREATE TRIGGER statement.	Identifier, p. 4-189

The *correlation* is a qualifier for the column value in the triggering table before the triggering statement executed. The *correlation* is in scope in the FOR EACH ROW trigger action list. See "Correlated Table Action" on page 2-288.

To use a correlation name in a trigger action to refer to an old column value, prefix the column name with the correlation name and a period (**.**) symbol. For example, if the NEW *correlation* name is **post**, refer to the new value for the column **fname** as **post.fname**.

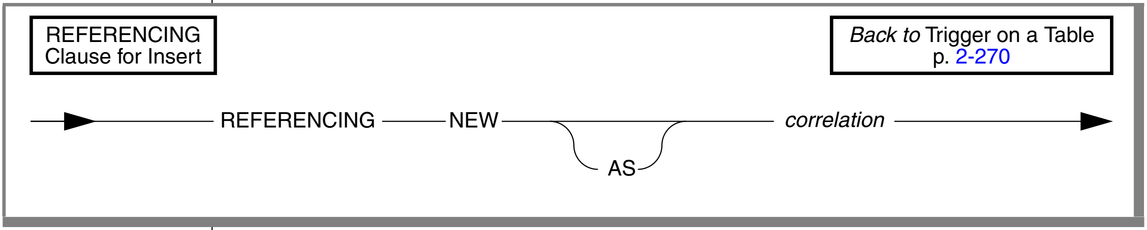
If the trigger event is a DELETE statement, using the NEW *correlation* name as a qualifier causes an error, because the column has no value after the row is deleted. For the rules that govern the use of correlation names, see "Using Correlation Names in Triggered Actions" on page 2-292.

You can use the REFERENCING clause for Delete only if you define a FOR EACH ROW trigger action.

XPS

The OLD *correlation* value cannot be a BYTE or TEXT value. That is, it cannot refer to a BYTE or TEXT column. ♦

REFERENCING Clause for Insert


REFERENCING Clause for Insert
Back to Trigger on a Table p. 2-270
REFERENCING
NEW
AS
correlation

Element	Purpose	Restrictions	Syntax
correlation	Name that you declare here for a new column value for use within the trigger action	Must be unique within this CREATE TRIGGER statement.	Identifier, p. 4-189

The *correlation* is a name for the new column value after the triggering statement has executed. Its scope of reference is only the FOR EACH ROW trigger action list; see "Correlated Table Action" on page 2-288. To use the correlation name, precede the column name with the *correlation* name, followed by a period (**.**) symbol. Thus, if the NEW *correlation* name is **post**, refer to the old value for the column **fname** as **post.fname**.

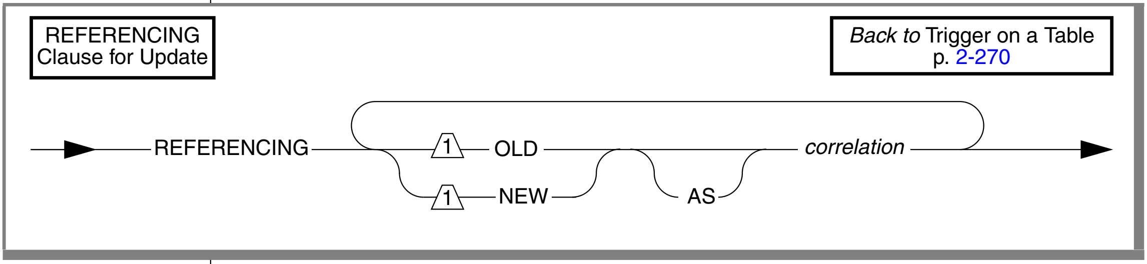
If the trigger event is an INSERT statement, using the OLD *correlation* name as a qualifier causes an error, because no value exists before the row is inserted. For the rules that govern how to use correlation names, see "Using Correlation Names in Triggered Actions" on page 2-292. You can use the INSERT REFERENCING clause only if you define a FOR EACH ROW trigger action.

The following example illustrates use of the INSERT REFERENCING clause. This example inserts a row into **backup_table1** for every row that is inserted into **table1**. The values that are inserted into **col1** and **col2** of **backup_table1** are an exact copy of the values that were just inserted into **table1**.

```
CREATE TABLE table1 (col1 INT, col2 INT);
CREATE TABLE backup_table1 (col1 INT, col2 INT);
CREATE TRIGGER before_trig
   INSERT ON table1 REFERENCING NEW AS new
   FOR EACH ROW
   (
   INSERT INTO backup_table1 (col1, col2)
   VALUES (new.col1, new.col2)
   );
```

As the preceding example shows, the INSERT REFERENCING clause allows you to refer to data values produced by the trigger action.

REFERENCING Clause for Update


REFERENCING Clause for Update
Back to Trigger on a Table p. 2-270
REFERENCING
1
OLD
1
NEW
AS
correlation

Element	Purpose	Restrictions	Syntax
correlation	Name that you declare here for old or new column value for use within the trigger action	Must be unique within this CREATE TRIGGER statement.	Identifier, p. 4-189

The OLD *correlation* is the name of the value of the column in the triggering table before execution of the triggering statement; the NEW *correlation* identifies the corresponding value after the triggering statement executes.

The scope of reference of the *correlation* names that you declare here is only within the FOR EACH ROW trigger action list. See "Correlated Table Action" on page 2-288.

To refer to an old or new column value, prefix the column name with the *correlation* name and a period (**.**) symbol. For example, if the NEW *correlation* name is **post**, you can refer to the new value in column **fname** as **post.fname**.

If the trigger event is an UPDATE statement, you can define both OLD and NEW *correlation* names to refer to column values before and after the triggering UPDATE statement. For rules that govern the use of *correlation* names, see "Using Correlation Names in Triggered Actions" on page 2-292.

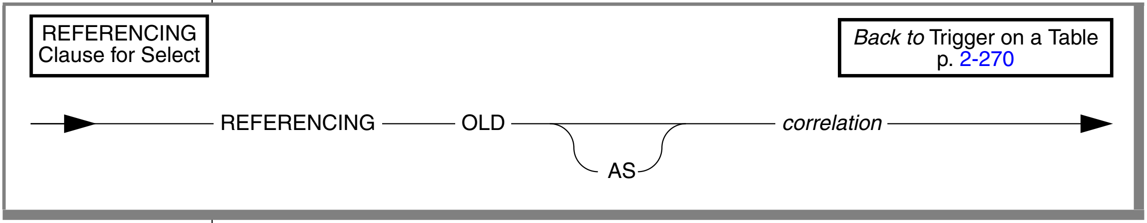
You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW trigger action.

XPS

The OLD *correlation* value cannot be a BYTE or TEXT value. That is, it cannot refer to a BYTE or TEXT column. ♦

IDS

REFERENCING Clause for Select


REFERENCING Clause for Select
Back to Trigger on a Table p. 2-270
REFERENCING
OLD
AS
correlation

Element	Purpose	Restrictions	Syntax
correlation	Name that you declare here for old column value for use within the trigger action	Must be unique within this CREATE TRIGGER statement.	Identifier, p. 4-189

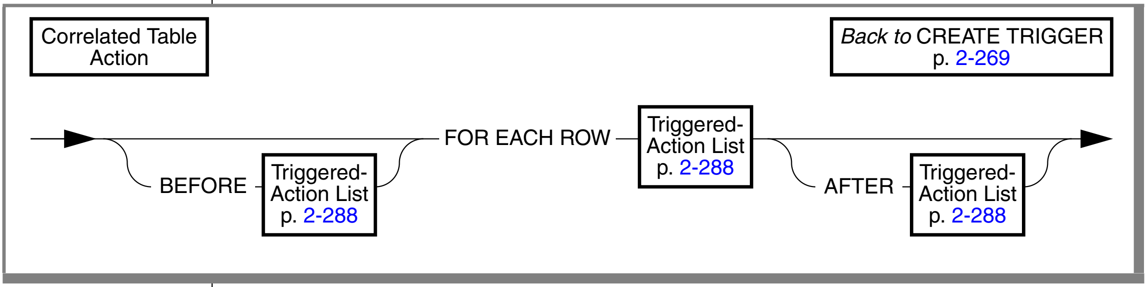
This clause has the same syntax as the "REFERENCING Clause for Delete" on page 2-284. The scope of reference of the *correlation* name that you declare here is only within the FOR EACH ROW trigger action list. See "Correlated Table Action" on page 2-288.

You use the *correlation* name to refer to an OLD column value by preceding the column name with the *correlation* name and a period (**.**) symbol. For example, if the OLD *correlation* name is **pre**, you can refer to the old value for the column **fname** as **pre.fname**.

If the trigger event is a SELECT statement, using the NEW *correlation* name as a qualifier causes an error because the column does not have a NEW value after the column is selected. For the rules that govern the use of correlation names, see "Using Correlation Names in Triggered Actions" on page 2-292.

You can use the SELECT REFERENCING clause only if you define a FOR EACH ROW trigger action.

Correlated Table Action


Correlated Table Action
Back to CREATE TRIGGER p. 2-269
BEFORE
Triggered-Action List p. 2-288
FOR EACH ROW
Triggered-Action List p. 2-288
AFTER
Triggered-Action List p. 2-288

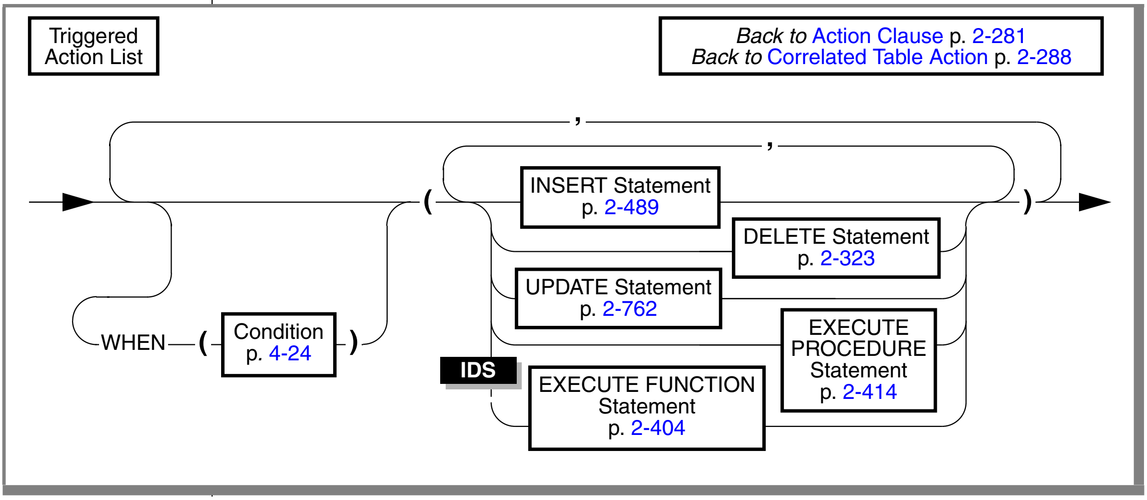
If the CREATE TRIGGER statement contains an INSERT REFERENCING clause, a DELETE REFERENCING clause, an UPDATE REFERENCING clause, or (for Dynamic Server only) a SELECT REFERENCING clause, you must include a FOR EACH ROW triggered-action list in the action clause. You can also include BEFORE and AFTER triggered-action lists, but they are optional.

For information on the BEFORE, FOR EACH ROW, and AFTER triggered-action lists, see "Action Clause" on page 2-281.

XPS

You cannot have FOR EACH ROW actions on tables that have globally-detached indexes. ♦

Triggered-Action List


Triggered Action List
Back to Action Clause p. 2-281
Back to Correlated Table Action p. 2-288
,
,
WHEN
(
Condition p. 4-24
)
(
INSERT Statement p. 2-489
DELETE Statement p. 2-323
UPDATE Statement p. 2-762
EXECUTE PROCEDURE Statement p. 2-414
IDS
EXECUTE FUNCTION Statement p. 2-404
)

For a trigger on a table, the trigger action consists of an optional WHEN condition and the action statements. You can specify a triggered-action list for each WHEN clause, or you can specify a single list (of one or more trigger actions) if you include no WHEN clause.

Database objects that are referenced explicitly in the trigger action or in the definition of the trigger event, such as tables, columns, and UDRs, must exist when the CREATE TRIGGER statement defines the new trigger.

Warning: *When you specify a date expression in the WHEN condition or in an action statement, make sure to specify 4 digits instead of 2 digits for the year. For more about abbreviated years, see the description of* ***DBCENTURY*** *in the "IBM Informix Guide to SQL: Reference," which also describes how the behavior of some database objects can be affected by environment variable settings. Like fragmentation expressions, check constraints, and UDRs, triggers are stored in the system catalog with the creation-time settings of environment variables that can affect the evaluation of expressions like the WHEN(condition).The database server ignores any subsequent changes to those settings when evaluating expressions in those database objects.*

WHEN Condition

The WHEN condition makes the triggered action dependent on the outcome of a test. When you include a WHEN condition in a triggered action, the statements in the triggered action list execute only if the *condition* evaluates to `true`. If the WHEN condition evaluates to `false` or `unknown`, then the statements in the triggered action list are not executed.

If the triggered action is in a FOR EACH ROW section, its *condition* is evaluated for each row. For example, the triggered action in the following trigger executes only if the condition in the WHEN clause is true:

```
CREATE TRIGGER up_price
   UPDATE OF unit_price ON stock
   REFERENCING OLD AS pre NEW AS post
   FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
   (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
      pre.unit_price, post.unit_price, CURRENT))
```

An SPL routine that executes inside the WHEN condition carries the same restrictions as a UDR that is called in a data-manipulation statement.That is, the SPL routine cannot contain certain SQL statements. For information on which statements are restricted, see "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279. ♦

Action Statements

The triggered-action statements can be INSERT, DELETE, UPDATE, EXECUTE FUNCTION, or EXECUTE PROCEDURE statements. If the action list contains multiple statements, and the WHEN *condition* is satisfied (or is absent), then these statements execute in the order in which they appear in the list.

UDRs as Triggered Actions

User-defined functions and procedures can be triggered actions.

IDS

Use the EXECUTE FUNCTION statement to call any user-defined function. Use the EXECUTE PROCEDURE statement to call any user-defined procedure. ♦

XPS

Use the EXECUTE PROCEDURE statement to execute any SPL routine. ♦

For restrictions on using SPL routines as triggered actions, see "Rules for SPL Routines" on page 298 and "Rules for Triggers" on page 2-271.

Achieving a Consistent Result

To guarantee that the triggering statement returns the same result with and without the triggered actions, make sure that the triggered actions in the BEFORE and FOR EACH ROW sections do not modify any table referenced in the following clauses:

- WHERE clause
- SET clause in the UPDATE statement
- SELECT clause
- EXECUTE PROCEDURE clause or EXECUTE FUNCTION clause in a multiple-row INSERT statement

Using Reserved Words

If you use the INSERT, DELETE, UPDATE, or EXECUTE reserved words as an identifier in any of the following clauses inside a triggered action list, you must qualify them by the *owner* name, the *table* name, or both:

- FROM clause of a SELECT statement
- INTO clause of the EXECUTE PROCEDURE or EXECUTE FUNCTION statement

- GROUP BY clause
- SET clause of the UPDATE statement

You get a syntax error if these keywords are *not* qualified when you use these clauses inside a triggered action.

If you use the keyword as a column name, it must be qualified by the table name; for example, **table.update**. If both the table name and the column name are keywords, they must be qualified by the owner name (for example, **owner.insert.update**). If the owner name, table name, and column name are all keywords, the owner name must be in quotes; for example, **'delete'.insert.update**. (These are general rules regarding reserved words as identifiers, rather than special cases for triggers. Your code will be easier to read and to maintain if you avoid using the keywords of SQL as identifiers.)

The only exception is when these keywords are the first table or column name in the list, and you do not have to qualify them. For example, **delete** in the following statement does not need to be qualified because it is the first column listed in the INTO clause:

```
CREATE TRIGGER t1 UPDATE OF b ON tab1
   FOR EACH ROW (EXECUTE PROCEDURE p2() INTO delete, d)
```

The following statements show examples in which you must qualify the column name or the table name:

- FROM clause of a SELECT statement

  ```
  CREATE TRIGGER t1 INSERT ON tab1
     BEFORE (INSERT INTO tab2 SELECT * FROM tab3,
     'owner1'.update)
  ```
- INTO clause of an EXECUTE PROCEDURE statement

  ```
  CREATE TRIGGER t3 UPDATE OF b ON tab1
     FOR EACH ROW (EXECUTE PROCEDURE p2() INTO
     d, tab1.delete)
  ```

  IDS

  (Note that an INSTEAD OF trigger on a view cannot include the EXECUTE PROCEDURE INTO statement among its trigger actions.) ♦
- GROUP BY clause of a SELECT statement

  ```
  CREATE TRIGGER t4 DELETE ON tab1
     BEFORE (INSERT INTO tab3 SELECT deptno, SUM(exp)
     FROM budget GROUP BY deptno, budget.update)
  ```
- SET clause of an UPDATE statement

  ```
  CREATE TRIGGER t2 UPDATE OF a ON tab1
     BEFORE (UPDATE tab2 SET a = 10, tab2.insert = 5)
  ```

Using Correlation Names in Triggered Actions

These rules apply when you use correlation names in triggered actions:

- You can use the correlation names for the old and new column values in SQL statements of the FOR EACH ROW triggered-action list and in the WHEN condition.
- The old and new correlation names refer to all rows affected by the triggering statement.
- You cannot use the correlation name to qualify a column name in the GROUP BY, the SET, or the COUNT DISTINCT clause.
- The scope of reference of the correlation names is the entire trigger definition. This scope is statically determined, meaning that it is limited to the trigger definition; it does not encompass cascading triggers or columns that are qualified by a table name in a UDR that is a triggered action.

When to Use Correlation Names

In SQL statements of the FOR EACH ROW list, you must qualify all references to columns in the triggering table with either the old or new correlation name, unless the statement is valid independent of the triggered action.

In other words, if a column name inside a FOR EACH ROW triggered action list is not qualified by a correlation name, even if it is qualified by the triggering table name, it is interpreted as if the statement is independent of the triggered action. No special effort is made to search the definition of the triggering table for the non-qualified column name.

For example, assume that the following DELETE statement is a triggered action inside the FOR EACH ROW section of a trigger:

```
DELETE FROM tab1 WHERE col_c = col_c2
```

For the statement to be valid, both **col_c** and **col_c2** must be columns from **tab1**. If **col_c2** is intended to be a correlation reference to a column in the triggering table, it must be qualified by either the old or the new correlation name. If **col_c2** is not a column in **tab1** and is not qualified by either the old or new correlation name, you get an error.

In a statement that is valid independent of the triggered action, a column name with no *correlation* qualifier refers to the current value in the database.

In the triggered action for trigger **t1** in the next example, **mgr** in the WHERE clause of the correlated subquery is an unqualified column in the triggering table. In this case, **mgr** refers to the current column value in **empsal** because the INSERT statement is valid independent of the triggered action.

```
CREATE DATABASE db1;
CREATE TABLE empsal (empno INT, salary INT, mgr INT);
CREATE TABLE mgr (eno INT, bonus INT);
CREATE TABLE biggap (empno INT, salary INT, mgr INT);

CREATE TRIGGER t1 UPDATE OF salary ON empsal
AFTER (INSERT INTO biggap SELECT * FROM empsal WHERE salary <
   (SELECT bonus FROM mgr WHERE eno = mgr));
```

In a triggered action, an unqualified column name from the triggering table refers to the current column value, but only when the triggered statement is valid independent of the triggered action.

Qualified Versus Unqualified Value

This table summarizes the value retrieved when the column name is qualified by the old OR BY THE NEW correlation name.

Trigger Event	old.col	new.col
INSERT	No value (error)	Inserted value
UPDATE (column updated)	Original value	Current value (N)
UPDATE (column not updated)	Original value	Current value (U)
DELETE	Original value	No value (error)

Refer to the following key when you read the previous table.

Term	Meaning
Original value	Value before the triggering statement
Current value	Value after the triggering statement
(N)	Cannot be changed by triggered action
(U)	Can be updated by triggered statements; value might be different from original value because of preceding triggered actions

Outside a FOR EACH ROW triggered-action list, you cannot qualify a column from the triggering table with either the old correlation name or the new correlation name; it always refers to the current value in the database.

IDS

Statements in the trigger action list use whatever collating order was in effect when the trigger was created, even if a diferent collation is in effect when the trigger action is executed. See **SET COLLATION** for details of how to specify a collating order different from what **DB_LOCALE** specifies. ♦

Re-Entrancy of Triggers

In some cases a trigger can be re-entrant. In these cases the triggered action can reference the triggering table. In other words, both the trigger event and the triggered action can operate on the same table. The following list summarizes the situations in which triggers can be re-entrant and the situations in which triggers cannot be re-entrant:

- The trigger action of an update trigger cannot be an INSERT or DELETE of the table that the trigger event updated.
- Similarly, the trigger action of an update trigger cannot be an UPDATE of a column that the trigger event updated. (But the trigger action of an update trigger can update a column that was not updated by the trigger event.)

  For example, assume that the following UPDATE statement, which updates columns **a** and **b** of **tab1**, is the triggering statement:

  ```
  UPDATE tab1 SET (a, b) = (a + 1, b + 1)
  ```

  Now consider the trigger actions in the following example. The first UPDATE statement is a valid trigger action, but the second one is not, because it updates column **b** again.

  ```
  UPDATE tab1 SET c = c + 1;    -- OK
  UPDATE tab1 SET b = b + 1;    -- INVALID
  ```

- If the trigger has an UPDATE event, the trigger action can be an EXECUTE PROCEDURE or EXECUTE FUNCTION statement with an INTO clause that references a column that was updated by the trigger event or any other column in the triggering table.

  When an EXECUTE PROCEDURE or EXECUTE FUNCTION statement is the trigger action, the INTO clause for an UPDATE trigger is valid only in FOR EACH ROW trigger actions, and column names that appear in the INTO clause must be from the triggering table.

  This statement illustrates the appropriate use of the INTO clause:

  ```
  CREATE TRIGGER upd_totpr UPDATE OF quantity ON items
     REFERENCING OLD AS pre_upd NEW AS post_upd
     FOR EACH ROW(EXECUTE PROCEDURE
        calc_totpr(pre_upd.quantity,post_upd.quantity,
        pre_upd.total_price) INTO total_price)
  ```

  The column that follows the INTO keyword must be in the triggering table, but need not have been updated by the trigger event.

  When the INTO clause appears in the EXECUTE PROCEDURE or EXECUTE FUNCTION statement, the database server updates the specified columns with values returned from the UDR, immediately upon returning from the UDR.

- If the trigger has an INSERT event, the trigger action cannot be an INSERT or DELETE statement that references the triggering table.

- If the trigger has an INSERT event, the trigger action can be an UPDATE statement that references a column in the triggering table. This column cannot, however, be a column for which a value was supplied by the trigger event.

  If the trigger has an INSERT event, and the trigger action updates the triggering table, the columns in both statements must be mutually exclusive. For example, assume that the triggering statement inserts values for columns **cola** and **colb** of table **tab1**:

  ```
  INSERT INTO tab1 (cola, colb) VALUES (1,10)
  ```

  Now consider the following trigger actions. The first UPDATE is valid, but the second one is not, because it updates column **colb** even though the trigger event already supplied a value for column **colb**:

  ```
  UPDATE tab1 SET colc=100; --OK
  UPDATE tab1 SET colb=100; --INVALID
  ```

- If the trigger has an INSERT event, the trigger action can be an EXECUTE PROCEDURE or EXECUTE FUNCTION statement with an INTO clause that references a column that was supplied by the trigger event or a column that was not supplied by the trigger event.

  When an EXECUTE PROCEDURE or EXECUTE FUNCTION statement is the trigger action, you can specify the INTO clause for an INSERT trigger only when the trigger action occurs in the FOR EACH ROW list. In this case, the INTO clause can contain only column names from the triggering table.

  The following statement illustrates the valid use of the INTO clause:

  ```
  CREATE TRIGGER ins_totpr INSERT ON items
     REFERENCING NEW AS new_ins
     FOR EACH ROW (EXECUTE PROCEDURE calc_totpr
        (0, new_ins.quantity, 0) INTO total_price).
  ```

  The column that follows the INTO keyword can be a column in the triggering table that was supplied by the trigger event, or a column in the triggering table that was not supplied by the trigger event.

  When the INTO clause appears in the EXECUTE PROCEDURE or (for Dynamic Server only) the EXECUTE FUNCTION statement, the database server immediately updates the specified columns with values returned from the UDR.

- If the trigger action is a SELECT statement, the SELECT statement can reference the triggering table. The SELECT statement can be a trigger action in the following instances:
  - ❑ The SELECT statement appears in a subquery in the WHEN clause or in a trigger-action statement.
  - ❑ The trigger action is a UDR, and the SELECT statement appears inside the UDR.

Re-Entrancy and Cascading Triggers

The cases when a trigger cannot be re-entrant apply recursively to all cascading triggers, which are considered part of the initial trigger. In particular, this rule means that a cascading trigger cannot update any columns in the triggering table that were updated by the original triggering statement, including any nontriggering columns affected by that statement. For example, assume this UPDATE statement is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1)
```

In the cascading triggers of the next example, **trig2** fails at runtime because it references column **b**, which the triggering UPDATE statement updates:

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1-- Valid
   AFTER (UPDATE tab2 set e = e + 1);

CREATE TRIGGER trig2 UPDATE of e ON tab2-- Invalid
   AFTER (UPDATE tab1 set b = b + 1);
```

Now consider the following SQL statements. When the final UPDATE statement is executed, column **a** is updated and the trigger **trig1** is activated.

The trigger action again updates column **a** with an EXECUTE PROCEDURE INTO statement.

```
CREATE TABLE temp1 (a int, b int, e int);
INSERT INTO temp1 VALUES (10, 20, 30);

CREATE PROCEDURE proc(val int) RETURNING int,int;
   RETURN val+10, val+20;
END PROCEDURE;

CREATE TRIGGER trig1 UPDATE OF a ON temp1
   FOR EACH ROW (EXECUTE PROCEDURE proc(50) INTO a, e);

CREATE TRIGGER trig2 UPDATE OF e ON temp1
   FOR EACH ROW (EXECUTE PROCEDURE proc(100) INTO a, e);

UPDATE temp1 SET (a,b) = (40,50);
```

XPS

In Extended Parallel Server, to re-create this example, use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

Several questions arise from this example of cascading triggers. First, should the update of column **a** activate trigger **trig1** again? The answer is no. Because the trigger was activated, it is not activated a second time. If the trigger action is an EXECUTE PROCEDURE INTO or EXECUTE FUNCTION INTO statement, the only triggers that are activated are those that are defined on columns that are mutually exclusive from the columns updated until then (in the cascade of triggers) in that table. Other triggers are ignored.

Another question that arises from the example is whether trigger **trig2** should be activated. The answer is yes. The trigger **trig2** is defined on column **e.** Until now, column **e** in table **temp1** has not been modified. Trigger **trig2** is activated.

A final question that arises from the example is whether triggers **trig1** and **trig2** should be activated after the trigger action in **trig2** is performed. The answer is no. Neither trigger is activated. By this time columns **a** and **e** have been updated once, and triggers **trig1** and **trig2** have been executed once. The database server ignores and does not activate these triggers. For more about cascading triggers, see "Cascading Triggers" on page 2-300.

IDS

As noted earlier, an INSTEAD OF trigger on a view cannot include the EXECUTE PROCEDURE INTO statement among its trigger actions. In addition, an error results if two views each have INSERT INSTEAD OF triggers with actions defined to perform insert operations on the other view. ♦

Rules for SPL Routines

In addition to the rules listed in "Re-Entrancy of Triggers" on page 2-294, the following rules apply to an SPL routine that is specified as a trigger action:

- The SPL routine cannot be a cursor function (one that returns more than one row) in a context where only one row is expected.
- You cannot use the old or new correlation name inside the SPL routine. If you need to use the corresponding values in the routine, you must pass them as parameters. The routine should be independent of triggers, and the old or new correlation name does not have any meaning outside the trigger.

When you use an SPL routine as a trigger action, the database objects that the routine references are not checked until the routine is executed.

See also the SPL restrictions in "Rules for Triggers" on page 2-271.

Privileges to Execute Trigger Actions

If you are not the trigger owner, but the privileges of the owner include WITH GRANT OPTION, you inherit the privileges of the owner (with grant option) in addition to your own privileges for each triggered SQL statement. If the trigger action is a UDR, you need Execute privilege on the UDR, or the trigger owner must have Execute privilege with grant option.

While executing the UDR, you do not carry the privileges of the trigger owner; instead you receive the privileges granted with the UDR, as follows:

1. Privileges for a DBA UDR

   When a UDR is registered with the CREATE DBA keywords and you are granted the Execute privilege on the UDR, the database server automatically grants you temporary DBA privileges that are available only when you are executing the UDR.
2. Privileges for a UDR without DBA restrictions

   If the UDR owner has the WITH GRANT OPTION right for the necessary privileges on the underlying database objects, you inherit these privileges when you are granted the Execute privilege.

For a UDR without DBA restrictions, all non-qualified database objects that the UDR references are implicitly qualified by the name of the UDR owner.

If the UDR owner has no WITH GRANT OPTION privilege, you have your original privileges on the underlying database objects when the UDR executes. For more information on privileges on SPL routines, refer to the *IBM Informix Guide to SQL: Tutorial*.

IDS

A view that has no INSTEAD OF trigger has only Select (with grant option) privilege. If an INSTEAD OF trigger is created on it, however, then the view has Insert (with grant option) privilege during creation of the trigger. The view owner can now grant only Select and Insert privileges to others. This is independent of the trigger action. It is not necessary to obtain Execute (with grant option) privilege on the procedure or function. By default, Execute (without grant option) privilege is granted on each UDR in the action list.

You can use roles with triggers. Role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements can appear within a UDR that the trigger action invokes. Privileges that a user has acquired through enabling a role or through a SET SESSION AUTHORIZATION statement are not relinquished when a trigger is executed.

On a complex view (with columns from more than one table), only the owner or DBA can create an INSTEAD OF trigger. The owner receives Select privileges when the trigger is created. Only after obtaining the required Execute privileges can the view owner grant privileges to other users. When the trigger on the complex view is dropped, all these privileges are revoked ♦

Creating a Trigger Action That Anyone Can Use

For a trigger to be executable by anyone who has the privileges to execute the triggering statement, you can ask the DBA to create a DBA-privileged UDR and grant you the Execute privilege with the WITH GRANT OPTION right.

You then use the DBA-privileged UDR as the trigger action. Anyone can execute the trigger action because the DBA-privileged UDR carries the WITH GRANT OPTION right. When you activate the UDR, the database server applies privilege-checking rules for a DBA.

Cascading Triggers

XPS

In this section and in sections that follow, any references to *nonlogging databases* do not apply to Extended Parallel Server. (In Extended Parallel Server, all databases support transaction logging.) ♦

The database server allows triggers other than select triggers to cascade, meaning that the trigger actions of one trigger can activate another trigger. (For further information on the restriction against cascading select triggers, see "Circumstances When a Select Trigger is Not Activated" on page 2-280.)

The maximum number of triggers in a cascading series is 61; the initial trigger plus a maximum of 60 cascading triggers. When the number of cascading triggers in a series exceeds the maximum, the database server returns error number -748, with the following message:

```
Exceeded limit on maximum number of cascaded triggers.
```

The next example illustrates a series of cascading triggers that enforce referential integrity on the **manufact**, **stock**, and **items** tables in the **stores_demo** database. When a manufacturer is deleted from the **manufact** table, the first trigger, **del_manu**, deletes all the items of that manufacturer from the **stock** table. Each DELETE in the **stock** table activates a second trigger, **del_items**, that deletes all **items** of that manufacturer from the **items** table. Finally, each DELETE in the **items** table triggers SPL routine **log_order**, creating a record of any orders in the **orders** table that can no longer be filled.

```
CREATE TRIGGER del_manu
   DELETE ON manufact REFERENCING OLD AS pre_del
   FOR EACH ROW(DELETE FROM stock
   WHERE manu_code = pre_del.manu_code);
CREATE TRIGGER del_stock
   DELETE ON stock REFERENCING OLD AS pre_del
   FOR EACH ROW(DELETE FROM items
   WHERE manu_code = pre_del.manu_code);
CREATE TRIGGER del_items
   DELETE ON items REFERENCING OLD AS pre_del
   FOR EACH ROW(EXECUTE PROCEDURE log_order(pre_del.order_num));
```

When you are not using logging, referential integrity constraints on both the **manufact** and **stock** tables prohibit the triggers in this example from executing. When you use logging, however, the triggers execute successfully because constraint checking is deferred until all the trigger actions are complete, including the actions of cascading triggers. For more information about how constraints are handled when triggers execute, see "Constraint Checking" on page 2-302.

The database server prevents loops of cascading triggers by not allowing you to modify the triggering table in any cascading trigger action, except an UPDATE statement that does not modify any column that the triggering UPDATE statement updated, or an INSERT statement. An INSERT trigger can define UPDATE trigger actions on the same table.

Constraint Checking

When you use logging, the database server defers constraint checking on the triggering statement until after the statements in the triggered-action list execute. This is equivalent to executing a SET CONSTRAINTS ALL DEFERRED statement before executing the triggering statement. After the trigger action is completed, the database server effectively executes a SET CONSTRAINTS *constraint* IMMEDIATE statement to check the constraints that were deferred. This action allows you to write triggers so that the trigger action can resolve any constraint violations that the triggering statement creates. For more information, see "SET Database Object Mode" on page 2-652.

Consider the following example, in which the table **child** has constraint **r1**, which references the table **parent**. You define trigger **trig1** and activate it with an INSERT statement. In the trigger action, **trig1** checks to see if **parent** has a row with the value of the current **cola** in **child**; if not, it inserts it.

```
CREATE TABLE parent (cola INT PRIMARY KEY);
CREATE TABLE child (cola INT REFERENCES parent CONSTRAINT r1);
CREATE TRIGGER trig1 INSERT ON child
   REFERENCING NEW AS new
   FOR EACH ROW
   WHEN((SELECT COUNT (*) FROM parent
      WHERE cola = new.cola) = 0)
-- parent row does not exist
   (INSERT INTO parent VALUES (new.cola));
```

When you insert a row into a table that is the child table in a referential constraint, the row might not exist in the parent table. The database server does not immediately return this error on a triggering statement. Instead, it allows the trigger action to resolve the constraint violation by inserting the corresponding row into the parent table. As the previous example shows, you can check within the trigger action to see whether the parent row exists, and if so, you can provide logic to bypass the INSERT action.

For a database without logging, the database server does *not* defer constraint checking on the triggering statement. In this case, the database server immediately returns an error if the triggering statement violates a constraint.

You cannot use the SET Transaction Mode statement in a trigger action. The database server checks this restriction when you activate a trigger, because the statement could occur inside a UDR.

XPS

Rows that cause constraint violations might appear in the violations table even if a later trigger action corrects the violation. ♦

Preventing Triggers from Overriding Each Other

When you activate multiple triggers with an UPDATE statement, a trigger can possibly override the changes that an earlier trigger made. If you do not want the trigger actions to interact, you can split the UPDATE statement into multiple UPDATE statements, each of which updates an individual column.

As another alternative, you can create a single update trigger for all columns that require a trigger action. Then, inside the trigger action, you can test for the column being updated and apply the actions in the desired order. This approach, however, is different than having the database server apply the actions of individual triggers, and it has the following disadvantages:

- If the triggering UPDATE statement sets a column to the current value, you cannot detect the UPDATE, so the trigger action is skipped. You might wish to execute the trigger action, even though the value of the column has not changed.
- If the trigger has a BEFORE action, it applies to all columns, because you cannot yet detect whether a column has changed.

External Tables

The trigger action can affect tables of other database servers. The following example shows an update trigger on **dbserver1**, which triggers an UPDATE to **items** on **dbserver2**:

```
CREATE TRIGGER upd_nt UPDATE ON newtab
   REFERENCING new AS post
   FOR EACH ROW(UPDATE stores_demo@dbserver2:items
      SET quantity = post.qty WHERE stock_num = post.stock
      AND manu_code = post.mc)
```

If, however, a statement from an external database server initiates a trigger whose action affects tables in an external database, the trigger actions fail.

For example, the following combination of trigger action and triggering statement results in an error when the triggering statement executes:

```
-- Trigger action from dbserver1 to dbserver3:

CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores_demo@dbserver3:items
   SET quantity = post.qty WHERE stock_num = post.stock
   AND manu_code = post.mc);
-- Triggering statement from dbserver2:

UPDATE stores_demo@dbserver1:newtab
   SET qty = qty * 2 WHERE s_num = 5
   AND mc = 'ANZ';
```

Logging and Recovery

You can create triggers for databases, with and without logging. If the trigger fails in a database that has transaction logging, the triggering statement and trigger actions are rolled back, as if the actions were an extension of the triggering statement, but the rest of the transaction is not rolled back.

In a database that does not have transaction logging, however, you cannot roll back when the triggering statement fails. In this case, you are responsible for maintaining data integrity in the database. The UPDATE, INSERT, or DELETE action of the triggering statement occurs before the trigger actions in the FOR EACH ROW section. If the trigger action fails for a database without logging, the application must restore the row that was changed by the triggering statement to its previous value.

If a trigger action calls a UDR, but the UDR terminates in an exception-handling section, any actions that modify data inside that section are rolled back with the triggering statement. In the following partial example, when the exception handler traps an error, it inserts a row into the table **logtab**:

```
ON EXCEPTION IN (-201)
   INSERT INTO logtab values (errno, errstr);
   RAISE EXCEPTION -201
END EXCEPTION
```

When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this INSERT because it is part of the trigger actions. If the UDR is executed outside a trigger action, the INSERT is not rolled back.

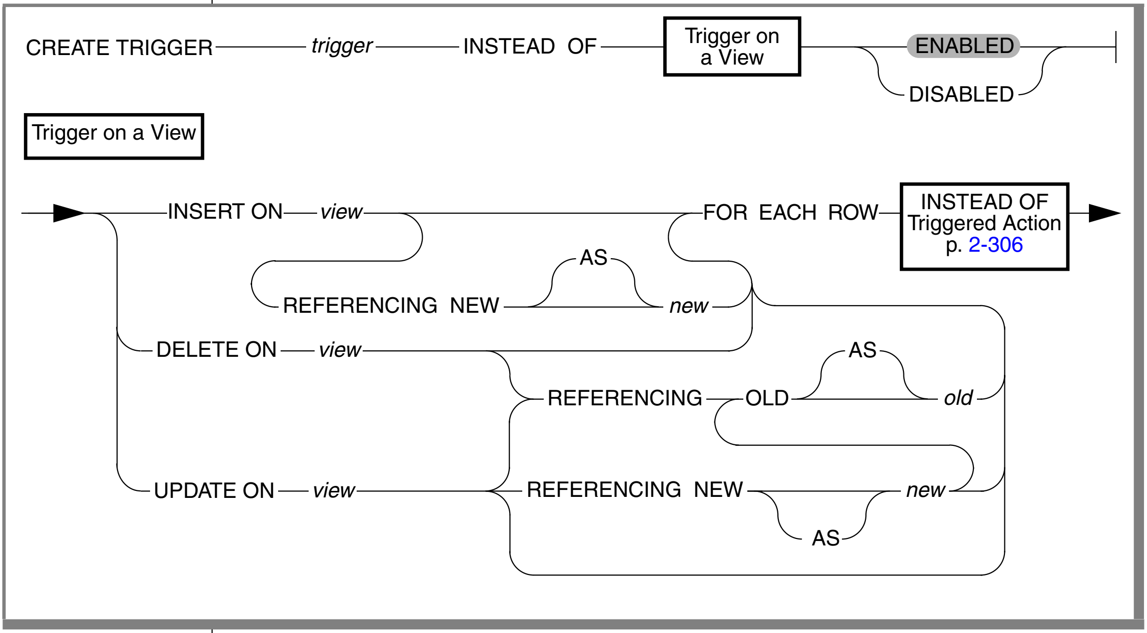
The UDR that implements a trigger action cannot contain any BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. If the database has transaction logging, you must either begin an explicit transaction before the triggering statement, or the statement itself must be an implicit transaction. In any case, no other transaction-related statement is valid inside the UDR.

You can use triggers to enforce referential actions that the database server does not currently support. In a database without logging, you are responsible for maintaining data integrity when the triggering statement fails.

IDS

INSTEAD OF Triggers on Views

Use INSTEAD OF triggers to perform a specified trigger action on a view, rather than execute the triggering INSERT, DELETE, or UPDATEevent.



Element	Purpose	Restrictions	Syntax
new	Name for old value in *view* column	Must be unique within this statement	Identifier, p. 4-189
old	Name for new value in *view* column.	Must be unique within this statement	Identifier, p. 4-189

Element	Purpose	Restrictions	Syntax
trigger	Name declared here for the trigger	Must be unique among the names of triggers in the database	Database Object Name, p. 4-46
view	Name or synonym of the triggering view. Can include *owner***.** qualifier.	The view or synonym must exist in the current database.	Database Object Name, p. 4-46

You can use the trigger action to update the table(s) underlying the view, in some cases updating an otherwise "non-updatable" view. You can also use INSTEAD OF triggers to substitute other actions when INSERT, DELETE, or UPDATE statements reference specific columns within the database.

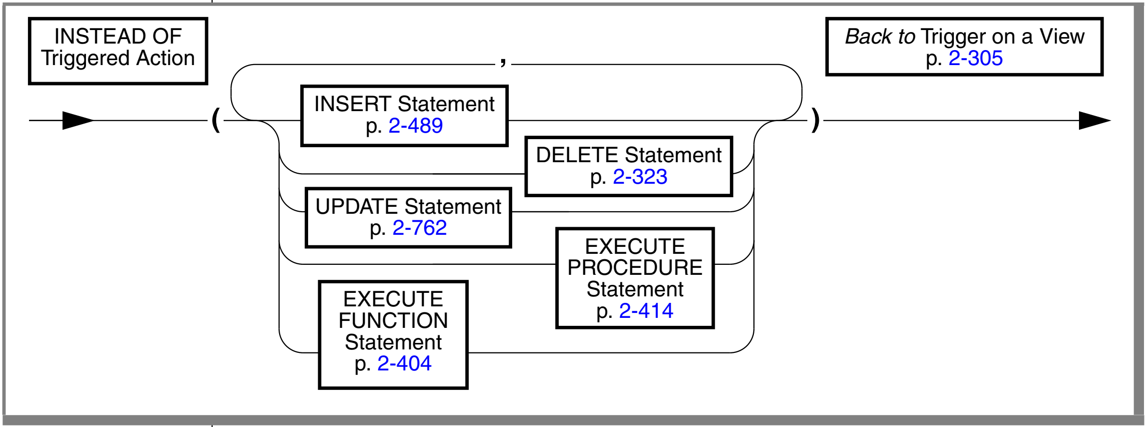
In the optional REFERENCING clause of an INSTEAD OF UPDATE trigger, the *new* correlation name can appear before or after the *old* correlation name.

The specified *view* is sometimes called the *triggering view*. The left-hand portion of this diagram (including the *view* specification) defines the *trigger event*. The rest of the diagram defines correlation names and the *trigger action*.

IDS

The Action Clause of INSTEAD OF Triggers

When the trigger event for the specified *view* is encountered, the SQL statements of the trigger action are executed, instead of the triggering statement. Triggers defined on a view support the following syntax in the action clause.


INSTEAD OF Triggered Action
Back to Trigger on a View p. 2-305
,
(
INSERT Statement p. 2-489
DELETE Statement p. 2-323
UPDATE Statement p. 2-762
EXECUTE PROCEDURE Statement p. 2-414
EXECUTE FUNCTION Statement p. 2-404
)

This is not identical to the syntax of the trigger action for a trigger on a table, as described in the section "Triggered-Action List" on page 288. Because no WHEN (*condition*) is supported, the same trigger action is executed whenever the INSTEAD OF trigger event is encountered, and only one action list can be specified, rather than a separate list for each *condition*.

IDS

Rules for INSTEAD OF Triggers on Views

You must be either the owner of the *view* or have the DBA status to create an INSTEAD OF trigger on a view. The owner of a simple view (based on one table) has Insert, Update, ans Delete privileges. For information about the relationship between the privileges of the trigger owner and the privileges of other users, see "Privileges to Execute Trigger Actions" on page 2-298.

An INSTEAD OF trigger defined on a view cannot violate the "Rules for Triggers" on page 2-271, and must observe the following additional rules:

- You can define an INSTEAD OF trigger only on a view, not on a table.
- The view must be local to the current database.
- The view cannot be an updatable view WITH CHECK OPTION.
- No SELECT event or WHEN clause is valid in an INSTEAD OF trigger,
- No BEFORE nor AFTER action is valid in an INSTEAD OF trigger.
- No OF *column* clause is valid in an INSTEAD OF UPDATE trigger.
- Every INSTEAD OF trigger must specify FOR EACH ROW.
- The triggered action cannot include EXECUTE PROCEDURE INTO.

A view can have no more than one INSTEAD OF trigger defined for each type of event (INSERT, DELETE, or UPDATE). It is possible, however, to define a trigger on one or more other views, each with its own INSTEAD OF trigger.

Updating Views

INSERT, DELETE, or UPDATE statements can directly modify a view only if all of the following are true of the SELECT statement that defines the view:

- All of the columns in the view are from a single table.
- No columns in the projection list are aggregate values.
- No UNIQUE or DISTINCT keyword in the SELECT projection list

- No GROUP BY clause nor UNION operator in the view definition
- The query selects no calculated values and no literal values

By using INSTEAD OF triggers, however, you can circumvent these restrictions on the view, if the trigger action modifies the base table(s).

Example of an INSTEAD OF Trigger on a View

Suppose that **dept** and **emp** are tables that list departments and employees:

```
CREATE TABLE dept (
   deptno INTEGER PRIMARY KEY,
   deptname CHAR(20),
   manager_num INT
);
CREATE TABLE emp (
   empno INTEGER PRIMARY KEY,
   empname CHAR(20),
   deptno INTEGER REFERENCES dept(deptno),
   startdate DATE
);
ALTER TABLE dept ADD CONSTRAINT(FOREIGN KEY (manager_num)
      REFERENCES emp(empno));
```

The next statement defines **manager_info**, a view of columns in the **dept** and **emp** tables that includes all the managers of each department:

```
CREATE VIEW manager_info AS
   SELECT d.deptno, d.deptname, e.empno, e.empname
      FROM emp e, dept d WHERE e.empno = d.manager_num;
```

The following CREATE TRIGGER statement creates **manager_info_insert**, an INSTEAD OF trigger that is designed to insert rows into the **dept** and **emp** tables through the **manager_info** view:

```
CREATE TRIGGER manager_info_insert
   INSTEAD OF INSERT ON manager_info    --defines trigger event
      REFERENCING NEW AS n              --new manager data
   FOR EACH ROW                         --defines trigger action
      (EXECUTE PROCEDURE instab(n.deptno, n.empno));

CREATE PROCEDURE instab (dno INT, eno INT)
   INSERT INTO dept(deptno, manager_num) VALUES(dno, eno);
   INSERT INTO emp (empno, deptno) VALUES (eno. dno);emp(empno));
END PROCEDURE;
```

After the tables, view, trigger, and SPL routine have been created, the database server treats the following INSERT statement as a triggering event:

```
INSERT INTO manager_info(deptno, empno) VALUES (08, 4232);
```

This triggering INSERT statement is not executed, but this event causes the trigger action to be executed instead, invoking the **instab( )** SPL routine. The INSERT statements in the SPL routine insert new values into both the **emp** and **dept** base tables of the **manager_info** view.

Related Information

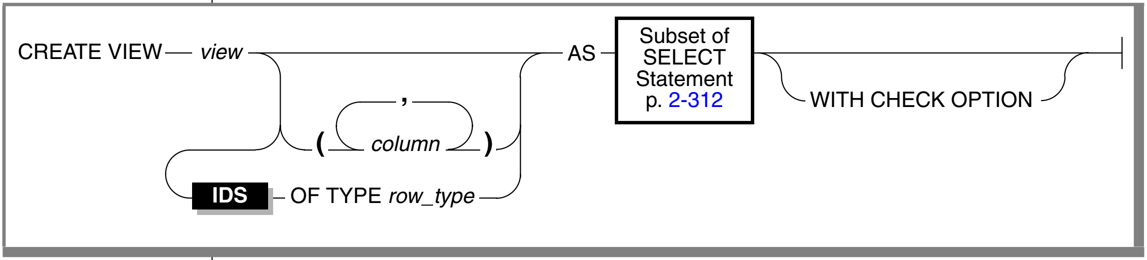
Related statements: CREATE PROCEDURE, **CREATE VIEW**, DROP TRIGGER, EXECUTE PROCEDURE, and SET Database Object Mode

For a task-oriented discussion of triggers, and for examples of INSTEAD OF DELETE (and UPDATE) triggers on views, see the *IBM Informix Guide to SQL: Tutorial*. For performance implications of triggers, see your *Performance Guide*.

CREATE VIEW

Use the CREATE VIEW statement to create a new view that is based on one or more existing tables and views in the database.

Syntax



Element	Purpose	Restrictions	Syntax
column	A column in the view	See "Naming View Columns" p. 2-313.	Identifier, p. 4-189
row_type	Named-row type for typed view	Must exist before you assign it to a view.	Data Type, p. 4-49
view	Name that you declare here for the view	Must be unique among view, table, and synonym names in the database.	Database Object Name, p. 4-46

Usage

A *view* is a virtual table, defined by a SELECT statement. Except for the statements in the following list, you can specify the name or synonym of a view in any SQL statement where the name of a table is syntactically valid:

ALTER FRAGMENT
CREATE INDEX
CREATE TABLE
CREATE TRIGGER
RENAME TABLE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE
UPDATE STATISTICS

"Updating Through Views" on page 2-315 prohibits "non-updatable" views in INSERT, DELETE, or UPDATE statements (where other views are valid).

To create a view, you must have the Select privilege on all columns from which the view is derived.You can query a view as if it were a table, and in some cases, you can update it as if it were a table; but a view is not a table.

The view consists of the set of rows and columns that the SELECT statement in the view definition returns each time you refer to the view in a query.

In some cases, the database server merges the SELECT statement of the user with the SELECT statement defining the view, and executes the combined statements. In other cases, a query against a view might execute more slowly than expected, if the complexity of the view definition causes the database server to create a temporary table (referred to as a "materialized view"). For more information on materialized views, see the *Performance Guide*.

The view reflects changes to the underlying tables with one exception. If a SELECT * specification defines the view, the view has only the columns that existed in the underlying tables at the time when the view was defined by CREATE VIEW. Any new columns that are subsequently added to the underlying tables with the ALTER TABLE statement do not appear in the view.

The view inherits the data types of the columns in the tables from which the view is derived. The database server determines data types of virtual columns from the nature of the expression.

The SELECT statement is stored in the **sysviews** system catalog table. When you subsequently refer to a view in another statement, the database server performs the defining SELECT statement while it executes the new statement.

DB

In DB-Access, if you create a view outside the CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set the **DBANSIWARN** environment variable. ♦

The following statement creates a view that is based on the **person** table. When you create a view like this, which has no OF TYPE clause, the view is referred to as an *untyped view.*

```
CREATE VIEW v1 AS SELECT * FROM person
```

Typed Views

You can create *typed views* if you have Usage privileges on the named-ROW type or if you are its owner or the DBA. If you omit the OF TYPE clause, rows in the view are considered untyped, and default to an unnamed-ROW type.

Typed views, like typed tables, are based on a named-ROW type. Each column in the view corresponds to a field in the named-ROW type. The following statement creates a typed view that is based on the table **person**.

```
CREATE VIEW v2 OF TYPE person_t AS SELECT * FROM person
```

To create a typed view, you must include an OF TYPE clause. When you create a typed view, the named-ROW type that you specify immediately after the OF TYPE keywords must already exist.

Subset of SELECT Statements Valid in View Definitions

You cannot create a view on a temporary table. The FROM clause of the SELECT statement cannot contain the name of a temporary table.

If Select privileges are revoked from a user for a table that is referenced in the SELECT statement defining a view that the same user owns, then that view is dropped, unless it also includes columns from tables in another database.

IDS

You cannot create a view on typed tables (including any table that is part of a table hierarchy) in a remote database. ♦

Do not use display labels in the select list of the projection clause. Display labels in the select list are interpreted as column names.

The SELECT statement in CREATE VIEW cannot include the FIRST, the INTO TEMP, or the ORDER BY clauses. For complete information about SELECT statement syntax and usage, see "SELECT" on page 2-581.

Union Views

A view that contains a UNION or UNION ALL operator in its SELECT statement is known as a *union view*. Certain restrictions apply to union views:

- If a CREATE VIEW statement defines a union view, you cannot specify the WITH CHECK OPTION keywords in the CREATE VIEW statement.
- All restrictions that apply to UNION or UNION ALL operations in standalone SELECT statements also apply to UNION and UNION ALL operations in the SELECT statement of a union view.

For a list of these restrictions, see "Restrictions on a Combined SELECT" on page 2-637. For an example of a CREATE VIEW statement that defines a union view, see "Naming View Columns."

Naming View Columns

The number of columns that you specify in the *column* list must match the number of columns returned by the SELECT statement that defines the view. If you do not specify a list of columns, the view inherits the column names of the underlying tables. In the following example, the view **herostock** has the same column names as the ones in the SELECT statement:

```
CREATE VIEW herostock AS
   SELECT stock_num, description, unit_price, unit, unit_descr
      FROM stock WHERE manu_code = 'HRO'
```

You must specify at least one column name in the following circumstances:

- If you provide names for some of the columns in a view, then you must provide names for all the columns. That is, the *column* list must contain an entry for every column that appears in the view.
- If the SELECT statement returns an expression, the corresponding column in the view is called a *virtual* column. You must provide a name for virtual columns. In the following example, the user must specify the *column* parameter because the select list of thr projection clause of the SELECT statement contains an aggregate expression:

  ```
  CREATE VIEW newview (firstcol, secondcol) AS
     SELECT sum(cola), colb FROM oldtab
  ```

- You must also specify a column name in cases where the selected columns have duplicate column names without the *table* qualifiers. For example, if both **orders.order_num** and **items.order_num** appear in the SELECT statement, the CREATE VIEW statement, must provide two separate column names to label them:

  ```
  CREATE VIEW someorders (custnum,ocustnum,newprice) AS
     SELECT orders.order_num,items.order_num,
           items.total_price*1.5
        FROM orders, items
        WHERE orders.order_num = items.order_num
        AND items.total_price > 100.00
  ```

  Here **custnum** and **ocustnum** replace the identical column names.

- The CREATE VIEW statement must also provide column names in the *column* list when the SELECT statement includes a UNION or UNION ALL operator and the names of the corresponding columns in the SELECT statements are not identical.

- Code in the following example must specify the *column* list because the second column in the first SELECT statement has a different name from the second column in the second SELECT statement:

```
CREATE VIEW myview (cola, colb) AS
   SELECT colx, coly from firsttab
   UNION
   SELECT colx, colz from secondtab
```

Using a View in the SELECT Statement

You can define a view in terms of other views, but you must abide by the restrictions on creating views that are discussed in the *IBM Informix Database Design and Implementation Guide.*

WITH CHECK OPTION Keywords

The WITH CHECK OPTION keywords instruct the database server to ensure that all modifications that are made through the view to the underlying tables satisfy the definition of the view.

The following example creates a view that is named **palo_alto**, which uses all the information in the **customer** table for customers in the city of Palo Alto. The database server checks any modifications made to the **customer** table through **palo_alto** because the WITH CHECK OPTION is specified.

```
CREATE VIEW palo_alto AS
   SELECT * FROM customer WHERE city = 'Palo Alto'
      WITH CHECK OPTION
```

You can insert into a view a row that does not satisfy the conditions of the view (that is, a row that is not visible through the view). You can also update a row of a view so that it no longer satisfies the conditions of the view. For example, if the view was created without the WITH CHECK OPTION keywords, you could insert a row through the view where the city is Los Altos, or you could update a row through the view by changing the city from Palo Alto to Los Altos.

To prevent such inserts and updates, you can add the WITH CHECK OPTION keywords when you create the view. These keywords ask the database server to test every inserted or updated row to ensure that it meets the conditions that are set by the WHERE clause of the view. The database server rejects the operation with an error if the row does not meet the conditions.

Even if the view was created with the WITH CHECK OPTION keywords, however, you can perform inserts and updates through the view to change columns that are not part of the view definition. A column is not part of the view definition if it does not appear in the WHERE clause of the SELECT statement that defines the view.

Updating Through Views

If a view is built on a single table, the view is *updatable* if the SELECT statement that defined the view did not contain any of the following elements:

- Columns in the select list that are aggregate values
- Columns in the select list that use the UNIQUE or DISTINCT keyword
- A GROUP BY clause
- A UNION operator
- The query selects calculated values or literal values

(You can DELETE from a view that selects calculated values from a single table, but INSERT and UPDATE are not valid operations on such views.)

In an updatable view, you can update the values in the underlying table by inserting values into the view. If a view is built on a table that has a derived value for a column, however, that column is not updatable through the view, but other columns in the view can be updated.

See also "Updating Views" on page 2-307 for information about using INSTEAD OF triggers to update views that are based on more than one table, or that include columns containing aggregates or other calculated values.

Important: *You cannot update or insert rows in a remote table through views that were created using the* WITH CHECK OPTION *keywords.*

Related Information

Related statements: CREATE TABLE, CREATE TRIGGER, DROP VIEW, GRANT, SELECT, and SET SESSION AUTHORIZATION

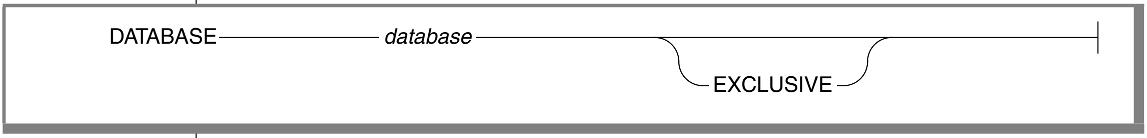
For a discussion of views, see the *IBM Informix Database Design and Implementation Guide*.

+

DATABASE

Use the DATABASE statement to open an accessible database as the current database.

Syntax



Element	Purpose	Restrictions	Syntax
database	Name of the database to select	The database must exist	Database Name, p. 4-44

Usage

You can use the DATABASE statement to select any database on your database server. To select a database on another database server, specify the name of the database server with the database name.

If you include the name of the current (or another) database server with the *database* name, the database server name cannot be uppercase. (See "Database Name" on page 4-44 for the syntax of specifying the database server name.)

Issuing a DATABASE statement when a database is already open closes the current database before opening the new one. Closing the current database releases any cursor resources that the database server holds, invalidating any cursors that you have declared up to that point. If the *user* specification was changed through a SET SESSION AUTHORIZATION statement, the original *user* name is restored when the new database is opened.

The current user (or PUBLIC) must have the Connect privilege on the database that is specified in the DATABASE statement. The current user cannot have the same user name as an existing role in the database.

DATABASE is not a valid statement in multistatement PREPARE operations.

E/C

SQLCA.SQLWARN Settings Immediately after DATABASE Executes

Immediately after DATABASE executes, you can identify characteristics of the specified *database* by examining warning flags in the **sqlca** structure.

- If the first field of **sqlca.sqlwarn** is blank, no warnings were issued.
- The second **sqlca.sqlwarn** field is set to the letter `W` if the *database* that was opened supports transaction logging.
- ANSI The third field is set to `W` if *database* is an ANSI-compliant database. ♦
- The fourth field is set to `W` if *database* is a Dynamic Server database.
- The fifth field is set to `W` if *database* converts all floating-point data to DECIMAL format. (System lacks FLOAT and SMALLFLOAT support.)
- The seventh field is set to `W` if *database* is the secondary server (that is, running in read-only mode) in a data-replication pair.
- GLS The eighth field is set to `W` if *database* has **DB_LOCALE** set to a locale different from the **DB_LOCALE** setting on the client system. ♦

EXCLUSIVE Keyword

The EXCLUSIVE keyword opens the database in exclusive mode and prevents access by anyone but the current user. To allow others access to the database, you must first execute the CLOSE DATABASE statement and then reopen the database without the EXCLUSIVE keyword. The following statement opens the **stores_demo** database on the **training** database server in exclusive mode:

```
DATABASE stores_demo@training EXCLUSIVE
```

If another user has already opened the database, exclusive access is denied, an error is returned, and no database is opened.

Related Information

Related statements: CLOSE DATABASE, CONNECT, DISCONNECT, and SET CONNECTION

For discussions of how to use different data models to design and implement a database, see the *IBM Informix Database Design and Implementation Guide*.

For descriptions of the **sqlca** structure, see the *IBM Informix Guide to SQL: Tutorial* or the *IBM Informix ESQL/C Programmer's Manual*.

+

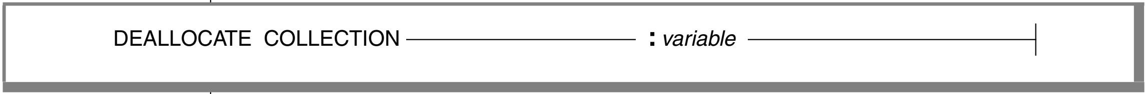
IDS

E/C

DEALLOCATE COLLECTION

Use the DEALLOCATE COLLECTION statement to release memory for a collection variable that was previously allocated with the ALLOCATE COLLECTION statement. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
variable	Name that identifies a typed or untyped collection variable for which to deallocate memory	Must be the name of an ESQL/C collection variable that has already been allocated	Name must conform to language-specific rules for names of variables

Usage

The DEALLOCATE COLLECTION statement frees all the memory that is associated with the ESQL/C collection variable that *variable* identifies. You must explicitly release memory resources for a collection variable with DEALLOCATE COLLECTION. Otherwise, deallocation occurs automatically at the end of the program.

The DEALLOCATE COLLECTION statement releases resources for both typed and untyped collection variables.

Tip: *The DEALLOCATE COLLECTION statement deallocates memory for an ESQL/C collection variable only. To deallocate memory for an ESQL/C row variable, use the DEALLOCATE ROW statement.*

If you deallocate a nonexistent collection variable or a variable that is not an ESQL/C collection variable, an error results. Once you deallocate a collection variable, you can use the ALLOCATE COLLECTION to reallocate resources and you can then reuse a collection variable.

Example

This example shows how to deallocate resources with the DEALLOCATE COLLECTION statement for the untyped collection variable, **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
. . .
EXEC SQL deallocate collection :a_set;
```

Related Information

Related example: refer to the collection variable example in PUT.

Related statements: ALLOCATE COLLECTION and DEALLOCATE ROW

For a discussion of collection data types, see the *IBM Informix ESQL/C Programmer's Manual*.

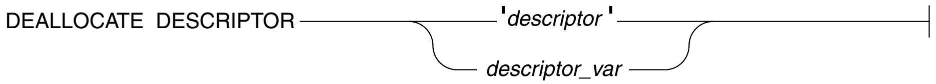
+

E/C

DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free a previously allocated, system-descriptor area. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Name of a system-descriptor area	Use single quotes. System-descriptor area must already be allocated	Quoted String, p. 4-243
descriptor_var	Host variable that contains the name of a system-descriptor area	System-descriptor area must already be allocated, and the variable must already have been declared	Name must conform to language-specific rules for names

Usage

The DEALLOCATE DESCRIPTOR statement frees all the memory that is associated with the system-descriptor area that *descriptor* or *descriptor_var* identifies. It also frees all the item descriptors (including memory for data items in the item descriptors).

You can reuse a descriptor or descriptor variable after it is deallocated. Otherwise, deallocation occurs automatically at the end of the program.

If you deallocate a nonexistent descriptor or descriptor variable, an error results.

You cannot use the DEALLOCATE DESCRIPTOR statement to deallocate an **sqlda** structure. You can use it only to free the memory that is allocated for a system-descriptor area.

The following examples show valid DEALLOCATE DESCRIPTOR statements. The first line uses an embedded-variable name, and the second line uses a quoted string to identify the allocated system-descriptor area.

```
EXEC SQL deallocate descriptor :descname;

EXEC SQL deallocate descriptor 'desc1';
```

Related Information

Related statements: ALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR

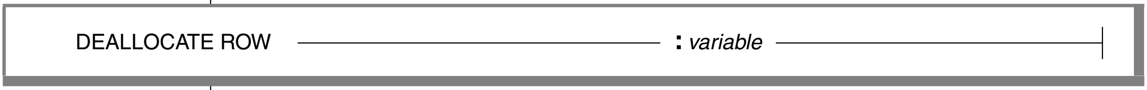
For more information on system-descriptor areas, refer to the *IBM Informix ESQL/C Programmer's Manual*.

+
IDS
E/C

DEALLOCATE ROW

Use the DEALLOCATE ROW statement to release memory for a ROW variable. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
variable	Typed or untyped row variable	Must be declared and allocated	Language specific

Usage

DEALLOCATE ROW frees all the memory that is associated with the ESQL/C typed or untyped **row** variable that *variable* identifies. If you do not explicitly release memory resources with DEALLOCATE ROW. deallocation occurs automatically at the end of the program. To deallocate memory for an ESQL/C collection variable, use the DEALLOCATE COLLECTION statement.

After you deallocate a ROW variable, you can use the ALLOCATE ROW statement to reallocate resources, and you can then reuse a ROW variable. The following example shows how to deallocate resources for the ROW variable, **a_row**, using the DEALLOCATE ROW statement:

```
EXEC SQL BEGIN DECLARE SECTION; row (a int, b int) a_row;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate row :a_row;
. . .
EXEC SQL deallocate row :a_row;
```

Related Information

Related statements: ALLOCATE ROW and DEALLOCATE COLLECTION

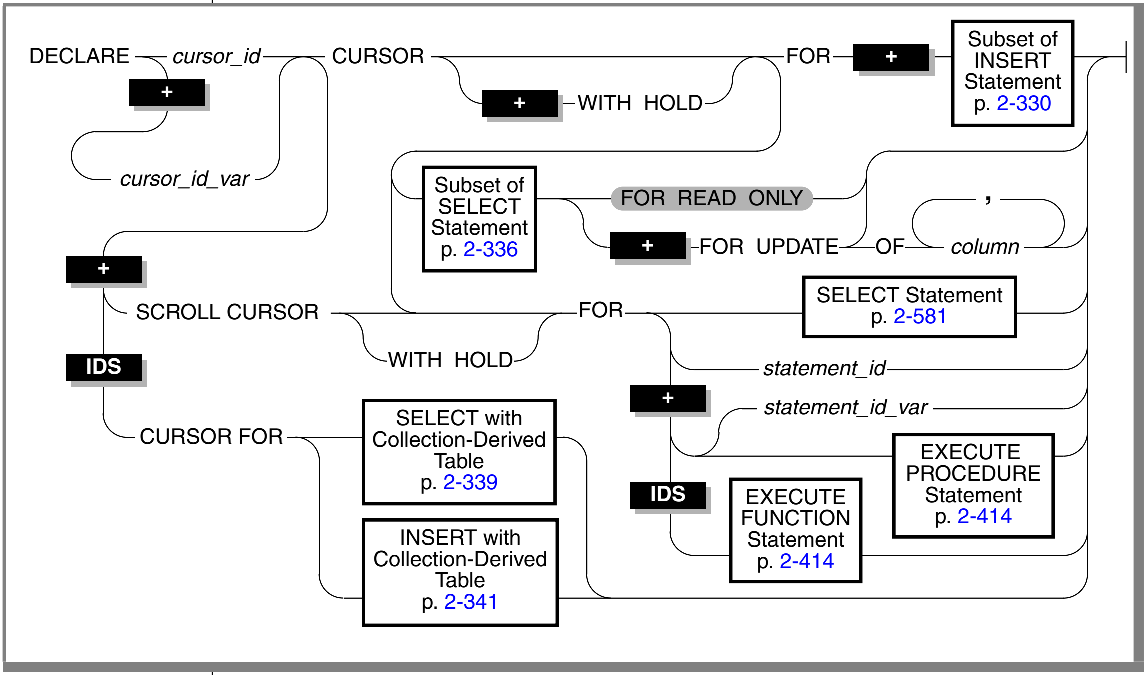
For a discussion of ROW data types, see the *IBM Informix Guide to SQL: Tutorial*. For complex data types, see the *IBM Informix ESQL/C Programmer's Manual*.

E/C

DECLARE

Use the DECLARE statement to associate a cursor with a set of rows. Use this statement with ESQL/C.

Syntax


DECLARE
cursor_id
+
cursor_id_var
CURSOR
+
WITH HOLD
FOR
+
Subset of INSERT Statement p. 2-330
Subset of SELECT Statement p. 2-336
FOR READ ONLY
+
FOR UPDATE
OF
,
column
+
SCROLL CURSOR
WITH HOLD
FOR
SELECT Statement p. 2-581
statement_id
+
statement_id_var
IDS
EXECUTE PROCEDURE Statement p. 2-414
IDS
EXECUTE FUNCTION Statement p. 2-414
IDS
CURSOR FOR
SELECT with Collection-Derived Table p. 2-339
INSERT with Collection-Derived Table p. 2-341

Element	Purpose	Restrictions	Syntax
column	Column to update with cursor	Must exist, but need not be listed in the Select list of Projection clause	Identifier, p. 4-189
cursor_id	Name declared here for cursor	Must be unique among names of cursors and prepared objects	Identifier, p. 4-189
cursor_id_var	Variable holding *cursor_id*	Must have a character data type	Language specific
statement_id	Name of prepared statement	Declared in PREPARE statement	Identifier, p. 4-189
statement_id_var	Variable holding *statement_id*	Must have a character data type	Language specific

Usage

A *cursor* is an identifier that you associate with a group of rows. DECLARE associates the cursor with one of the following database objects:

- With an SQL statement, such as SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE), or INSERT

  Each of these SQL statements creates a different type of cursor. For more information, see "Overview of Cursor Types" on page 2-325.
- With the statement identifier (*statement id* or *statement id variable*) of a prepared statement

  You can prepare one of the previous SQL statements and associate the prepared statement with a cursor. For more information, see "Associating a Cursor with a Prepared Statement" on page 2-338.

IDS

- With a collection variable in an ESQL/C program

  The name of the collection variable appears in the FROM clause of a SELECT or the INTO clause of an INSERT. For more information, see "Associating a Cursor with a Collection Variable" on page 2-339. ♦

DECLARE assigns an identifier to the cursor, specifies its uses, and directs the ESQL/C preprocessor to allocate storage for it. DECLARE must precede any other statement that refers to the cursor during program execution.

The maximum length of a DECLARE statement is 64 kilobytes. The number of cursors and prepared objects that can exist concurrently in a single program is limited by the available memory. To avoid exceeding the limit, use the FREE statement to release some prepared statements or cursors.

A program can consist of one or more source-code files. By default, the scope of reference of a cursor is global to a program, so a cursor that was declared in one source file can be referenced from a statement in another file. In a multiple-file program, if you want to limit the scope of cursor names to the files in which they are declared, you must preprocess all of the files with the **-local** command-line option.

Multiple cursors can be declared for the same prepared statement identifier. For example, the following ESQL/C example does not return an error:

```
EXEC SQL prepare id1 from 'select * from customer';
EXEC SQL declare x cursor for id1;
EXEC SQL declare y scroll cursor for id1;
EXEC SQL declare z cursor with hold for id1;
```

If you include the -**ansi** compilation flag (or if **DBANSIWARN** is set), warnings are generated for statements that use dynamic cursor names or dynamic statement identifiers, and (for Dynamic Server only) statements that use collection-derived tables. Some error checking is performed at runtime, such as these typical checks:

- Invalid use of sequential cursors as scroll cursors
- Use of undeclared cursors
- Invalid cursor names or statement names (empty)

Checks for multiple declarations of a cursor of the same name are performed at compile time only if the cursor or statement is specified as an identifier. The following example uses a host variable to store the cursor name:

```
EXEC SQL declare x cursor for select * from customer;
. . .
stcopy("x", s);
EXEC SQL declare :s cursor for select * from customer;
```

IDS

A cursor uses the collating order of the session when the cursor was declared, even if this is different from the collation of the session at runtime. ♦

Overview of Cursor Types

Cursors are typically required for data manipulation language (DML) operations on more than one row of data (or on an ESQL/C collection variable). You can declare the following types of cursors with the DECLARE statement:

- A *select cursor* is a cursor associated with A SELECT statement.
- A *function cursor* is a cursor associated with an EXECUTE FUNCTION (OR EXECUTE PROCEDURE) statement.
- An *insert cursor* is a cursor associated with an INSERT statement.

Sections that follow describe each of these cursor types. Cursors can also have *sequential*, *scroll*, and *hold* characteristics (but an insert cursor cannot be a scroll cursor). These characteristics determine the structure of the cursor; see "Cursor Characteristics" on page 2-332. In addition, a select or function cursor can specify *read-only* or *update* mode. For more information, see "Select Cursor or Function Cursor" on page 2-326.

Tip: *Function cursors behave the same as select cursors that are enabled as update cursors.*

A cursor that is associated with a statement identifier can be used with an INSERT, SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that is prepared dynamically, and to use different statements with the same cursor at different times. In this case, the type of cursor depends on the statement that is prepared at the time the cursor is opened. (See "Associating a Cursor with a Prepared Statement" on page 2-338.)

Select Cursor or Function Cursor

When an SQL statement returns more than one group of values to an ESQL/C program, you must declare a cursor to save the multiple groups, or rows, of data and to access these rows one at a time. You must associate the following SQL statements with cursors:

- If you associate a SELECT statement with a cursor, the cursor is called a *select cursor*.

  A select cursor is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved.
- If you associate an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor, the cursor is called a *function cursor*.

  The function cursor represents the columns or values that a user-defined function returns. Function cursors behave the same as select cursors that are enabled as update cursors.

XPS

In Extended Parallel Server, to create a function cursor, you must use the EXECUTE PROCEDURE statement. Extended Parallel Server does not support the EXECUTE FUNCTION statement. ♦

IDS

In Dynamic Server, for backward compatibility, if an SPL function was created with the CREATE PROCEDURE statement, you can create a function cursor with the EXECUTE PROCEDURE statement. With external functions, you must use the EXECUTE FUNCTION statement. ♦

When you associate a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor, the statement can include an INTO clause. However, if you prepare the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, you must omit the INTO clause in the PREPARE statement and use the INTO clause of the FETCH statement to retrieve the values from the collection cursor.

A select or function cursor can scan returned rows of data and to move data row by row into a set of receiving variables, as the following steps describe:

1. DECLARE

   Use DECLARE to define a cursor and associate it with a statement.

2. OPEN

   Use OPEN to open the cursor. The database server processes the query until it locates or constructs the first row of the active set.

3. FETCH

   Use FETCH to retrieve successive rows of data from the cursor.

4. CLOSE

   Use CLOSE to close the cursor when its active set is no longer needed.

5. FREE

   Use FREE to release the resources that are allocated for the cursor.

Using the FOR READ ONLY Option

Use the FOR READ ONLY keywords to define a cursor as a read-only cursor. A cursor declared to be read-only cannot be used to update (or delete) any row that it fetches.

The need for the FOR READ ONLY keywords depends on whether your database is ANSI compliant or not ANSI compliant.

In a database that is not ANSI compliant, the cursor that the DECLARE statement defines is a read-only cursor by default, so you do not need to specify the FOR READ ONLY keywords if you want the cursor to be a read-only cursor. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation.

ANSI

In an ANSI-compliant database, the cursor associated with a SELECT statement through the DECLARE statement is an update cursor by default, provided that the SELECT statement conforms to all of the restrictions for update cursors listed in "Subset of SELECT Statement Associated with Cursors" on page 2-336. If you want a select cursor to be read only, you must use the FOR READ ONLY keywords when you declare the cursor. ♦

The database server can use less stringent locking for a read-only cursor than for an update cursor.

The following example creates a read-only cursor:

```
EXEC SQL declare z_curs cursor for
   select * from customer_ansi
   for read only;
```

Using the FOR UPDATE Option

Use the FOR UPDATE option to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.

ANSI

In an ANSI-compliant database, you can use a select cursor to update or delete data if the cursor was not declared with the FOR READ ONLY keywords and it follows the restrictions on update cursors that are described in "Subset of SELECT Statement Associated with Cursors" on page 2-336. You do not need to use the FOR UPDATE keywords when you declare the cursor. ♦

The following example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
   select * from customer_notansi
   for update;
```

In an update cursor, you can update or delete rows in the active set. After you create an update cursor, you can update or delete the currently selected row by using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they take the place of the usual test expressions in the WHERE clause.

An update cursor lets you perform updates that are not possible with the UPDATE statement because the decision to update and the values of the new data items can be based on the original contents of the row. Your program can evaluate or manipulate the selected data before it decides whether to update. The UPDATE statement cannot interrogate the table that is being updated.

You can specify particular columns that can be updated. The columns need not appear in the Select list of the Projection clause.

Using FOR UPDATE with a List of Columns

When you declare an update cursor, you can limit the update to specific columns by including the OF keyword and a list of columns. You can modify only those named columns in subsequent UPDATE statements. The columns need not be in the select list of the SELECT clause.

The next example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_notansi** table:

```
EXEC SQL declare name_curs cursor for
   select * from customer_notansi
   for update of fname, lname;
```

ANSI

By default, unless declared as FOR READ ONLY, a select cursor in a database that is ANSI compliant is an update cursor, so the FOR UPDATE keywords are optional. If you want an update cursor to be able to modify only some of the columns in a table, however, you must specify these columns in the FOR UPDATE OF *column* list. ♦

The principal advantage to specifying columns is documentation and preventing programming errors. (The database server refuses to update any other columns.) An additional advantage is improved performance, when the SELECT statement meets the following criteria:

- The SELECT statement can be processed using an index.
- The columns that are listed are not part of the index that is used to process the SELECT statement.

If the columns that you intend to update are part of the index that is used to process the SELECT statement, the database server keeps a list of each updated row, to ensure that no row is updated twice. If the OF keyword specifies which columns can be updated, the database server determines whether or not to keep the list of updated rows. If the database server determines that the work of keeping the list is unnecessary, performance improves. If you do not use the OF *column* list, the database server always maintains a list of updated rows, although the list might be unnecessary.

The following example contains ESQL/C code that uses an update cursor with a DELETE statement to delete the current row.

Whenever the row is deleted, the cursor remains between rows. After you delete data, you must use a FETCH statement to advance the cursor to the next row before you can refer to the cursor in a DELETE or UPDATE statement.

```
EXEC SQL declare q_curs cursor for
   select * from customer where lname matches :last_name for update;

EXEC SQL open q_curs;
for (;;)
{
   EXEC SQL fetch q_curs into :cust_rec;
   if (strncmp(SQLSTATE, "00", 2) != 0)
      break;
```

```
         /* Display customer values and prompt for answer */
         printf("\n%s %s", cust_rec.fname, cust_rec.lname);
         printf("\nDelete this customer? ");
         scanf("%s", answer);

         if (answer[0] == 'y')
            EXEC SQL delete from customer where current of q_curs;
         if (strncmp(SQLSTATE, "00", 2) != 0)
            break;
      }
      printf("\n");
      EXEC SQL close q_curs;
```

Locking with an Update Cursor

The FOR UPDATE keywords notify the database server that updating is possible and cause it to use more stringent locking than with a select cursor. You declare an update cursor to let the database server know that the program might update (or delete) any row that it fetches as part of the SELECT statement. The update cursor employs *promotable* locks for rows that the program fetches. Other programs can read the locked row, but no other program can place a promotable lock (also called a *write lock*). Before the program modifies the row, the row lock is promoted to an exclusive lock.

It is possible to declare an update cursor with the WITH HOLD keywords, but the only reason to do so is to break a long series of updates into smaller transactions. You must fetch and update a particular row in the same transaction.

If an operation involves fetching and updating a large number of rows, the lock table that the database server maintains can overflow. The usual way to prevent this overflow is to lock the entire table that is being updated. If this action is impossible, an alternative is to update through a hold cursor and to execute COMMIT WORK at frequent intervals. You must plan such an application carefully, however, because COMMIT WORK releases all locks, even those that are placed through a hold cursor.

Subset of INSERT Statement Associated with a Sequential Cursor

As indicated in the diagram for "DECLARE" on page 2-323, to create an insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

The following example contains ESQL/C code that declares an insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can flush the buffer (that is, to write its contents into the database) when it is less than full, using the CLOSE, FLUSH, or COMMIT WORK statements. This topic is discussed further under the CLOSE, FLUSH, and PUT statements.

You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly. For a complete description of INSERT syntax and usage, see "INSERT" on page 2-489.

Insert Cursor

When you associate an INSERT statement with a cursor, the cursor is called an *insert cursor*. An insert cursor is a data structure that represents the rows that the INSERT statement is to add to the database. The insert cursor simply inserts rows of data; it cannot be used to fetch data. To create an insert cursor, you associate a cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

Create an insert cursor if you want to add multiple rows to the database in an INSERT operation. An insert cursor allows bulk insert data to be buffered in memory and written to disk when the buffer is full, as these steps describe:

1. Use DECLARE to define an insert cursor for the INSERT statement.
2. Open the cursor with the OPEN statement. The database server creates the insert buffer in memory and positions the cursor at the first row of the insert buffer.
3. Copy successive rows of data into the insert buffer with the PUT statement.

4. The database server writes the rows to disk only when the buffer is full. You can use the CLOSE, FLUSH, or COMMIT WORK statement to flush the buffer when it is less than full. This topic is discussed further under the PUT and CLOSE statements.
5. Close the cursor with the CLOSE statement when the insert cursor is no longer needed. You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.
6. Free the cursor with the FREE statement. The FREE statement releases the resources that are allocated for an insert cursor.

Using an insert cursor is more efficient than embedding the INSERT statement directly. This process reduces communication between the program and the database server and also increases the speed of the insertions.

In addition to select and function cursors, insert cursors can also have the sequential cursor characteristic. To create an insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. (For more information, see "Insert Cursor" on page 2-331.) The following example contains IBM Informix ESQL/C code that declares a sequential insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

Cursor Characteristics

You can declare a cursor as a *sequential* cursor (the default), a *scroll* cursor (by using the SCROLL keyword), or a *hold* cursor (by using the WITH HOLD keywords). The SCROLL and WITH HOLD keywords are not mutually exclusive. Sections that follow explain these structural characteristics.

A select or function cursor can be either a sequential or a scroll cursor. An insert cursor can only be a sequential cursor. Select, function, and insert cursors can optionally be hold cursors.

Creating a Sequential Cursor by Default

If you use only the CURSOR keyword, you create a sequential cursor, which can fetch only the next row in sequence from the active set. The sequential cursor can read through the active set only once each time it is opened.

If you are using a sequential cursor for a select cursor, on each execution of the FETCH statement, the database server returns the contents of the current row and locates the next row in the active set.

The following example creates a read-only sequential cursor in a database that is not ANSI compliant and an update sequential cursor in an ANSI-compliant database:

```
EXEC SQL declare s_cur cursor for
   select fname, lname into :st_fname, :st_lname
   from orders where customer_num = 114;
```

In addition to select and function cursors, insert cursors can also have the sequential cursor characteristic. To create an insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. (For more information, see "Insert Cursor" on page 2-331.) The following example declares a sequential insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

Using the SCROLL Keyword to Create a Scroll Cursor

Use the SCROLL keyword to create a scroll cursor, which can fetch rows of the active set in any sequence.

The database server retains the active set of the cursor as a temporary table until the cursor is closed. You can fetch the first, last, or any intermediate rows of the active set as well as fetch rows repeatedly without having to close and reopen the cursor. (See FETCH.)

On a multiuser system, the rows in the tables from which the active-set rows were derived might change after the cursor is opened and a copy is made in the temporary table. If you use a scroll cursor within a transaction, you can prevent copied rows from changing either by setting the isolation level to Repeatable Read or by locking the entire table in share mode during the transaction. (See SET ISOLATION and LOCK TABLE.)

The following example creates a scroll cursor for a SELECT statement:

```
DECLARE sc_cur SCROLL CURSOR FOR SELECT * FROM orders
```

You can create scroll cursors for select and function cursors but *not* for insert cursors. Scroll cursors cannot be declared as FOR UPDATE.

Using the WITH HOLD Keywords to Create a Hold Cursor

Use the WITH HOLD keywords to create a hold cursor. A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors close at the end of a transaction. A hold cursor does not close; it remains open after a transaction ends.

A hold cursor can be either a sequential cursor or a scroll cursor.

You can use the WITH HOLD keywords to declare select and function cursors (sequential and scroll), and insert cursors. These keywords follow the CURSOR keyword in the DECLARE statement. The following example creates a sequential hold cursor for a SELECT:

```
DECLARE hld_cur CURSOR WITH HOLD FOR
   SELECT customer_num, lname, city FROM customer
```

You can use a select hold cursor as the following ESQL/C code example shows. This code fragment uses a hold cursor as a *master* cursor to scan one set of records and a sequential cursor as a *detail* cursor to point to records that are located in a different table. The records that the master cursor scans are the basis for updating the records to which the detail cursor points. The COMMIT WORK statement at the end of each iteration of the first WHILE loop leaves the hold cursor **c_master** open but closes the sequential cursor **c_detail** and releases all locks. This technique minimizes the resources that the database server must devote to locks and unfinished transactions, and it gives other users immediate access to updated rows.

```
EXEC SQL BEGIN DECLARE SECTION;
   int p_custnum, int save_status; long p_orddate;
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare st_1 from
   'select order_date from orders where customer_num = ? for update';
EXEC SQL declare c_detail cursor for st_1;
EXEC SQL declare c_master cursor with hold for
   select customer_num from customer where city = 'Pittsburgh';

EXEC SQL open c_master;
if(SQLCODE==0) /* the open worked */
   EXEC SQL fetch c_master into :p_custnum; /* discover first customer */
while(SQLCODE==0) /* while no errors and not end of pittsburgh customers */
    {
   EXEC SQL begin work; /* start transaction for customer p_custnum */
   EXEC SQL open c_detail using :p_custnum;
   if(SQLCODE==0) /* detail open succeeded */
      EXEC SQL fetch c_detail into :p_orddate; /* get first order */
   while(SQLCODE==0) /* while no errors and not end of orders */
        {
      EXEC SQL update orders set order_date = '08/15/94'
         where current of c_detail;
      if(status==0) /* update was ok */
         EXEC SQL fetch c_detail into :p_orddate; /* next order */
      }
   if(SQLCODE==SQLNOTFOUND) /* correctly updated all found orders */
      EXEC SQL commit work; /* make updates permanent, set status */
   else /* some failure in an update */
        {
      save_status = SQLCODE; /* save error for loop control */
      EXEC SQL rollback work;
      SQLCODE = save_status; /* force loop to end */
       }
   if(SQLCODE==0) /* all updates, and the commit, worked ok */
      EXEC SQL fetch c_master into :p_custnum; /* next customer? */
   }
EXEC SQL close c_master;
```

Use either the CLOSE statement to close the hold cursor explicitly or the CLOSE DATABASE or DISCONNECT statements to close it implicitly. The CLOSE DATABASE statement closes all cursors.

IDS

Releases earlier than Version 9.40 of Dynamic Server) do not support the PDQPRIORITY feature with cursors that were declared WITH HOLD. ♦

Using an Insert Cursor with Hold

If you associate a hold cursor with an INSERT statement, you can use transactions to break a long series of PUT statements into smaller sets of PUT statements. Instead of waiting for the PUT statements to fill the buffer and cause an automatic write to the database, you can execute a COMMIT WORK statement to flush the row buffer. With a hold cursor, COMMIT WORK commits the inserted rows but leaves the cursor open for further inserts. This method can be desirable when you are inserting a large number of rows, because pending uncommitted work consumes database server resources.

Subset of SELECT Statement Associated with Cursors

As indicated in the syntax diagram for "DECLARE" on page 2-323, not all SELECT statements can be associated with a read-only or update cursor.

If the DECLARE statement includes one of these options, you must observe certain restrictions on the SELECT statement that is included in the DECLARE statement (either directly or as a prepared statement).

If the DECLARE statement includes the FOR READ ONLY option, the SELECT statement cannot have a FOR READ ONLY nor FOR UPDATE option. (For a description of SELECT syntax and usage, see "SELECT" on page 2-581.)

If the DECLARE statement includes the FOR UPDATE option, the SELECT statement must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot include any of the following clauses or keywords: DISTINCT, FOR READ ONLY, FOR UPDATE, GROUP BY, INTO TEMP, ORDER BY, UNION, or UNIQUE.
- XPS In Extended Parallel Server, the statement cannot include the INTO EXTERNAL and INTO SCRATCH clauses. ♦

Examples of Cursors in Non-ANSI Compliant Databases

In a database that is not ANSI compliant, a cursor associated with a SELECT statement is a read-only cursor by default. The following example declares a read-only cursor in a non-ANSI compliant database:

```
EXEC SQL declare cust_curs cursor for
   select * from customer_notansi;
```

If you want to make it clear in the program code that this cursor is a read-only cursor, specify the FOR READ ONLY option as the following example shows:

```
EXEC SQL declare cust_curs cursor for
   select * from customer_notansi for read only;
```

If you want this cursor to be an update cursor, specify the FOR UPDATE option in your DECLARE statement. This example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
   select * from customer_notansi for update;
```

If you want an update cursor to be able to modify only some columns in a table, you must specify those columns in the FOR UPDATE clause. The following example declares an update cursor that can update only the **fname** and **lname** columns in the **customer_notansi** table:

```
EXEC SQL declare name_curs cursor for
   select * from customer_notansi for update of fname, lname;
```

ANSI

Examples of Cursors in ANSI-Compliant Databases

In an ANSI-compliant database, a cursor associated with a SELECT statement is an update cursor by default.

The following example declares an update cursor in an ANSI-compliant database:

```
EXEC SQL declare x_curs cursor for select * from customer_ansi;
```

To make it clear in the program documentation that this cursor is an update cursor, you can specify the FOR UPDATE option as in this example:

```
EXEC SQL declare x_curs cursor for
   select * from customer_ansi for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you must specify these columns in the FOR UPDATE option. The following example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_ansi** table:

```
EXEC SQL declare y_curs cursor for
   select * from customer_ansi for update of fname, lname;
```

If you want a cursor to be a read-only cursor, you must override the default behavior of the DECLARE statement by specifying the FOR READ ONLY option in your DECLARE statement. The following example declares a read-only cursor:

```
EXEC SQL declare z_curs cursor for
   select * from customer_ansi for read only;
```

Associating a Cursor with a Prepared Statement

The PREPARE statement lets you assemble the text of an SQL statement at runtime and pass the statement text to the database server for execution. If you anticipate that a dynamically prepared SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that returns values could produce more than one row of data, the prepared statement must be associated with a cursor. (See PREPARE.)

The result of a PREPARE statement is a statement identifier (*statement id* or *id variable*), which is a data structure that represents the prepared statement text. To declare a cursor for the statement text, associate a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement. You cannot associate a scroll cursor with a prepared INSERT statement or with a SELECT statement that was prepared to include a FOR UPDATE clause.

After a cursor is opened, used, and closed, a different statement can be prepared under the same statement identifier. In this way, it is possible to use a single cursor with different statements at different times. The cursor must be redeclared before you use it again.

The following example contains ESQL/C code that prepares a SELECT statement and declares a sequential cursor for the prepared statement text. The statement identifier **st_1** is first prepared from a SELECT statement that returns values; then the cursor **c_detail** is declared for **st_1**.

```
EXEC SQL prepare st_1 from
   'select order_date
      from orders where customer_num = ?';
EXEC SQL declare c_detail cursor for st_1;
```

If you want to use a prepared SELECT statement to modify data, add a FOR UPDATE clause to the statement text that you want to prepare, as the following ESQL/C example shows:

```
EXEC SQL prepare sel_1 from
   'select * from customer for update';
EXEC SQL declare sel_curs cursor for sel_1;
```

IDS

Associating a Cursor with a Collection Variable

The DECLARE statement allows you to declare a cursor for an ESQL/C collection variable. Such a cursor is called a *collection cursor*. You use a collection variable to access the elements of a collection (SET, MULTISET, LIST) column. Use a cursor when you want to access one or more elements in a collection variable.

The Collection-Derived Table segment identifies the collection variable for which to declare the cursor. For more information, see "Collection-Derived Table" on page 4-7.

Select with a Collection-Derived Table

The diagram for "DECLARE" on page 2-323 refers to this section.

To declare a select cursor for a collection variable, include the Collection-Derived Table segment with the SELECT statement that you associate with the collection cursor. A select cursor allows you to select one or more elements *from* the collection variable. (For a description of SELECT syntax and usage, see "SELECT" on page 2-581.)

When you declare a select cursor for a collection variable, the DECLARE statement has the following restrictions:

- It cannot include the FOR READ ONLY keywords as cursor mode.

  The select cursor is an update cursor.
- It cannot include the SCROLL or WITH HOLD keywords.

  The select cursor must be a sequential cursor.

In addition, the SELECT statement that you associate with the collection cursor has the following restrictions:

- It cannot include the following clauses or options: WHERE, GROUP BY, ORDER BY, HAVING, INTO TEMP, and WITH REOPTIMIZATION.
- It cannot contain expressions in the select list.
- If the collection contains elements of opaque, distinct, built-in, or other collection data types, the select list must be an asterisk (*).
- Column names in the select list must be simple column names.

  These columns cannot use the following syntax:

  ```
  database@server:table.column --INVALID SYNTAX
  ```
- It *must* specify the name of the collection variable in the FROM clause.

  You cannot specify an input parameter (the question-mark (?) symbol) for the collection variable. Likewise you cannot use the virtual table format of the Collection-Derived Table segment.

Using a SELECT Cursor with a Collection Variable

A collection cursor that includes a SELECT statement with the Collection-Derived Table clause provides access to the elements in a collection variable.

To select more than one element

1. Create a client collection variable in your ESQL/C program.
2. Declare the collection cursor for the SELECT statement with the DECLARE statement.

   To modify elements of the collection variable, declare the select cursor as an update cursor with the FOR UPDATE keywords. You can then use the WHERE CURRENT OF clause of the DELETE and UPDATE statements to delete or update elements of the collection.

3. Open this cursor with the OPEN statement.
4. Fetch the elements from the collection cursor with the FETCH statement and the INTO clause.
5. If necessary, perform any updates or deletes on the fetched data and save the modified collection variable in the collection column.

   Once the collection variable contains the correct elements, use the UPDATE or INSERT statement to save the contents of the collection variable in the actual collection column (SET, MULTISET, or LIST).
6. Close the collection cursor with the CLOSE statement.

This DECLARE statement declares a select cursor for a collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL declare set_curs cursor for select * from table(:a_set);
```

For an extended code example that uses a collection cursor for a SELECT statement, see "Fetching from a Collection Cursor" on page 2-432.

Insert with a Collection-Derived Table

To declare an insert cursor for a collection variable, include the Collection-Derived Table segment in the INSERT statement associated with the collection cursor. An insert cursor can insert one or more elements in the collection. For a description of INSERT syntax and usage, see "INSERT" on page 2-489.

The insert cursor must be a sequential cursor. That is, the DECLARE statement cannot specify the SCROLL keyword.

When you declare an insert cursor for a collection variable, the Collection-Derived Table clause of the INSERT statement *must* contain the name of the collection variable. You cannot specify an input parameter (the question-mark (?) symbol) for the collection variable. However, you can use an input parameter in the VALUES clause of the INSERT statement. This parameter indicates that the collection element is to be provided later by the FROM clause of the PUT statement.

A collection cursor that includes an INSERT statement with the Collection-Derived Table clause allows you to insert more than one element into a collection variable.

To insert more than one element

1. Create a client collection variable in your ESQL/C program.
2. Declare the collection cursor for the INSERT statement with the DECLARE statement.
3. Open the cursor with the OPEN statement.
4. Put the elements into the collection cursor with the PUT statement and the FROM clause.
5. Once the collection variable contains all the elements, use the UPDATE statement or the INSERT statement on a table name to save the contents of the collection variable in a collection column (SET, MULTISET, or LIST).
6. Close the collection cursor with the CLOSE statement.

This example declares an insert cursor for the **a_set** collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(smallint not null) a_mset;
   int an_element;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL declare mset_curs cursor for
   insert into table(:a_mset) values (?);
EXEC SQL open mset_curs;
while (1)
{
...
   EXEC SQL put mset_curs from :an_element;
...
}
```

To insert the elements into the collection variable, use the PUT statement with the FROM clause. For a code example that uses a collection cursor for an INSERT statement, see "Inserting into a Collection Cursor" on page 2-544.

Using Cursors with Transactions

To roll back a modification, you must perform the modification within a transaction. A transaction in a database that is not ANSI compliant begins only when the BEGIN WORK statement is executed.

ANSI

In an ANSI-compliant database, transactions are always in effect. ♦

The database server enforces these guidelines for select and update cursors to ensure that modifications can be committed or rolled back properly:

- Open an insert or update cursor within a transaction.
- Include PUT and FLUSH statements within one transaction.
- Modify data (update, insert, or delete) within one transaction.

The database server lets you open and close a hold cursor for an update outside a transaction; however, you should fetch all the rows that pertain to a given modification and then perform the modification all within a single transaction. You cannot open and close a hold cursor or an update cursor outside a transaction.

The following example uses an update cursor within a transaction:

```
EXEC SQL declare q_curs cursor for
   select customer_num, fname, lname from customer
   where lname matches :last_name for update;
EXEC SQL open q_curs;
EXEC SQL begin work;
EXEC SQL fetch q_curs into :cust_rec; /* fetch after begin */
EXEC SQL update customer set lname = 'Smith'
   where current of q_curs;
/* no error */
EXEC SQL commit work;
```

When you update a row within a transaction, the row remains locked until the cursor is closed or the transaction is committed or rolled back. If you update a row when no transaction is in effect, the row lock is released when the modified row is written to disk. If you update or delete a row outside a transaction, you cannot roll back the operation.

In a database that uses transactions, you cannot open an insert cursor outside a transaction unless it was also declared with the WITH HOLD keywords.

Related Information

Related statements: CLOSE, DELETE, EXECUTE PROCEDURE, FETCH, FREE, INSERT, OPEN, PREPARE, PUT, SELECT, and UPDATE

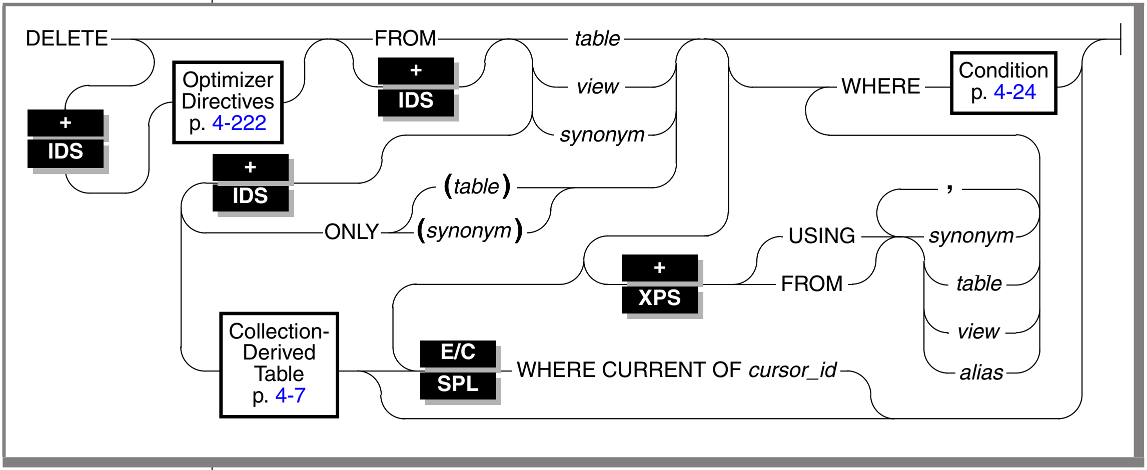
For discussions of cursors and data modification, see the *IBM Informix Guide to SQL: Tutorial*.

For more advanced issues related to cursors or using cursors with collection variables, see the *IBM Informix ESQL/C Programmer's Manual*.

DELETE

Use the DELETE statement to delete one or more rows from a table, or one or more elements in an SPL or ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
alias	Temporary name for a table	You cannot use an alias for an indexed table	Identifier, p. 4-189
cursor_id	Previously declared cursor	Must have been declared FOR UPDATE.	Identifier, p. 4-189
synonym, *table*, *view*	Table, view, or synonym with row(s) to be deleted	The *table* or *view* (or the *synonym* and the table or view to which it points) must exist	Database Object Name, p. 4-46

Usage

If you use DELETE without a WHERE clause (to specify either a condition or the active set of the cursor), all rows in the table are deleted.

In a database with explicit transaction logging, any DELETE statement that you execute outside a transaction is treated as a single transaction.

If you specify a *view* name, the view must be updatable. For an explanation of an updatable view, see "Updating Through Views" on page 2-315.

The database server locks each row affected by a DELETE statement within a transaction for the duration of the transaction. The type of lock that the database server uses is determined by the lock mode of the table, as set by a CREATE TABLE or ALTER TABLE statement, as follows:

- If the lock mode is ROW, the database server acquires one lock for each row affected by the delete.

XPS

- In Extended Parallel Server, if the lock mode is PAGE, the database server acquires one lock for each page affected by the delete. ♦

If the number of rows affected is very large and the lock mode is ROW, you might exceed the limits your operating system places on the maximum number of simultaneous locks. If this occurs, you can either reduce the scope of the DELETE statement or lock the table in exclusive mode before you execute the statement.

DB

If you omit the WHERE clause while working at the SQL menu, DB-Access prompts you to verify that you want to delete all rows from a table. You do not receive a prompt if you run execute DELETE within a command file. ♦

ANSI

In an ANSI-compliant database, data manipulation language (DML) statements are always in a transaction. You cannot execute a DELETE statement outside a transaction. ♦

IDS

On Dynamic Server, the FROM keyword that immediately follows DELETE can be omitted if the **DELIMIDENT** environment variable has been set. ♦

IDS

Using the ONLY Keyword

If you use DELETE to remove rows of a supertable, rows from both the supertable and its subtables can be deleted. To delete rows from the supertable only, specify the ONLY keyword before the table name.

```
DELETE FROM ONLY(super_tab)
   WHERE name = "johnson"
```

Warning: *If you use the DELETE statement on a supertable and omit the ONLY keyword and WHERE clause, all rows of the supertable and its subtables are deleted.*

You cannot specify the ONLY keyword if you plan to use the WHERE CURRENT OF clause to delete the current row of the active set of a cursor.

Considerations When Tables Have Cascading Deletes

When you use the ON DELETE CASCADE option of the REFERENCES clause of either the CREATE TABLE or ALTER TABLE statement, you specify that you want deletes to cascade from one table to another. For example, in the **stores_demo** database, the **stock** table contains the column **stock_num** as a primary key. The **catalog** and **items** tables each contain the column **stock_num** as foreign keys with the ON DELETE CASCADE option specified. When a delete is performed from the **stock** table, rows are also deleted in the **catalog** and **items** tables, which are referenced through the foreign keys.

To have DELETE actions cascade to a table that has a referential constraint on a parent table, you need the Delete privilege only on the parent table that you reference in the DELETE statement.

If a DELETE without a WHERE clause is performed on a table that one or more child tables reference with cascading deletes, the database server deletes all rows from that table and from any affected child tables.

For an example of how to create a referential constraint that uses cascading deletes, see "Using the ON DELETE CASCADE Option" on page 2-226.

Restrictions on DELETE When Tables Have Cascading Deletes

You cannot use a child table in a correlated subquery to delete a row from a parent table. If two child tables reference the same parent table, and one child specifies cascading deletes but the other child does not, then if you attempt to delete a row that applies to both child tables from the parent table, the DELETE fails, and no rows are deleted from the parent or child tables.

Locking and Logging Implications of Cascading Deletes

During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables.

IDS

Transaction logging is required for cascading deletes. If logging is turned off in a database that is not ANSI-compliant, even temporarily, deletes do not cascade, because you cannot roll back any actions. For example, if a parent row is deleted, but the system fails before the child rows are deleted, the database will have dangling child records, in violation of referential integrity. After logging is turned back on, however, subsequent deletes cascade. ♦

Using the WHERE Keyword to Introduce a Condition

Use the WHERE *condition* clause to specify which rows you want to delete from the table. The *condition* after the WHERE keyword is equivalent to the *condition* in the SELECT statement. For example, the next statement deletes all the rows of the **items** table where the order number is less than 1034:

```
DELETE FROM items WHERE order_num < 1034
```

DB

If you include a WHERE clause that selects all rows in the table, DB-Access gives no prompt and deletes all rows. ♦

IDS

If you are deleting from a supertable in a table hierarchy, a subquery in the WHERE clause cannot reference a subtable.

When deleting from a subtable, a subquery in the WHERE clause can reference the supertable only in SELECT…FROM ONLY (*supertable*)... syntax. ♦

E/C

SPL

Using the WHERE CURRENT OF Keywords

The WHERE CURRENT OF clause deletes the current row of the active set of a cursor. When you include this clause, the DELETE statement removes the row of the active set at the current position of the cursor. After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement.

You access the current row of the active set of a cursor with an update cursor. Before you can use the WHERE CURRENT OF clause, you must first create an update cursor by using the FOREACH statement (SPL) or the DECLARE statement with the FOR UPDATE clause (ESQL/C).

ANSI

Unless they are declared with the FOR READ ONLY keywords, all select cursors are potentially update cursors in an ANSI-compliant database. You can use the WHERE CURRENT OF clause with any select cursor that was not declared with the FOR READ ONLY keywords. ♦

IDS

You cannot use WHERE CURRENT OF if you are selecting from only one table in a table hierarchy. That is, this clause is not valid with the ONLY keyword.

The WHERE CURRENT OF clause can be used to delete an element from a collection by deleting the current row of the collection-derived table that a collection variable holds. For more information, see "Collection-Derived Table" on page 4-7. ♦

+

XPS

Using the USING or FROM Keyword to Introduce a Join Condition

To delete information from a table based on information contained in one or more other tables, use the USING keyword or a second FROM keyword to introduce the list of tables that you want to join in the WHERE clause. When you use this syntax, the WHERE clause can include any complex join.

If you do not list a join in the WHERE clause, the database server ignores the tables listed after the introductory keyword (either USING or FROM). That is, the query performs as if the list of tables was not included.

You can use a second FROM keyword to introduce the list of tables, but your code will be easier to read if you use the USING keyword instead.

E/C

When you use a delete join, the entire operation occurs as a single transaction. For example, if a delete join query is supposed to delete 100 rows and an error occurs after the 50th row, the first 50 rows that are already deleted will reappear in the table. ♦

When you introduce a list of tables that you want to join in the WHERE clause, the following restrictions for the DELETE statement exist:

- You must list the target table (the one from which rows are to be deleted) and any table that will be part of a join after the USING (or second FROM) keyword.
- The WHERE clause cannot contain outer joins.
- The target table cannot be a static or external table.
- The statement cannot contain cursors.
- If the target table is a view, the view must be updatable.

  That implies that the SELECT statement that defines the view cannot contain any of the following syntax elements:

  - Aggregate expressions
  - UNIQUE or DISTINCT keywords
  - UNION operator
  - GROUP BY keywords

The next example deletes the rows from the **lineitem** table whose corresponding rows in the **order** table show a **qty** of less than one.

```
DELETE FROM lineitem USING order o, lineitem l
   WHERE o.qty < 1 AND o.order_num = l.order_num
```

A delete join makes it easier to incorporate new data into a database. For example, you can:

1. Store new values in a temporary table.
2. Use a delete join (DELETE...USING statement) to remove any records from the temporary table that already exist in the table into which you want to insert the new records.
3. Insert the remaining records into the table.

In addition, you can use this syntax instead of deleting from the results of a SELECT statement that includes a join.

IDS

Deleting Rows That Contain Opaque Data Types

Some opaque data types require special processing when they are deleted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

To accomplish this process, call a user-defined support function called **destroy( )**. When you use DELETE to remove a row that contains one of these opaque types, the database server automatically invokes **destroy( )** for the opaque type. This function decides how to remove the data, regardless of where it is stored. For more information on the **destroy( )** support function, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Deleting Rows That Contain Collection Data Types

When a row contains a column that is a collection data type (LIST, MULTISET, or SET), you can search for a particular element in the collection, and delete the row or rows in which the element is found.

For example, the following statement deletes any rows from the **new_tab** table in which the **set_col** column contains the element `jimmy smith`:

```
DELETE FROM new_tab WHERE 'jimmy smith' IN set_col
```

E/C

SPL

You can also use a collection variable to delete values in a collection column by deleting one or more individual elements in a collection. For more information, see "Collection-Derived Table" on page 4-7, and the examples in "Example of Deleting from a Collection in ESQL/C" on page 4-17 and "Example of Deleting from a Collection" on page 4-18. ♦

ANSI

SQLSTATE Values in an ANSI-Compliant Database

If no rows satisfy the WHERE clause of a DELETE operation on a table in an ANSI-compliant database, the database server issues a warning. You can detect this warning condition in either of the following ways:

- The GET DIAGNOSTICS statement sets the **RETURNED_SQLSTATE** field to the value `02000`. In an SQL API application, the **SQLSTATE** variable contains this same value.
- In an SQL API application, the **sqlca.sqlcode** and **SQLCODE** variables contain the value `100`.

The database server also sets **SQLSTATE** and **SQLCODE** to these values if the DELETE . . . WHERE statement is part of a multistatement prepared object, and the database server returns no rows.

SQLSTATE Values in a Database That Is Not ANSI-Compliant

In a database that is not ANSI compliant, the database server does not return a warning when it finds no rows satisfying the WHERE clause of a DELETE statement. In this case, the **SQLSTATE** code is `00000` and the **SQLCODE** code is zero (`0`). If the DELETE . . . WHERE is part of a multistatement prepared object, however, and no rows are returned, the database server does issue a warning. It sets **SQLSTATE** to `02000` and sets the **SQLCODE** value to `100`.

Related Information

Related Statements: DECLARE, FETCH, GET DIAGNOSTICS, INSERT, OPEN, SELECT, and UPDATE

For discussions of the DELETE statement, SPL routines, statement modification, cursors, and the **SQLCODE** code, see the *IBM Informix Guide to SQL: Tutorial*.

For information on how to access row and collections with ESQL/C host variables, see the chapter on complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

For a discussion of the GLS aspects of the DELETE statement, see the *IBM Informix GLS User's Guide*.

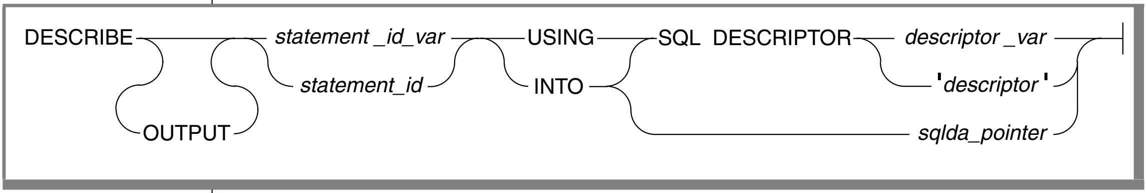
+

E/C

DESCRIBE

Use the DESCRIBE statement to obtain information about output parameters and other features of a prepared statement before you execute it. Use this statement with ESQL/C. (See also "DESCRIBE INPUT" on page 2-359.)

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Name of a system-descriptor area	System-descriptor area must already be allocated	Quoted String, p. 4-243
descriptor_var	Host variable that identifies a system-descriptor area	Must contain the name of an allocated system-descriptor area	Language-specific rules for names
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with dollar ($) sign nor colon (:). An **sqlda** structure is required if dynamic SQL is used.	See the **sqlda** structure in the *IBM Informix ESQL/C Programmer's Manual*
statement_id	Statement identifier for a prepared SQL statement	Must be defined in a previous PREPARE statement	PREPARE, p. 2-527; Identifier, p. 4-189
statement _id_var	Host variable that contains the value of *statement_id*	Must be declared in a previous PREPARE statement	Language-specific rules for names

Usage

DESCRIBE can provide information at runtime about a prepared statement:

- The type of SQL statement that was prepared.
- Whether an UPDATE or DELETE statement contains a WHERE clause.

IDS

- For a SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE), INSERT, or UPDATE statement, the DESCRIBE statement also returns the number, data types, and size of the values, and the name of the column or expression that the query returns. ♦

XPS

- For a SELECT, EXECUTE PROCEDURE, or INSERT statement, DESCRIBE also returns the number, data types, and size of the values, and the name of the column or expression that the query returns. ♦

With this information, you can write code to allocate memory to hold retrieved values and display or process them after they are fetched.

The OUTPUT Keyword

The OUTPUT keyword specifies that only information about output parameters of the prepared statement are stored in the **sqlda** descriptor area. If you omit this keyword, DESCRIBE can return input parameters, but only for INSERT statements (and for UPDATE, if the **IFX_UPDESC** environment variable is set in the environment where the database server is initialized).

Describing the Statement Type

The DESCRIBE statement takes a statement identifier from a PREPARE statement as input. When the DESCRIBE statement executes, the database server sets the value of the SQLCODE field of the **sqlca** to indicate the statement type (that is, the keyword with which the statement begins). If the prepared statement text contains more than one SQL statement, the DESCRIBE statement returns the type of the first statement in the text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer. You can test the number against the constant names that are defined. In ESQL/C, the constant names are defined in the **sqlstypes.h** header file.

The DESCRIBE statement (and the DESCRIBE INPUT statement) use the SQLCODE field differently from any other statement, possibly returning a nonzero value when it executes successfully. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for the Existence of a WHERE Clause

If the DESCRIBE statement detects that a prepared statement contains an UPDATE or DELETE statement without a WHERE clause, the DESCRIBE statement sets the **sqlca.sqlwarn.sqlwarn4** variable to `W`.

When you do not specify a WHERE clause in either a DELETE or UPDATE statement, the database server performs the delete or update operation on the entire table. Check the **sqlca.sqlwarn.sqlwarn4** variable to avoid unintended global changes to your table.

Describing a Statement with Runtime Parameters

If the prepared statement contains parameters for which the number of parameters or parameter data types is to be supplied at runtime, you can describe these input values. If the prepared statement text includes one of the following statements, the DESCRIBE statement returns a description of each column or expression that is included in the list:

- EXECUTE FUNCTION (or EXECUTE PROCEDURE)
- INSERT
- SELECT (without an INTO TEMP clause)
- UPDATE

  IDS

  The **IFX_UPDDESC** environment variable must be set before you can usn DESCRIBE to obtain information about an UPDATE statement. For more information, see the *IBM Informix Guide to SQL: Reference*. ♦

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression

For a prepared INSERT or UPDATE statement, DESCRIBE returns only the dynamic parameters (those expressed with a question mark (?) symbol). Using the OUTPUT keyword, however, prevents these from being returned.

You can specify a destination for the returned informations as a new or existing system-descriptor area, or as a pointer to an **sqlda** structure.

X/O

A system-descriptor area conforms to the X/Open standards. ♦

Using the SQL DESCRIPTOR Keywords

Use the USING SQL DESCRIPTOR clause to store the description of a prepared statement list in a previously allocated system-descriptor area.

Use the INTO SQL DESCRIPTOR clause to create a new system-descriptor structure and store the description of a statement list in that structure.

To describe one of the previously mentioned statements into a system-descriptor area, DESCRIBE updates the system-descriptor area in these ways:

- It sets the COUNT field in the system-descriptor area to the number of values in the statement list. An error results if COUNT is greater than the number of item descriptors in the system-descriptor area.
- It sets the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields in the system-descriptor area.

  IDS

  If the column has an opaque data type, the database server sets the EXTYPEID, EXTYPENAME, EXTYPELENGTH, EXTYPEOWNER-LENGTH, and EXTYPEOWNERNAME fields of the item descriptor. ♦
- It allocates memory for the DATA field for each item descriptor, based on the TYPE and LENGTH information.

After a DESCRIBE statement is executed, the SCALE and PRECISION fields contain the scale and precision of the column, respectively. If SCALE and PRECISION are set in the SET DESCRIPTOR statement, and TYPE is set to DECIMAL or MONEY, the LENGTH field is modified to adjust for the scale and precision of the decimal value. If TYPE is not set to DECIMAL or MONEY, the values for SCALE and PRECISION are not set, and LENGTH is unaffected.

You must modify system-descriptor-area information with SET DESCRIPTOR statements to show the address in memory that is to receive the described value. You can change the data type to another compatible type. This change causes data conversion to take place when data values are fetched.

You can use the system-descriptor area in prepared statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT.

The following examples show the use of a system descriptor in a DESCRIBE statement. In the first example, the system descriptor is a quoted string; in the second example, it is an embedded variable name.

```
main()
{
. . .
EXEC SQL allocate descriptor 'desc1' with max 3;
EXEC SQL prepare curs1 FROM 'select * from tab';
EXEC SQL describe curs1 using sql descriptor 'desc1';
}
EXEC SQL describe curs1 using sql descriptor :desc1var;
```

Using the INTO sqlda Pointer Clause

Use the INTO *sqlda_pointer* clause to allocate memory for an **sqlda** structure and store its address in an **sqlda** pointer. The DESCRIBE statement fills in the allocated memory with descriptive information. Unlike the USING clause, the INTO clause creates new **sqlda** structures to store the output from DESCRIBE.

The DESCRIBE statement sets the **sqlda.sqld** field to the number of values in the statement list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the statement list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

IDS

If the column has an opaque data type, DESCRIBE...INTO sets the **sqlxid**, **sqltypename**, **sqltypelen**, **sqlownerlen**, and **sqlownername** fields of the item descriptor. ♦

The DESCRIBE statement allocates memory for an **sqlda** pointer once it is declared in a program. The application program, however, must designate the storage area of the **sqlda.sqlvar.sqldata** fields.

IDS

Describing a Collection Variable

The DESCRIBE statement can provide information about a collection variable when you use the USING SQL DESCRIPTOR or INTO clause.

You must perform the DESCRIBE statement *after* you open the select or insert cursor. Otherwise, DESCRIBE cannot get information about the collection variable because it is the OPEN...USING statement that specifies the name of the collection variable to use.

The next ESQL/C code fragment dynamically selects the elements of the **:a_set** collection variable into a system-descriptor area called **desc1**:

```
EXEC SQL BEGIN DECLARE SECTION;
      client collection a_set;
      int i, set_count;
      int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from
      'select * from table(?)'

EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';

do
{
      EXEC SQL fetch set_curs using sql descriptor 'desc1';
      ...
      EXEC SQL get descriptor 'desc1' :set_count = count;
      for (i = 1; i <= set_count; i++)
      {
         EXEC SQL get descriptor 'desc1' value :i
            :element_type = TYPE;
         switch
         {
            case SQLINTEGER:
               EXEC SQL get descriptor 'desc1' value :i
                  :element_value = data;
            ...
         } /* end switch */
      } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
EXEC SQL deallocate descriptor 'desc1';
```

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE INPUT, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR

For a task-oriented discussion of the DESCRIBE statement, see the *IBM Informix Guide to SQL: Tutorial*.

For more information about how to use a system-descriptor area and **sqlda**, refer to the *IBM Informix ESQL/C Programmer's Manual*.

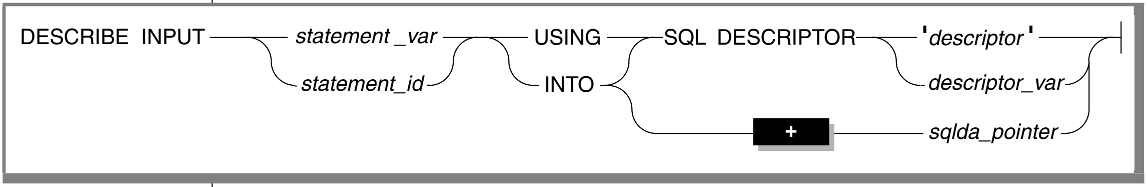
IDS

E/C

DESCRIBE INPUT

Use the DESCRIBE INPUT statement to return input parameter information before a prepared statement is executed. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Name of a system-descriptor area	System-descriptor area must already be allocated	Quoted String, p. 4-243
descriptor_var	Host variable that identifies a system-descriptor area	Must contain the name of an allocated system-descriptor area	Language-specific rules for names
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with dollar ($) sign or colon (:). An **sqlda** structure is required if dynamic SQL is used	See the **sqlda** structure in the *IBM Informix ESQL/C Programmer's Manual*.
statement_id	Statement identifier for a prepared SQL statement	Must be defined in a previously executed PREPARE statement	PREPARE, p. 2-527; Identifier, p. 4-189
statement_var	Host variable that contains the value of *statement_id*	Variable and *statement_id* both must be declared	Language-specific rules for names

Usage

The DESCRIBE INPUT and the DESCRIBE OUTPUT statements can return information about a prepared statement to an SQL Descriptor Area (**sqlda**):

- For a SELECT, EXECUTE FUNCTION (or PROCEDURE), INSERT, or UPDATE statement, the DESCRIBE statement (with no INPUT keyword) returns the number, data types, and size of the returned values, and the name of the column or expression.
- For a SELECT, EXECUTE FUNCTION, EXECUTE PROCEDURE, DELETE, INSERT, or UPDATE statement, the DESCRIBE INPUT statement returns all the input parameters of a prepared statement.

Tip: *Dynamic Server versions earlier than 9.40 do not support the INPUT keyword. For compatibility with legacy applications, DESCRIBE without INPUT is supported. In new applications, you should use DESCRIBE INPUT statements to provide information about dynamic parameters in the WHERE clause, in subqueries, and in other syntactic contexts where the old form of DESCRIBE cannot provide information.*

With this information, you can write code to allocate memory to hold retrieved values that you can display or process after they are fetched.

The **IFX_UPDDESC** environment variable does not need to be set before you can use DESCRIBE INPUT to obtain information about an UPDATE statement.

Describing the Statement Type

This statement takes a statement identifier from a PREPARE statement as input. After DESCRIBE INPUT executes, the SQLCODE field of the **sqlca** indicates the statement type (that is, the keyword with which the statement begins). If a prepared object contains more than one SQL statement, DESCRIBE INPUT returns the type of the first statement in the prepared text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer. You can compare the number with the named constants that are defined in the **sqlstypes.h** header file.

The DESCRIBE and DESCRIBE INPUT statements use SQLCODE differently from other statements, under some circumstances returning a nonzero value after successful execution. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for Existence of a WHERE Clause

If the DESCRIBE INPUT statement detects that a prepared object contains an UPDATE or DELETE statement without a WHERE clause, the database server sets the **sqlca.sqlwarn.sqlwarn4** variable to `W`.

When you do not specify a WHERE clause in either a DELETE or UPDATE statement, the database server performs the delete or update action on the entire table. Check the **sqlca.sqlwarn.sqlwarn4** variable after DESCRIBE INPUT executes to avoid unintended global changes to your table.

Describing a Statement with Dynamic Runtime Parameters

If the prepared statement specifies a set of parameters whose cardinality or data types must be supplied at runtime, you can describe these input values. If the prepared statement text includes one of the following statements, the DESCRIBE INPUT statement returns a description of each column or expression that is included in the list:

- EXECUTE FUNCTION (or EXECUTE PROCEDURE)
- INSERT or SELECT
- UPDATE or DELETE

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression
- Information about *dynamic parameters* (parameters that are expressed as question (?) mark symbols within the prepared statement).

If the database server cannot infer the data type of an expression parameter, the DESCRIBE INPUT statement returns SQLUNKNOWN as the data type.

You can specify a destination for the returned informations as a new or existing system-descriptor area, or as a pointer to an **sqlda** structure.

Using the SQL DESCRIPTOR Keywords

Specify INTO SQL DESCRIPTOR to create a new system-descriptor structure and store the description of a prepared statement list in that structure.

Use the USING SQL DESCRIPTOR clause to store the description of a prepared statement list in a previously allocated system-descriptor area. Executing the DESCRIBE INPUT . . . USING SQL DESCRIPTOR statement updates an existing system-descriptor area in the following ways:

- It allocates memory for the DATA field for each item descriptor, based on the TYPE and LENGTH information.
- It sets the COUNT field in the system-descriptor area to the number of values in the statement list. An error results if COUNT is greater than the number of item descriptors in the system-descriptor area.
- It sets the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields in the system-descriptor area.

For columns of opaque data types, the DESCRIBE INPUT statement sets the EXTYPEID, EXTYPENAME, EXTYPELENGTH, EXTYPEOWNERLENGTH, and EXTYPEOWNERNAME fields of the item descriptor.

After a DESCRIBE INPUT statement executes, the SCALE and PRECISION fields contain the scale and precision of the column, respectively. If SCALE and PRECISION are set in the SET DESCRIPTOR statement, and TYPE is set to DECIMAL or MONEY, the LENGTH field is modified to adjust for the decimal scale and precision. If TYPE is not set to DECIMAL or MONEY, the values for SCALE and PRECISION are not set, and LENGTH is unaffected.

You must modify the system-descriptor-area information with the SET DESCRIPTOR statement to specify the address in memory that is to receive the described value. You can change the data type to another compatible type. This causes data conversion to take place when the data values are fetched.

You can also use the system-descriptor area in other statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT.

The following examples show the use of a system descriptor in a DESCRIBE statement. In the first example, the system descriptor is a quoted string; in the second example, it is an embedded variable name.

```
main()
{
. . .
EXEC SQL allocate descriptor 'desc1' with max 3;
EXEC SQL prepare curs1 FROM 'select * from tab';
EXEC SQL describe curs1 using sql descriptor 'desc1';
}
EXEC SQL describe curs1 using sql descriptor :desc1var;
```

X/O

A system-descriptor area conforms to the X/Open standards. ♦

Using the INTO sqlda Pointer Clause

The INTO *sqlda_pointer* clause allocates memory for an **sqlda** structure and store its address in an **sqlda** pointer. The DESCRIBE INPUT statement fills in the allocated memory with descriptive information.

The DESCRIBE INPUT statement sets the **sqlda.sqld** field to the number of values in the statement list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the statement list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

If the column has an opaque data type, DESCRIBE INPUT . . . INTO sets the **sqlxid**, **sqltypename**, **sqltypelen**, **sqlownerlen**, and **sqlownername** fields of the item descriptor. ♦

The DESCRIBE INPUT statement allocates memory for an **sqlda** pointer once it is declared in a program. The application program, however, must designate the storage area of the **sqlda.sqlvar.sqldata** fields.

Describing a Collection Variable

The DESCRIBE INPUT statement can provide information about a collection variable if you use the INTO or USING SQL DESCRIPTOR clause.

You must execute the DESCRIBE INPUT statement *after* you open the select or insert cursor. Otherwise, DESCRIBE INPUT cannot get information about the collection variable because it is the OPEN . . . USING statement that specifies the name of the collection variable to use.

The next ESQL/C program fragment dynamically selects the elements of the **:a_set** collection variable into a system-descriptor area called **desc1**:

```
EXEC SQL BEGIN DECLARE SECTION;
      client collection a_set;
      int i, set_count;
      int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from
      'select * from table(?)'


EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';

do
{
      EXEC SQL fetch set_curs using sql descriptor 'desc1';
      ...
      EXEC SQL get descriptor 'desc1' :set_count = count;
      for (i = 1; i <= set_count; i++)
      {
         EXEC SQL get descriptor 'desc1' value :i
            :element_type = TYPE;
         switch
         {
            case SQLINTEGER:
               EXEC SQL get descriptor 'desc1' value :i
                  :element_value = data;
            ...
         } /* end switch */
      } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
EXEC SQL deallocate descriptor 'desc1';
```

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR

For a task-oriented discussion of the DESCRIBE statement, see the *IBM Informix Guide to SQL: Tutorial*.

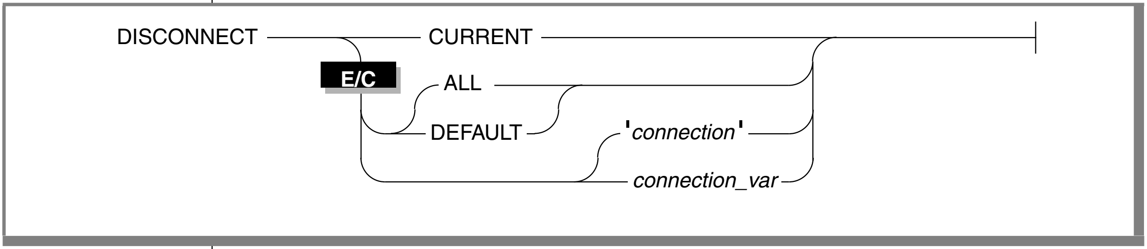
For more information about how to use a system-descriptor area and **sqlda**, refer to the *IBM Informix ESQL/C Programmer's Manual*.

+

DISCONNECT

Use the DISCONNECT statement to terminate a connection between an application and a database server.

Syntax



Element	Purpose	Restrictions	Syntax
connection	String that specifies a connection to terminate	Connection name that the CONNECT statement assigned	Quoted String, p. 4-243
connection_var	Host variable that holds the name of a connection	Must be a fixed-length character data type	Language specific

Usage

DISCONNECT terminates a connection to a database server. If a database is open, it closes before the connection drops. Even if you made a connection to a specific database only, the connection to the database server is terminated by DISCONNECT. If DISCONNECT does not terminate the current connection, the connection context of the current environment is not changed.

DISCONNECT is not valid as statement text in a PREPARE statement.

E/C

If you disconnect with *connection* or *connection_var*, DISCONNECT generates an error if the specified connection is not a current or dormant connection. ♦

DEFAULT Option

DISCONNECT DEFAULT disconnects the default connection.

The *default connection* is one of the following connections:

- A connection established by the CONNECT TO DEFAULT statement
- An implicit default connection established by the DATABASE or CREATE DATABASE statement

You can use DISCONNECT to disconnect the default connection. If the DATABASE statement does not specify a database server, as in the following example, the default connection is made to the default database server:

```
EXEC SQL database 'stores_demo';
. . .
EXEC SQL disconnect default;
```

If the DATABASE statement specifies a database server, as the following example shows, the default connection is made to that database server:

```
EXEC SQL database 'stores_demo@mydbsrvr';
. . .
EXEC SQL disconnect default;
```

In either case, the DEFAULT option of DISCONNECT disconnects this default connection. For more information, see "DEFAULT Option" on page 2-94.

Specifying the CURRENT Keyword

The DISCONNECT CURRENT statement terminates the current connection. For example, the DISCONNECT statement in the following program fragment terminates the current connection to the database server **mydbsrvr**:

```
CONNECT TO 'stores_demo@mydbsrvr'
. . .
DISCONNECT CURRENT
```

When a Transaction is Active

DISCONNECT generates an error during a transaction. The transaction remains active, and the application must explicitly commit it or roll it back. If an application terminates without issuing DISCONNECT (because of a system failure or program error, for example), active transactions are rolled back.

DB-Access

In an ANSI-compliant database, however, if no error is encountered while you exit from DB-Access in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction. ♦

E/C

Disconnecting in a Thread-Safe Environment

If you issue the DISCONNECT statement in a thread-safe ESQL/C application, keep in mind that an active connection can only be disconnected from within the thread in which it is active. Therefore, one thread cannot disconnect the active connection of another thread. The DISCONNECT statement generates an error if such an attempt is made.

Once a connection becomes dormant, however, any other thread can disconnect it unless an ongoing transaction is associated with the dormant connection that was established with the WITH CONCURRENT TRANSACTION clause of CONNECT. If the dormant connection was not established with the WITH CONCURRENT TRANSACTION clause, DISCONNECT generates an error when it tries to disconnect it.

For an explanation of connections in a thread-safe ESQL/C application, see "SET CONNECTION" on page 2-646.

Specifying the ALL Option

Use the keyword ALL to terminate all connections established by the application up to that time. For example, the following DISCONNECT statement disconnects the current connection as well as all dormant connections:

```
DISCONNECT ALL
```

E/C

In ESQL/C, the ALL keyword has the same effect on multithreaded applications that it has on single-threaded applications. Execution of the DISCONNECT ALL statement causes all connections in all threads to be terminated. However, the DISCONNECT ALL statement fails if any of the connections is in use or has an ongoing transaction associated with it. If either of these conditions is true, none of the connections is disconnected. ♦

Related Information

Related statements: CONNECT, DATABASE, and SET CONNECTION

For information on multithreaded applications, see the *IBM Informix ESQL/C Programmer's Manual*.

+

IDS

DROP ACCESS_METHOD

Use the DROP ACCESS_METHOD statement to remove a previously defined access method from the database.

Syntax

```
DROP ──── ACCESS_METHOD ──── access_method ──── RESTRICT ────┤
```

Element	Purpose	Restrictions	Syntax
access_method	Name of access method to drop	Must be registered in the **sysams** system catalog table by a prior CREATE ACCESS_METHOD statement.	Database Object Name, p. 4-46

Usage

The RESTRICT keyword is required. You cannot drop an access method if virtual tables or indexes exist that use the access method. You must be the owner of the access method or have DBA privileges to drop an access method.

If a transaction is in progress, the database server waits to drop the access method until the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

Related Information

Related statements: ALTER ACCESS_METHOD and CREATE ACCESS_METHOD

For a description of the RESTRICT keyword, see “Specifying RESTRICT Mode” on page 2-389. For more information on primary-access methods, see the *IBM Informix Virtual-Table Interface Programmer’s Guide*.

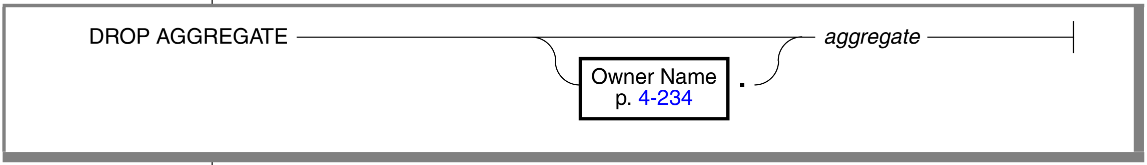
For more information on secondary-access methods, see the *IBM Informix Virtual-Index Interface Programmer’s Guide*. For a discussion of privileges, see the GRANT statement or the *IBM Informix Database Design and Implementation Guide*.

+
IDS

DROP AGGREGATE

Use the DROP AGGREGATE statement to drop a user-defined aggregate that you created with the CREATE AGGREGATE statement.

Syntax



Element	Purpose	Restrictions	Syntax
aggregate	Name of the user-defined aggregate to be dropped	Must have been previously created with the CREATE AGGREGATE statement.	Identifier, p. 4-189

Usage

Dropping a user-defined aggregate does not drop the support functions that you defined for the aggregate in the CREATE AGGREGATE statement. The database server does not track dependency of SQL statements on user-defined aggregates that you use in the statements. For example, you can drop a user-defined aggregate that is used in an SPL routine. In the following example, the user drops the aggregate named **my_avg**:

```
DROP AGGREGATE my_avg
```

Related Information

Related statements: CREATE AGGREGATE

For information about how to invoke a user-defined aggregate, see the discussion of user-defined aggregates in the Expression segment. For a description of the **sysaggregates** system catalog table that holds information about user-defined aggregates, see the *IBM Informix Guide to SQL: Reference*. For a discussion of user-defined aggregates, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

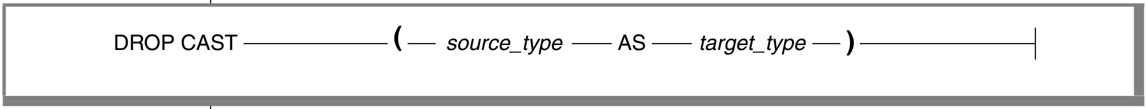
+

IDS

DROP CAST

Use the DROP CAST statement to remove an existing cast from the database.

Syntax



Element	Purpose	Restrictions	Syntax
source_type	Data type on which the cast operates	Must exist	Identifier, p. 4-189
target_type	Data type that results when the cast is invoked	Must exist	Identifier, p. 4-189

Usage

You must be owner of the cast or have the DBA privilege to use DROP CAST. Dropping a cast removes its definition from the **syscasts** catalog table, so the cast cannot be invoked explicitly or implicitly. Dropping a cast has no effect on the user-defined function associated with the cast. Use the DROP FUNCTION statement to remove the user-defined function from the database.

Warning: *Do not drop the built-in casts, which user* ***informix*** *owns. The database server uses built-in casts for automatic conversions between built-in data types.*

A cast defined on a given data type can also be used on any DISTINCT types created from that source type. If you drop the cast, you can no longer invoke it for the DISTINCT types, but dropping a cast that is defined for a DISTINCT type has no effect on casts for its source type. When you create a DISTINCT type, the database server automatically defines an explicit cast from the DISTINCT type to its source type and another explicit cast from the source type to the DISTINCT type. When you drop the DISTINCT type, the database server automatically drops these two casts.

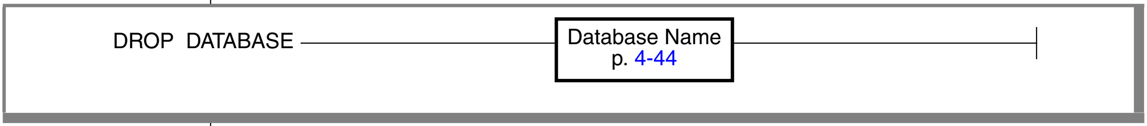
Related Information

Related statements: CREATE CAST and DROP FUNCTION. For more information about data types, refer to the *IBM Informix Guide to SQL: Reference*.

DROP DATABASE

Use the DROP DATABASE statement to delete an entire database, including all system catalog tables, indexes, and data.

Syntax



Usage

This statement is an extension to ANSI-standard syntax. The ANSI/ISO standard for SQL does not provide syntax for the destruction of a database.

You must have the DBA privilege or be user **informix** to run the DROP DATABASE statement successfully. Otherwise, the database server issues an error message and does not drop the database.

You cannot drop the current database or a database that is being used by another user. All the current users of the database must first execute the CLOSE DATABASE statement before DROP DATABASE can be successful.

The DROP DATABASE statement cannot appear in a multistatement PREPARE statement.

During a DROP DATABASE operation, the database server acquires a lock on each table in the database and holds the locks until the entire operation is complete. Configure your database server with enough locks to accommodate this fact.

For example, if the database to be dropped has 2500 tables, but fewer than 2500 locks were configured for your database server, the DROP DATABASE statement will fail. For more information on how to configure the number of locks available to the database server, see the discussion of the LOCKS configuration parameter in your *Administrator's Reference*.

The following statement drops the **stores_demo** database:

```
DROP DATABASE stores_demo
```

DB

In DB-Access, use this statement with caution. DB-Access does not prompt you to verify that you want to delete the entire database. ♦

E/C

You can use a simple database name in a program or host variable, or you can use the full database server and database name. For more information, see "Database Name" on page 4-44. ♦

Related Information

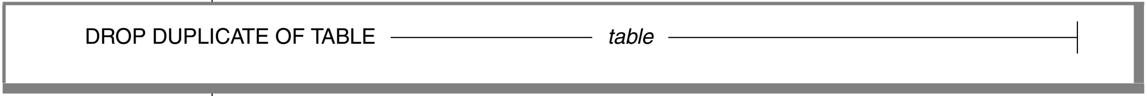
Related statements: CLOSE DATABASE, CREATE DATABASE, and CONNECT

+
XPS

DROP DUPLICATE

Use the DROP DUPLICATE statement to remove from the database all duplicate copies of a specified existing table that the CREATE DUPLICATE statement created in a specified dbslice or in specified dbspaces across coservers. The original table is not affected by DROP DUPLICATE.

Syntax



```
DROP DUPLICATE OF TABLE ---------- table ----------|
```

Element	Description	Restrictions	Syntax
table	Name of the table for which you want to remove all duplicates	Must exist and must be a duplicated table.	Database Object Name

Usage

To drop all duplicate copies of a duplicated table and leave only the original table, enter the DROP DUPLICATE statement. Because duplicate tables are read-only, to update a duplicated table, you must first drop all duplicate copies.

Attached indexes on the copies of the duplicate table are also dropped when DROP DUPLICATE is successfully executed.

Related Statement

CREATE DUPLICATE

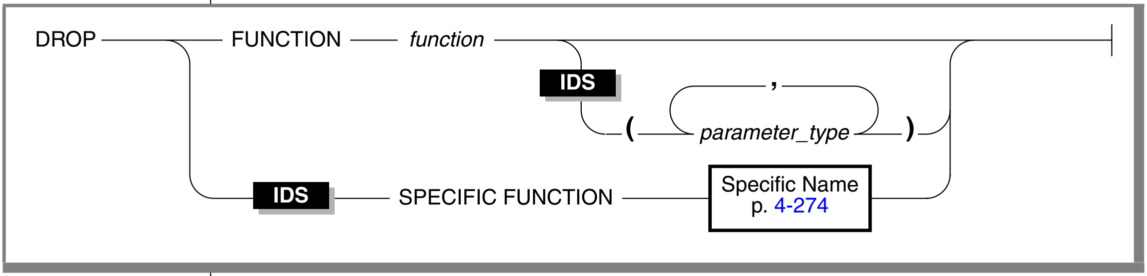
+

IDS

DROP FUNCTION

Use the DROP FUNCTION statement to remove a user-defined function from the database.

Syntax



Element	Purpose	Restrictions	Syntax
function	Name of the user-defined function to drop	Must exist (that is, be registered) in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type*.	Database Object Name, p. 4-46
parameter_type	Data type of the parameter	The data type (or list of data types) must be the same data types (and specified in the same order) as those specified in the CREATE FUNCTION statement when the function was created.	Identifier, p. 4-189

Usage

Dropping a user-defined function removes the text and executable versions of the function from the database.

If you do not know if a UDR is a user-defined function or a user-defined procedure, you can drop the UDR by using the DROP ROUTINE statement.

To use the DROP FUNCTION statement, you must be the owner of the user-defined function or have the DBA privilege.

If the *function* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If the database server cannot resolve an ambiguous function name whose signature differs from that of another function only in an unnamed ROW type parameter, an error is returned. (This error cannot be anticipated by the database server at the time when the ambiguous *function* is defined.)

Examples

If you use parameter data types to identify a user-defined function, they follow the function name, as in the following example:

```
DROP FUNCTION compare(int, int)
```

If you use the specific name for the user-defined function, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC FUNCTION compare_point
```

SPL

Dropping SPL Functions

The Informix implementation of the SQL language does not support ALTER PROCEDURE, ALTER ROUTINE, or ALTER FUNCTION statements. To change the text of an SPL function, you must drop it and then re-create it. Make sure to keep a copy of the SPL function text somewhere outside the database, in case you need to re-create a function after it is dropped.

You cannot drop an SPL function from within the same SPL function.

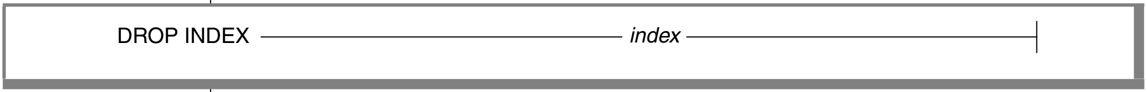
Related Information

Related statements: ALTER FUNCTION, CREATE FUNCTION, CREATE FUNCTION FROM, DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, and EXECUTE FUNCTION

DROP INDEX

Use the DROP INDEX statement to remove an index.

Syntax



Element	Purpose	Restrictions	Syntax
index	Name of the index to be dropped	Must exist	Database Object Name, p. 4-46

Usage

You must be the owner of the index or have the DBA privilege to use the DROP INDEX statement. The following example drops the index **o_num_ix** that **joed** owns. The **stores_demo** database must be the current database.

```
DROP INDEX stores_demo:joed.o_num_ix
```

You cannot use DROP INDEX to drop a unique constraint; you must use ALTER TABLE to drop indexes that implement constraints that CREATE TABLE (or ALTER TABLE) created. The index is not actually dropped if it is shared by constraints. Instead, it is renamed in the **sysindexes** system catalog table with the following format:

```
[space]<tabid>_<constraint_id>
```

Here *tabid* and *constraint_id* are from the **systables** and **sysconstraints** system catalog tables, respectively. The **sysconstraints.idxname** column is then updated to reflect this change. Thus, the updated name might be something like: `“ 121_13”` (where quotes have been used to show the blank space). If this index is a unique index with only referential constraints sharing it, the index is downgraded to a duplicate index after it is renamed.

Related Information

Related statements: ALTER TABLE, CREATE INDEX, and CREATE TABLE. For the performance characteristics of indexes, see your *Performance Guide*.

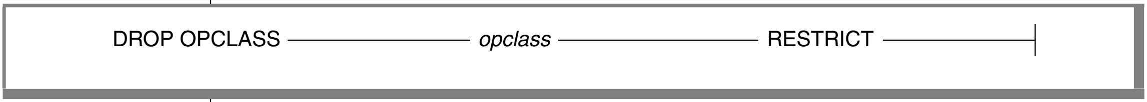
+

IDS

DROP OPCLASS

Use the DROP OPCLASS statement to remove an existing operator class from the database.

Syntax



Element	Purpose	Restrictions	Syntax
opclass	Name of operator class to be dropped	Must have been created by a previous CREATE OPCLASS statement.	Identifier, p. 4-189

Usage

You must be the owner of the operator class or have the DBA privilege to use the DROP OPCLASS statement.

The RESTRICT keyword causes DROP OPCLASS to fail if the database contains indexes defined on the operator class you plan to drop. Therefore, before you drop the operator class, you must use DROP INDEX to drop dependent indexes.

The following DROP OPCLASS statement drops an operator class called **abs_btree_ops**:

```
DROP OPCLASS abs_btree_ops RESTRICT
```

Related Information

Related statement: CREATE OPCLASS

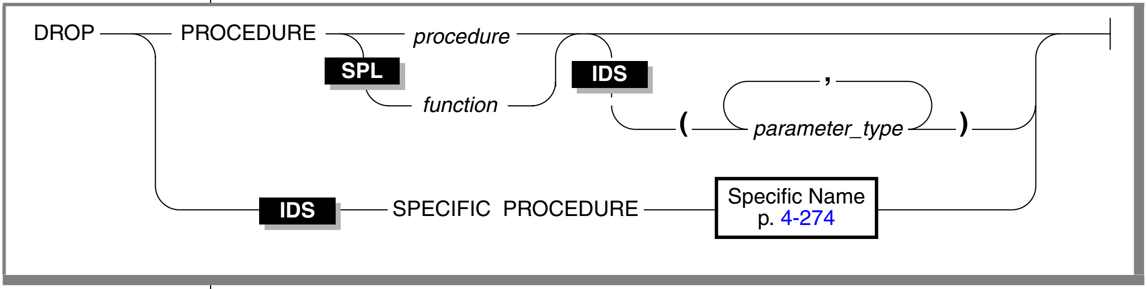
For information on how to create or extend an operator class, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

+

DROP PROCEDURE

Use the DROP PROCEDURE statement to remove a user-defined procedure from the database.

Syntax



Element	Purpose	Restrictions	Syntax
function	Name of SPL function to drop	Must exist (that is, be registered) in the database.	Database Object Name, p.4-46
parameter_type	The data type of the parameter	The data type (or list of data types) must be the same types (and specified in the same order) as those specified in the CREATE PROCEDURE statement when the procedure was created.	Identifier, p. 4-189
procedure	Name of user-defined procedure to drop	Must exist (that is, be registered) in the database.	Database Object Name, p.4-46

Usage

Dropping a user-defined procedure removes the text and executable versions of the procedure.

To use the DROP PROCEDURE statement, you must be the owner of the procedure or have the DBA privilege.

XPS

In Extended Parallel Server, use the DROP PROCEDURE statement to drop any SPL routine. Extended Parallel Server does not support the DROP FUNCTION statement. ♦

IDS

In Dynamic Server, for backward compatibility, you can use the DROP PROCEDURE statement to drop an SPL function that was created with the CREATE PROCEDURE statement.

If the *function* or *procedure* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If the database server cannot resolve an ambiguous UDR name whose signature differs from that of another UDR only in an unnamed ROW type parameter, an error is returned. (This error cannot be anticipated by the database server at the time when the ambiguous *function* or *procedure* is defined.)

If you do not know whether a UDR is a user-defined procedure or a user-defined function, you can use the DROP ROUTINE statement. For more information, see "DROP ROUTINE" on page 2-382. ♦

SPL

The Informix implementation of the SQL language does not support ALTER PROCEDURE, ALTER ROUTINE, or ALTER FUNCTION statements. To change the text of an SPL procedure, you must drop it and then re-create it. Make sure to keep a copy of the SPL procedure text somewhere outside the database, in case you need to re-create the procedure after it is dropped.

You cannot drop an SPL procedure within the same SPL procedure. ♦

IDS

For backward compatibility, you can use this statement to drop an SPL function that was created with the CREATE PROCEDURE statement. ♦

Examples

If you use parameter data types to identify a user-defined procedure, they follow the procedure name, as in the following example:

```
DROP PROCEDURE compare(int, int)
```

If you use the specific name for the user-defined procedure, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC PROCEDURE compare_point
```

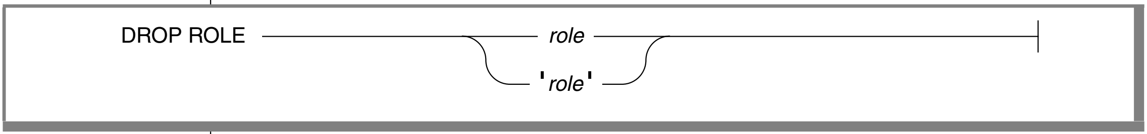
Related Information

Related statements: CREATE PROCEDURE, CREATE PROCEDURE FROM, DROP FUNCTION, DROP ROUTINE, and EXECUTE PROCEDURE

DROP ROLE

Use the DROP ROLE statement to remove a previously created role.

Syntax



Element	Purpose	Restrictions	Syntax
role	Name of the role to drop	Must have been created with the CREATE ROLE statement. When a role name is enclosed in quotation marks, it is case sensitive.	Identifier, p. 4-189

Usage

Either the DBA or a user to whom the role was granted with the WITH GRANT OPTION can issue the DROP ROLE statement.

After a role is dropped, the privileges associated with that role, such as table-level privileges or routine-level privileges, are dropped, and a user cannot grant or enable a role. If a user is using the privileges of a role when the role is dropped, the user automatically loses those privileges.

The following statement drops the role **engineer**:

```
DROP ROLE engineer
```

Related Information

Related statements: CREATE ROLE, GRANT, REVOKE, and SET ROLE

For a discussion on how to use roles, see the *IBM Informix Guide to SQL: Tutorial*.

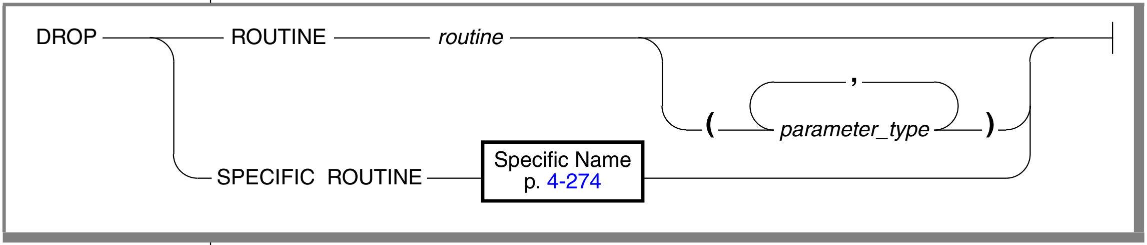
+

IDS

DROP ROUTINE

Use the DROP ROUTINE statement to remove a user-defined routine (UDR) from the database.

Syntax



Element	Purpose	Restrictions	Syntax
parameter_type	Data type of the parameter	The data type (or list of data types) must be the same type (and specified in the same order) as in the UDR definition.	Identifier, p. 4-189
routine	Name of the UDR to drop	The UDR must exist (that is, be registered) in the database.	Database Object Name, p. 4-46

Usage

Dropping a UDR removes the text and executable versions of the UDR from the database. If you do not know whether a UDR is a user-defined function or a user-defined procedure, this statement instructs the database server to drop the specified user-defined function or user-defined procedure.

To use the DROP ROUTINE statement, you must be the owner of the UDR or have the DBA privilege.

Restrictions

When you use this statement, the type of UDR cannot be ambiguous. The UDR that you specify must refer to either a user-defined function or a user-defined procedure.

If either of the following conditions exist, the database server returns an error:

- The name (and parameters) that you specify apply to both a user-defined procedure and a user-defined function
- The *specific* name that you specify applies to both a user-defined procedure and a user-defined function

If the *routine* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If the database server cannot resolve an ambiguous routine name whose signature differs from that of another routine only in an unnamed ROW type parameter, an error is returned. (This error cannot be anticipated by the database server at the time when the ambiguous *routine* is defined.)

Examples

If you use parameter data types to identify a UDR, they follow the UDR name, as in the following example:

```
DROP ROUTINE compare(int, int)
```

If you use the specific name for the UDR, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC ROUTINE compare_point
```

SPL

Dropping SPL Routines

Because you cannot change the text of an SPL routine, you must drop it and then re-create it. Make sure that you have a copy of the SPL function text somewhere outside the database, in case you want to re-create it after it is dropped.

You cannot drop an SPL routine from within the same SPL routine.

Related Information

Related statements: CREATE FUNCTION, CREATE PROCEDURE, DROP FUNCTION, DROP PROCEDURE, EXECUTE FUNCTION, and EXECUTE PROCEDURE

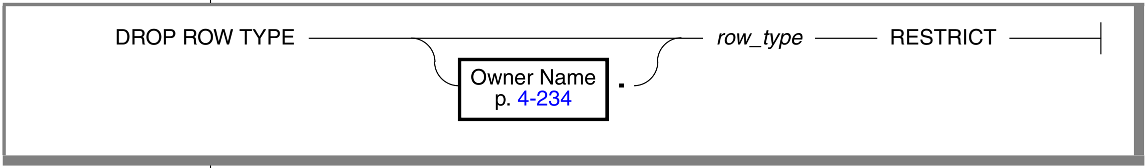
+

IDS

DROP ROW TYPE

Use the DROP ROW TYPE statement to remove an existing named-ROW type from the database.

Syntax



Element	Purpose	Restrictions	Syntax
row_type	Name of an existing named-ROW type to be dropped	Must exist. See also the Usage section that follows.	Identifier, p. 4-189

Usage

You must be the owner of the named-ROW type or have the DBA privilege to use the DROP ROW TYPE statement.

You cannot drop a named-ROW type if its name is in use. You cannot drop a named-ROW type when any of the following conditions are true:

- Any existing tables or columns are using the named-ROW type.
- The named-ROW type is a supertype in an inheritance hierarchy.
- A view is defined on a column of the named ROW type.

To drop a named-ROW type column from a table, use ALTER TABLE.

The DROP ROW TYPE statement cannot drop unnamed-ROW types.

The RESTRICT Keyword

The RESTRICT keyword is required with the DROP ROW TYPE statement. RESTRICT causes DROP ROW TYPE to fail if dependencies on that named-ROW type exist.

The DROP ROW TYPE statement fails and returns an error message if any of the following conditions is true:

- The named-ROW type is used for an existing table or column

  Check the **systables** and **syscolumns** system catalog tables to find out whether any tables or types use the named-ROW type.
- The named-ROW type is the supertype in an inheritance hierarchy

  Look in the **sysinherits** system catalog table to see which named-ROW types have child types.

The following statement drops the ROW type named **employee_t**:

```
DROP ROW TYPE employee_t RESTRICT
```

Related Information

Related statement: CREATE ROW TYPE

For a description of the system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

For a discussion of named-ROW data types, see the *IBM Informix Guide to SQL: Tutorial*.

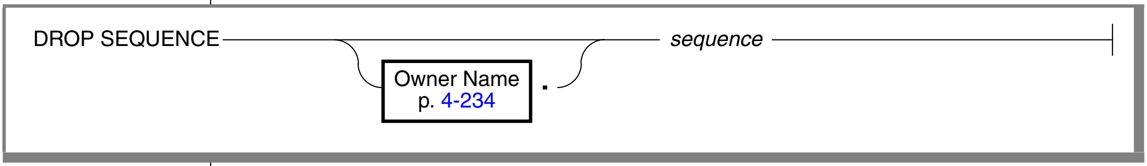
+

IDS

DROP SEQUENCE

Use the DROP SEQUENCE statement to remove a sequence from the database.

Syntax



Element	Purpose	Restrictions	Syntax
sequence	Name of a sequence	Must exist in the current database	Identifier, p. 4-189

Usage

To drop a sequence, you must be the owner of the sequence or have the DBA privilege on the database.

You cannot use a synonym to specify the identifier of the *sequence* in the DROP SEQUENCE statement.

If you drop a sequence, any synonyms for the name of the sequence are also dropped automatically by the database server.

ANSI

In an ANSI-compliant database, you must qualify the name the sequence with the name of its owner (*owner.sequence*) if you are not the owner. ♦

Related Information

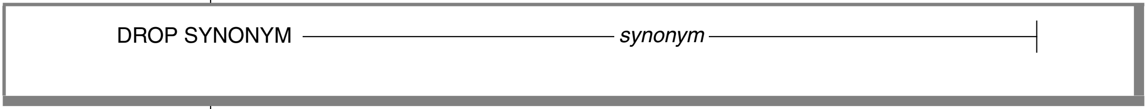
Related statements: ALTER SEQUENCE, CREATE SEQUENCE, RENAME SEQUENCE, CREATE SYNONYM, DROP SYNONYM, GRANT, REVOKE, INSERT, UPDATE, and SELECT

For information about generating values from a sequence, see “NEXTVAL and CURRVAL Operators” on page 4-102.

DROP SYNONYM

Use the DROP SYNONYM statement to remove an existing synonym.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Name of a synonym to drop	The synonym and the table or view to which the synonym points must exist.	Database Object Name, p. 4-46

Usage

You must be the owner of the synonym or have the DBA privilege to use the DROP SYNONYM statement.

The following statement drops the synonym **nj_cust**, which **cathyg** owns:

```
DROP SYNONYM cathyg.nj_cust
```

If a table, view, or sequence is dropped, any synonyms that are in the same database and that refer to that table, view, or sequence are also dropped.

If a synonym refers to an external table or view that is dropped, the synonym remains in place until you explicitly drop it using DROP SYNONYM. You can create another table, view, or synonym in place of the dropped table or view and give the new database object the name of the dropped table or view. The old synonym then refers to the new database object. For a complete discussion of synonym chaining, see the CREATE SYNONYM statement.

Related Information

Related statement: CREATE SYNONYM

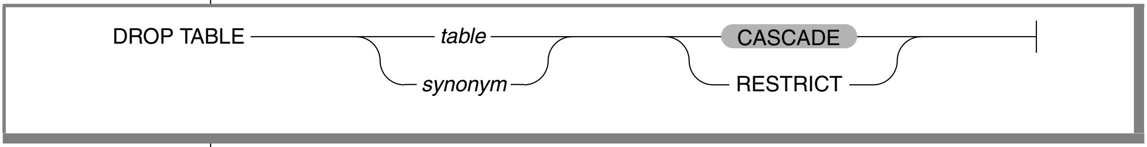
For a discussion of synonyms, see the *IBM Informix Guide to SQL: Tutorial*.

+

DROP TABLE

Use the DROP TABLE statement to remove a table with its associated indexes and data.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Local synonym for a table that is to be dropped	The synonym and its table must exist, and **USETABLENAME** must not be set to `1`	Database Object Name, p. 4-46
table	Name of a table to drop	The table must exist	Database Object Name, p. 4-46

Usage

You must be the owner of the table or have the DBA privilege to use the DROP TABLE statement.

XPS

You cannot drop an Extended Parallel Server table that includes a dependent GK index unless that index is entirely dependent on the affected table. ♦

DB

If you issue a DROP TABLE statement, DB-Access does not prompt you to verify that you want to delete an entire table. ♦

Effects of the DROP TABLE Statement

Use the DROP TABLE statement with caution. When you remove a table, you also delete the data stored in it, the indexes or constraints on the columns (including all the referential constraints placed on its columns), any local synonyms assigned to it, any triggers created on it, and any authorizations granted on the table. You also drop all views based on the table and any violations and diagnostics tables associated with the table.

DROP TABLE does not remove any synonyms for the table that were created in an external database. To remove external synonyms for the dropped table, you must do so explicitly with the DROP SYNONYM statement.

You can prevent users from specifying a synonym in the DROP TABLE statement by setting the **USETABLENAME** environment variable. When **USETABLENAME** is set, the database server issues an error if any user attempts to specify DROP TABLE *synonym*.

Specifying CASCADE Mode

The CASCADE keyword in DROP TABLE removes related database objects, including referential constraints built on the table, views defined on the table, and any violations and diagnostics tables associated with the table.

IDS

If the table is the supertable in an inheritance hierarchy, CASCADE drops all of the subtables as well as the supertable. ♦

The CASCADE mode is the default mode of the DROP TABLE statement. You can also specify this mode explicitly with the CASCADE keyword.

Specifying RESTRICT Mode

The RESTRICT keyword can control the drop operation for supertables, for tables that have referential constraints and views defined on them, or for tables that have violations and diagnostics tables associated them. Using the RESTRICT option causes the drop operation to fail and an error message to be returned if any of the following conditions are true:

- Existing referential constraints reference *table.*
- Existing views are defined on *table.*
- Any violations tables or diagnostics tables are associated with *table*.

IDS

- The *table* is the supertable in an inheritance hierarchy. ♦

IDS

Dropping a Table with Rows That Contain Opaque Data Types

Some opaque data types require special processing when they are deleted. For example, if an opaque type contains spatial or multi-representational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

The database server removes opaque types by calling a user-defined support function called **destroy( )**. When you execute the DROP TABLE statement on a table whose rows contain an opaque type, the database server automatically invokes the **destroy( )** function for the type. The **destroy( )** function can perform certain operations on columns of the opaque data type before the table is dropped. For more information about the **destroy( )** support function, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Tables That Cannot Be Dropped

Observe the following restrictions on the types of tables that you can drop:

- You cannot drop any system catalog tables.
- You cannot drop a table that is not in the current database.
- You cannot drop a violations table or diagnostics table.

  Before you can drop such a table, you must first issue a STOP VIOLATIONS TABLE statement on the base table with which the violations and diagnostics tables are associated.
- XPS If you are using Extended Parallel Server, you cannot drop a table that appears in the FROM clause of a GK index. ♦

Examples of Dropping a Table

The following example deletes two tables. Both tables are within the current database and are owned by the current user. Neither table has a violations or diagnostics table associated with it. Neither table has a referential constraint or view defined on it.

```
DROP TABLE customer;
DROP TABLE stores_demo@accntg:joed.state;
```

Related Information

Related statements: CREATE TABLE and DROP DATABASE

For a discussion on the data integrity of tables, see the *IBM Informix Guide to SQL: Tutorial*.

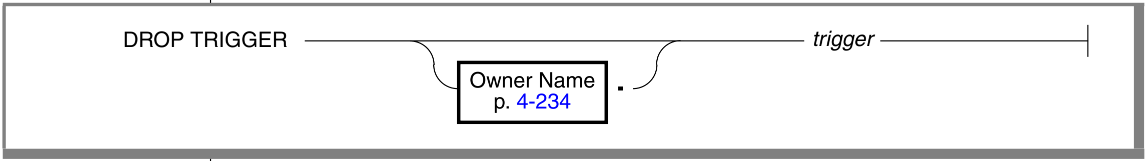
For a discussion on how to create a table, see the *IBM Informix Database Design and Implementation Guide*.

+

DROP TRIGGER

Use the DROP TRIGGER statement to remove a trigger definition from a database.

Syntax



Element	Purpose	Restrictions	Syntax
trigger	Name of the trigger to drop	The trigger must exist.	Identifier, p. 4-189

Usage

You must be the owner of the trigger or have the DBA privilege to drop the trigger. Dropping a trigger removes the text of the trigger definition and the executable trigger from the database. The row describing the specified trigger is deleted from the **systriggers** system catalog table.

IDS

Dropping an INSTEAD OF trigger on a complex view (a view with columns from more than one table) revokes any privileges on the view that the owner of the trigger received automatically when creating the trigger, and also revokes any privileges that the owner of the trigger granted to other users. (Dropping a trigger on a simple view does not revoke any privileges.) ♦

The following statement drops the **items_pct** trigger:

```
DROP TRIGGER items_pct
```

If a DROP TRIGGER statement appears inside an SPL routine that is called by a data manipulation (DML) statement, the database server returns an error.

Related Information

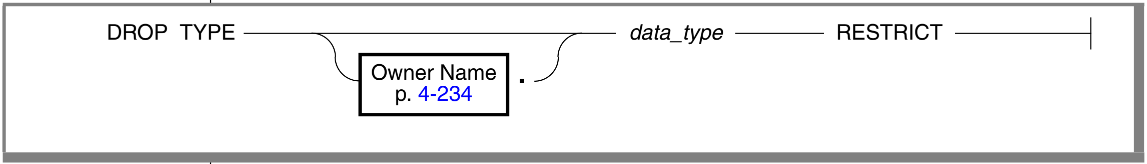
Related statements: CREATE TRIGGER

+
IDS

DROP TYPE

Use the DROP TYPE statement to remove an existing distinct or opaque data type from the database. (You cannot use this to drop a built-in data type.)

Syntax



Element	Purpose	Restrictions	Syntax
data_type	Distinct or opaque data type to be removed	Must be an existing distinct or opaque type; must not be a built-in data type	Identifier, p. 4-189

Usage

To drop a distinct or opaque data type with the DROP TYPE statement, you must be the owner of the data type or have the DBA privilege. When you use this statement, you remove the type definition from the database (in the **sysxtdtypes** system catalog table). In general, this statement does not remove any definitions for casts or support functions associated with that data type.

Important: *When you drop a distinct type, the database server automatically drops the two explicit casts between the distinct type and the type on which it is based.*

You cannot drop a distinct or opaque type if the database contains any casts, columns, or user-defined functions whose definitions reference the type.

The following statement drops the **new_type** data type:

```
DROP TYPE new_type RESTRICT
```

Related Information

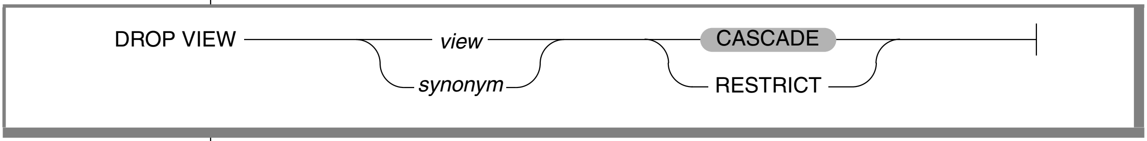
Related statements: CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, CREATE ROW TYPE, DROP ROW TYPE, and CREATE TABLE

+

DROP VIEW

Use the DROP VIEW statement to remove a view from the database.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Name of a synonym to drop	The synonym and the view to which it points must exist.	Database Object Name, p. 4-46
view	Name of a view to drop	The view must exist.	Database Object Name, p. 4-46

Usage

To drop a view, you must be the owner or have the DBA privilege.

When you drop a view or its synonym, you also drop any other views and INSTEAD OF triggers whose definitions depend on that view. (You can also specify this default behavior explicitly with the CASCADE keyword.)

When you use the RESTRICT keyword in the DROP VIEW statement, the drop operation fails if any other existing views are defined on *view*; otherwise, these would be abandoned in the drop operation.

You can query the **sysdepend** system catalog table to determine which views, if any, depend on another view.

The following statement drops the view that is named **cust1**:

```
DROP VIEW cust1
```

Related Information

Related statements: CREATE VIEW and DROP TABLE

For a discussion of views, see the *IBM Informix Guide to SQL: Tutorial*.

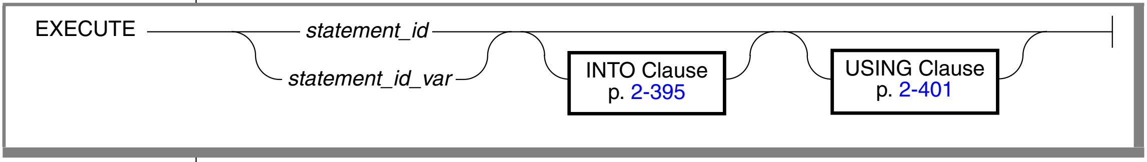
E/C

IDS

EXECUTE

Use the EXECUTE statement to run a previously prepared statement or set of statements. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
statement_id	Identifier of a prepared SQL statement	Must have been defined in a previous PREPARE statement.	PREPARE, p. 2-527
statement_id_var	Host variable that contains an SQL statement	Must have been defined in a previous PREPARE statement and must be a character data type.	PREPARE, p. 2-527

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. The following example shows an EXECUTE statement within an ESQL/C program:

```
EXEC SQL PREPARE del_1 FROM
   'DELETE FROM customer
      WHERE customer_num = 119';
EXEC SQL EXECUTE del_1;
```

Once prepared, an SQL statement can be executed as often as needed.

After you release the database server resources (using a FREE statement), you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.

If the statement contained question mark (?) placeholders, use the USING clause to provide specific values for them before execution. For more information, see the "USING Clause" on page 2-401.

You can execute any prepared statement except those in the following list:

- A prepared SELECT statement that returns more than one row

  When you use a prepared SELECT statement to return multiple rows of data, you must use a cursor to retrieve the data rows. As an alternative, you can EXECUTE a prepared SELECT INTO TEMP statement to achieve the same result.

  For more information on cursors, see "DECLARE" on page 2-323.
- A prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns more than one row

  When you prepare an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns multiple rows, you must use a cursor to retrieve the data rows.

  For more information on how to execute a SELECT or an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, see "PREPARE" on page 2-527.

If you create or drop a trigger after you prepare a triggering INSERT, DELETE, or UPDATE statement, the prepared statement returns an error when you execute it.

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of reference of a statement identifier is global to the program. A statement identifier created in one file can be referenced from another file.

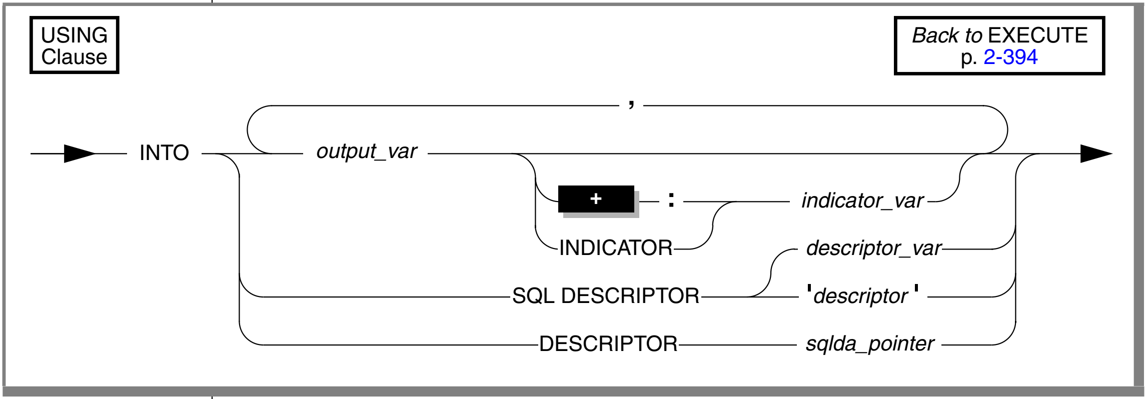
In a multiple-file program, if you want to limit the scope of reference of a statement identifier to the file in which it is executed, you can preprocess all the files with the **-local** command-line option.

INTO Clause

Use the INTO clause to save the returned values of these SQL statements:

- A prepared singleton SELECT statement that returns only one row of column values for the columns in the select list
- A prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns only one set of values

The INTO clause of the EXECUTE statement has the following syntax:



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	Must already be allocated. Use single (') quotation marks.	Quoted String, p. 4-243
descriptor_var	Host variable that identifies a system-descriptor area	System-descriptor area must already be allocated.	Language specific
indicator_var	Host variable that receives a return code if corresponding *parameter_var* is NULL value, or if truncation occurs	Cannot be DATETIME or INTERVAL data type.	Language specific
output_var	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Must be a character data type.	Language specific
sqlda_pointer	Pointer to an **sqlda** structure that defines data type and memory location of values to replace a question-mark (?) placeholder in a prepared object	Cannot begin with a dollar sign ($) or a colon (**:**) symbol. An **sqlda** structure is required with dynamic SQL.	DESCRIBE, p. 2-351

This closely resembles the syntax of the "USING Clause" on page 2-401.

The INTO clause provides a concise and efficient alternative to more complicated and lengthy syntax. In addition, by placing values into variables that can be displayed, the INTO clause simplifies and enhances your ability to retrieve and display data values. For example, if you use the INTO clause, you do not have to use a cursor to retrieve values from a table.

You can store the returned values in output variables, in output SQL descriptors, or in output **sqlda** pointers.

Restrictions with the INTO Clause

If you execute a prepared SELECT statement that returns more than one row or a prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns more than one group of return values, you receive an error message. In addition, if you prepare and declare a statement and then attempt to execute that statement, you receive an error message.

You cannot select a null value from a table column and place that value into an output variable. If you know in advance that a table column contains a null value, after you select the data, check the indicator variable that is associated with the column to determine if the value is null.

To use the INTO clause with the EXECUTE statement

1. Declare the output variables that the EXECUTE statement uses.
2. Use PREPARE to prepare your SELECT statement or to prepare your EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
3. Use the EXECUTE statement, with the INTO clause, to execute your SELECT statement or to execute your EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.

Storage Location for Returned Values

You can specify any of the following items to replace the question-mark placeholders in a statement before you execute it:

- A host variable name (if the number and data type of the question marks are known at compile time)
- A system descriptor that identifies a system
- A descriptor that is a pointer to an **sqlda** structure

Saving Values In Host or Program Variables

If you know the number of return values to be supplied at runtime and their data types, you can define the values that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns as host variables in your program. Use these host variables with the INTO keyword, followed by the names of the variables. These variables are matched with the return values in a one-to-one correspondence, from left to right.

You must supply one variable name for each value that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) returns. The data type of each variable must be compatible with the corresponding returned value from the prepared statement.

Saving Values in a System-Descriptor Area

If you do not know the number of return values to be supplied at runtime or their data types, you can associate output values with a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values.

X/O

A system-descriptor area conforms to the X/Open standards. ♦

To specify a system-descriptor area as the location of output values, use the INTO SQL DESCRIPTOR clause of the EXECUTE statement. Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are stored in the system-descriptor area.

The following example shows how to use the system-descriptor area to execute prepared statements in IBM Informix ESQL/C:

```
EXEC SQL allocate descriptor 'desc1';
...
sprintf(sel_stmt, "%s %s %s",
   "select fname, lname from customer",
   "where customer_num =",
   cust_num);
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL execute sel1 into sql descriptor 'desc1';
```

The COUNT field corresponds to the number of values that the prepared statement returns. The value of COUNT must be less than or equal to the value of the occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

For more information, refer to the discussion of the system-descriptor area in the *IBM Informix ESQL/C Programmer's Manual*.

E/C

Saving Values in an sqlda Structure

If you do not know the number of output values to be returned at runtime or their data types, you can associate output values from an **sqlda** structure. An **sqlda** structure lists the data type and memory location of one or more return values. To specify an **sqlda** structure as the location of return values, use the INTO DESCRIPTOR clause of the EXECUTE statement. Each time the EXECUTE statement is run, the database server places the returns values that the **sqlda** structure describes into the **sqlda** structure.

The following example shows how to use an **sqlda** structure to execute a prepared statement in IBM Informix ESQL/C:

```
struct sqlda *pointer2;
...
sprintf(sel_stmt, "%s %s %s",
   "select fname, lname from customer",
   "where customer_num =",
   cust_num);
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL describe sel1 into pointer2;
EXEC SQL execute sel1 into descriptor pointer2;
```

The **sqld** value specifies the number of output values that are described in occurrences of **sqlvar**. This number must correspond to the number of values that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns.

For more information, refer to the **sqlda** discussion in the *IBM Informix ESQL/C Programmer's Manual*.

This example uses the INTO clause with an EXECUTE statement in ESQL/C:

```
EXEC SQL prepare sel1 from 'select fname, lname from customer
    where customer_num =123';
EXEC SQL execute sel1 into :fname, :lname using :cust_num;
```

The next example uses the INTO clause to return multiple rows of data:

```
EXEC SQL BEGIN DECLARE SECTION;
int customer_num =100;
char fname[25];
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare sel1 from 'select fname from customer
    where customer_num=?';
for ( ;customer_num < 200; customer_num++)
    {
    EXEC SQL execute sel1 into :fname using customer_num;
    printf("Customer number is %d\n", customer_num);
    printf("Customer first name is %s\n\n", fname);
    }
```

The sqlca Record and EXECUTE

Following an EXECUTE statement, the sqlca can reflect two results:

- The **sqlca** can reflect an error within the EXECUTE statement.

  For example, when an UPDATE …WHERE statement in a prepared statement processes zero rows, the database server sets **sqlca** to `100`.
- The **sqlca** can reflect the success or failure of the executed statement.

Error Conditions with EXECUTE

If a prepared statement fails to access any rows, the database server returns zero `(0)`. In a multistatement prepare, if any statement in the following list fails to access rows, the database server returns `SQLNOTFOUND (100)`:

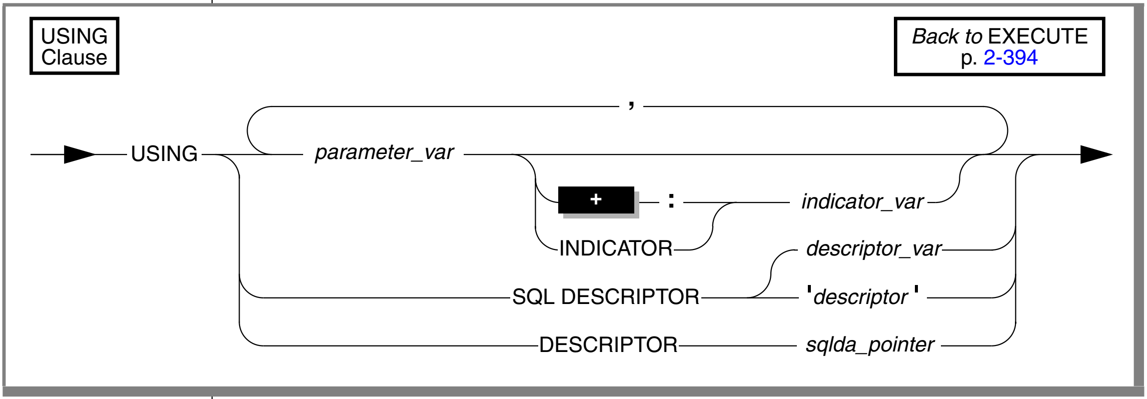
- INSERT INTO *table* SELECT … WHERE
- SELECT INTO TEMP…WHERE
- DELETE … WHERE
- UPDATE … WHERE

ANSI

In an ANSI-compliant database, if you prepare and execute any of the statements in the preceding list, and no rows are returned, the database server returns **SQLNOTFOUND** `( = 100).` ♦

USING Clause

Use the USING clause to specify the values that are to replace question-mark (?) placeholders in the prepared statement. Providing values in the EXECUTE statement that replace the question-mark placeholders in the prepared statement is sometimes called *parameterizing* the prepared statement.



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. Use single (**'**) quotation marks.	Quoted String, p. 4-243
descriptor_var	Host variable that identifies a system-descriptor area	System-descriptor area must already be allocated.	Language specific
indicator_var	Host variable that receives a return code if corresponding *parameter_var* is NULL value, or if truncation occurs	Cannot be DATETIME or INTERVAL data type.	Language specific
parameter_var	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Must be a character data type.	Language specific
sqlda_pointer	Pointer to an sqlda structure that defines data type and memory location of values tot replace question-mark (?) placeholder in a prepared object	Cannot begin with a dollar sign ($) or a colon (**:**). An **sqlda** structure is required with dynamic SQL.	DESCRIBE, p. 2-351

This closely resembles the syntax of the "INTO Clause" on page 2-395.

If you know the number of parameters to be supplied at runtime and their data types, you can define the parameters that are needed by the EXECUTE statement as host variables in your program.

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate input values from a system-descriptor area or an **sqlda** structure. Both of these descriptor structures describe the data type and memory location of one or more values to replace question-mark (?) placeholders.

Supplying Parameters Through Host or Program Variables

You pass parameters to the database server by opening the cursor with the USING keyword, followed by the names of the variables. These variables are matched with prepared statement question-mark (?) placeholders in a one-to-one correspondence, from left to right. You must supply one storage-parameter variable for each placeholder. The data type of each variable must be compatible with the corresponding value that the prepared statement requires.

The following example executes the prepared UPDATE statement in ESQL/C:

```
stcopy ("update orders set order_date = ?
   where po_num = ?", stm1);
EXEC SQL prepare statement_1 from :stm1;
EXEC SQL execute statement_1 using :order_date, :po_num;
```

Supplying Parameters Through a System Descriptor

You can create a system-descriptor area that describes the data type and memory location of one or more values and then specify the descriptor in the USING SQL DESCRIPTOR clause of the EXECUTE statement.

Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The COUNT field corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

The following example shows how to use system descriptors to execute a prepared statement in ESQL/C:

```
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

E/C

Supplying Parameters Through an sqlda Structure

You can specify the **sqlda** pointer in the USING DESCRIPTOR clause of the EXECUTE statement.

Each time the EXECUTE statement is run, the values that the descriptor structure describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The **sqld** value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement.

The following example shows how to use an **sqlda** structure to execute a prepared statement in ESQL/C:

```
EXEC SQL execute prep_stmt using descriptor pointer2
```

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE IMMEDIATE, FETCH, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR

For a task-oriented discussion of the EXECUTE statement, see the *IBM Informix Guide to SQL: Tutorial*.

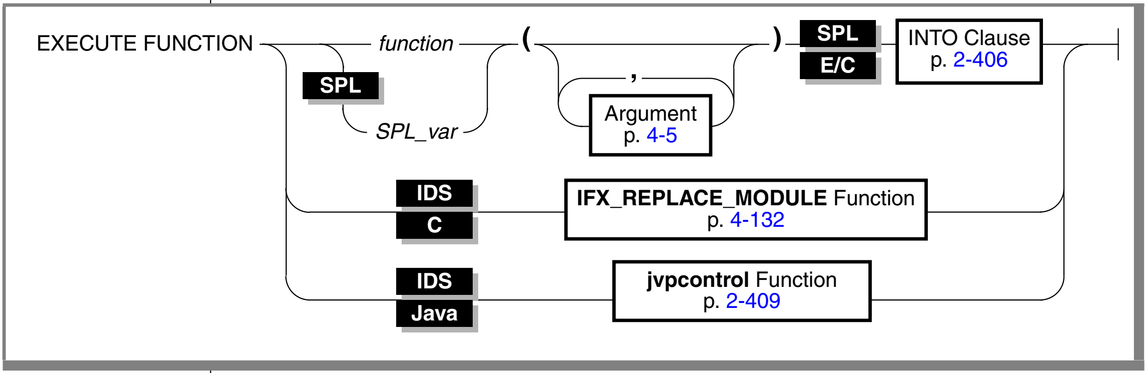
For more information about concepts that relate to the EXECUTE statement, see the *IBM Informix ESQL/C Programmer's Manual*.

+

EXECUTE FUNCTION

Use the EXECUTE FUNCTION statement to execute a user-defined function.

Syntax



Element	Purpose	Restrictions	Syntax
function	Name of a user-defined function to execute	Must exist.	Database Object Name, p. 4-46
SPL_var	Variable that contains the name of an SPL routine to be executed	Must be a CHAR, VARCHAR, NCHAR, or NVARCHAR data type that contains the non-NULL name of an existing SPL function.	Identifier, p. 4-189

Usage

The EXECUTE FUNCTION statement invokes a user-defined function, with arguments, and specifies where the results are to be returned.

An external function returns exactly one value.

An SPL function can return one or more values.

You cannot use the EXECUTE FUNCTION statement to execute any type of user-defined procedure that returns no value. Instead, use the EXECUTE PROCEDURE or EXECUTE ROUTINE statement to execute procedures.

You must have the Execute privilege on the user-defined function.

If a user-defined function has a companion function, any user who executes the function must have the Execute privilege on both the function and its companion. For example, if a function has a negator function, any user who executes the function must have the Execute privilege on both the function and its negator.

For more information, see "GRANT" on page 2-459.

How the EXECUTE FUNCTION Statement Works

For a user-defined function to be executed with the EXECUTE FUNCTION statement, the following conditions must exist:

- The qualified function name or the function signature (the function name with its parameter list) must be unique within the name space or database.
- The function must exist.
- The function must not have any OUT parameters.

If EXECUTE FUNCTION specifies fewer arguments than the user-defined function expects, the unspecified arguments are said to be *missing*. Missing arguments are initialized to their corresponding parameter default values, if these were defined. The syntax of specifying default values for parameters is described in "Routine Parameter List" on page 4-266.

EXECUTE FUNCTION returns an error under the following conditions:

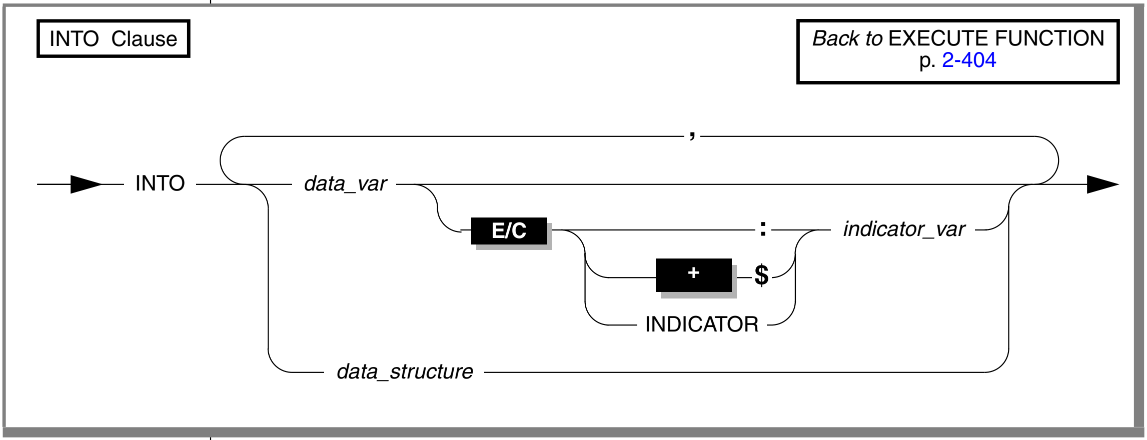
- It specifies more arguments than the user-defined function expects.
- One or more arguments are missing and do not have default values.

  In this case, the arguments are initialized to the value of UNDEFINED.
- The fully qualified function name or the signature is not unique.
- No function with the specified name or signature that you specify is found.
- You use it to try to execute a user-defined procedure.

If the *function* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name.

If the database server cannot resolve an ambiguous function name whose signature differs from that of another routine only in an unnamed-ROW type parameter, an error is returned. (This error cannot be anticipated by the database server at the time when the ambiguous function is defined.)

INTO Clause



Element	Purpose	Restrictions	Syntax
data_structure	Structure that was declared as a host variable	Individual elements of structure must be compatible with the data types of the returned values.	Language specific
data_var	Variable to receive the value that a user-defined function returns	See "Data Variables" on page 2-407.	Language specific
indicator_var	Program variable to store a return code if the corresponding *data_var* receives a NULL value	Use an indicator variable if the value of the corresponding *data_var* might be NULL.	Language specific

You must include an INTO clause with EXECUTE FUNCTION to specify the variables that receive the values that a user-defined function returns. If the function returns more than one value, the values are returned into the list of variables in the order in which you specify them.

If the EXECUTE FUNCTION statement stands alone (that is, it is not part of a DECLARE statement and does not use the INTO clause), it must execute a noncursor function. A noncursor function returns only one row of values. The following example shows a SELECT statement in IBM Informix ESQL/C:

```
EXEC SQL execute function cust_num(fname, lname, company_name)
   into :c_num;
```

Data Variables

If you issue the EXECUTE FUNCTION statement within an ESQL/C program, *data_var* must be a host variable. Within an SPL routine, *data_var* must be an SPL variable.

If you issue the EXECUTE FUNCTION statement within a CREATE TRIGGER statement, *data_var* must be column names within the triggering table or another table.

E/C

INTO Clause with Indicator Variables

You should use an indicator variable if the possibility exists that data returned from the user-defined function statement is null. For more information about indicator variables, see the *IBM Informix ESQL/C Programmer's Manual*.

INTO Clause with Cursors

If the EXECUTE FUNCTION statement executes a user-defined function that returns more than one row of values, it must execute a cursor function. A cursor function can return one or more rows of values and must be associated with a function cursor to execute.

If the SPL function returns more than one row or a collection data type, you must access the rows or collection elements with a cursor.

Ext

To return more than one row of values, an external function must be defined as an iterator function. For more information on how to write iterator functions, see the *IBM Informix DataBlade API Programmer's Guide*. ♦

SPL

To return more than one row of values, an SPL function must include the WITH RESUME keywords in its RETURN statement. For more information on how to write SPL functions, see the *IBM Informix Guide to SQL: Tutorial*. ♦

E/C

In an IBM Informix ESQL/C program, the DECLARE statement can declare a function cursor and the FETCH statement can return rows individually from the cursor. You can put the INTO clause in the FETCH or in the EXECUTE FUNCTION statement, but you cannot put it in both. The following IBM Informix ESQL/C code examples show different ways you can use the INTO clause:

- Using the INTO clause in the EXECUTE FUNCTION statement:

```
EXEC SQL declare f_curs cursor for
   execute function get_orders(customer_num)
   into :ord_num, :ord_date;
EXEC SQL open f_curs;
while (SQLCODE == 0)
   EXEC SQL fetch f_curs;
EXEC SQL close f_curs;
```

- Using the INTO clause in the FETCH statement:

```
EXEC SQL declare f_curs cursor for
   execute function get_orders(customer_num);
EXEC SQL open f_curs;
while (SQLCODE == 0)
   EXEC SQL fetch f_curs into :ord_num, :ord_date;
EXEC SQL close f_curs;
```

♦

SPL

In an SPL routine, if a SELECT returns more than one row, you must use the FOREACH statement to access the rows individually. The INTO clause of the SELECT statement can store the fetched values. For more information, see "FOREACH" on page 3-27. ♦

Alternatives to PREPARE ... EXECUTE FUNCTION ... INTO

E/C

You cannot prepare an EXECUTE FUNCTION statement that includes the INTO clause. For similar functionality, however, you can follow these steps:

1. Prepare the EXECUTE FUNCTION statement with no INTO clause.
2. Declare a function cursor for the prepared statement.
3. Open the cursor.
4. Execute the FETCH statement with an INTO clause to fetch the returned values into program variables.

Alternatively, you can do the following:

1. Declare a cursor for the EXECUTE FUNCTION statement without first preparing the statement, and include the INTO clause in the EXECUTE FUNCTION when you declare the cursor.
2. Open the cursor.
3. Fetch the returned values from the cursor without using the INTO clause of the FETCH statement. ♦

SPL

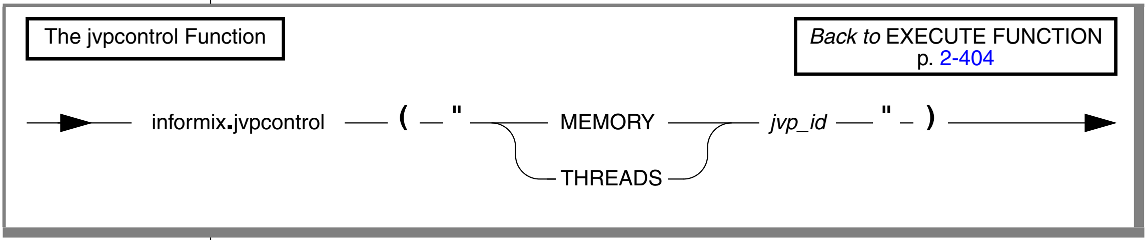
Dynamic Routine-Name Specification of SPL Functions

Dynamic routine-name specification simplifies the writing of an SPL function that calls another SPL routine whose name is not known until runtime. To specify the name of an SPL routine in the EXECUTE FUNCTION statement, instead of listing the explicit name of an SPL routine, you can use an SPL variable to hold the routine name. For more information about how to execute SPL functions dynamically, see the *IBM Informix Guide to SQL: Tutorial*.

Java

The jvpcontrol Function

The **jvpcontrol( )** function is a built-in iterative function that you can use to obtain information about a Java Virtual Processor (JVP) class.



Element	Purpose	Restrictions	Syntax
jvp_id	Name of the Java Virtual Processor (JVP) class for which you want information	The specified Java Virtual Processor class must exist.	Identifier, p. 4-189

You must associate this function with the equivalent of a cursor in the Java language.

Using the MEMORY Keyword

When you specify the MEMORY keyword, the **jvpcontrol** function returns the memory usage on the JVP class that you specify. The following example requests information about the memory usage of the JVP class named **4**:

```
EXECUTE FUNCTION INFORMIX.JVPCONTROL ("MEMORY 4");
```

Using the THREADS Keyword

When you specify the THREADS keyword, the **jvpcontrol** function returns a list of the threads running on the JVP class that you specify. The following example requests information about the threads running on the JVP class named `4`:

```
EXECUTE FUNCTION INFORMIX.JVPCONTROL ("THREADS 4");
```

Related Information

Related statements: CALL, CREATE FUNCTION, CREATE FUNCTION FROM, DROP FUNCTION, DROP ROUTINE, EXECUTE PROCEDURE, and FOREACH

+

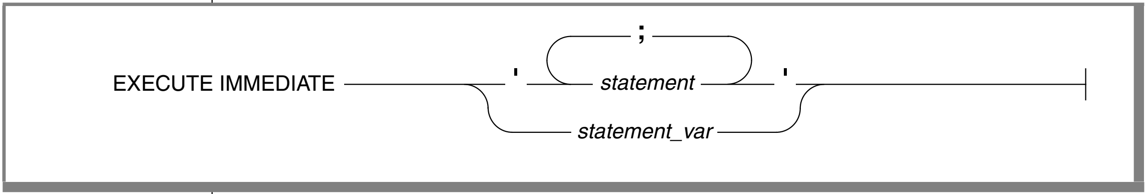
E/C

EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to perform the functions of the PREPARE, EXECUTE, and FREE statements.

Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
statement	A valid SQL statement	See the same sections of Usage that are listed below for *statement_var.*	See this chapter.
statement_var	Host variable that contains a character string of one or more SQL statements	Must be a previously declared character-type variable. See "EXECUTE IMMEDIATE and Restricted Statements" on page 2-412 and "Restrictions on Allowed Statements" on page 2-412.	Language specific

Usage

The EXECUTE IMMEDIATE statement makes it easy to execute dynamically a single simple SQL statement that is constructed during program execution. For example, you can obtain the name of a database from program input, construct the DATABASE statement as a program variable, and then use EXECUTE IMMEDIATE to execute the statement, which opens the database.

The quoted string that includes one or more SQL statements, or the contents of *statement_var*, is parsed and executed if correct; then all data structures and memory resources are released immediately. In the usual method of dynamic execution, these operations require separate PREPARE, EXECUTE, and FREE statements.

The maximum length of an EXECUTE IMMEDIATE statement is 64 kilobytes.

EXECUTE IMMEDIATE and Restricted Statements

You cannot use the EXECUTE IMMEDIATE statement to execute the following SQL statements. Although the EXECUTE PROCEDURE statement appears on this list, the restriction applies only to EXECUTE PROCEDURE statements that return values.

CLOSE
CONNECT
DECLARE
DISCONNECT
EXECUTE
EXECUTE FUNCTION
EXECUTE PROCEDURE
FETCH
GET DESCRIPTOR
GET DIAGNOSTICS
OPEN
OUTPUT
PREPARE
SELECT
SET AUTOFREE
SET CONNECTION
SET DEFERRED_PREPARE
SET DESCRIPTOR
WHENEVER

In addition, you cannot use the EXECUTE IMMEDIATE statement to execute the following statements in text that contains multiple statements that are separated by semicolons:

CLOSE DATABASE
CREATE DATABASE
DATABASE
DROP DATABASE
SELECT
(except SELECT INTO TEMP)

Use a PREPARE and either a cursor or the EXECUTE statement to execute a dynamically constructed SELECT statement.

Restrictions on Allowed Statements

The following restrictions apply to the statement that is contained in the quoted string or in the statement variable:

- The statement cannot contain a host-language comment.
- Names of host-language variables are not recognized as such in prepared text.

  The only identifiers that you can use are names defined in the database, such as table names and columns.

- The statement cannot reference a host-variable list or a descriptor; it must not contain any question-mark (?) placeholders, which are allowed with a PREPARE statement.
- The text must not include any embedded SQL statement prefix, such as the dollar sign ($) or the keywords EXEC SQL.

  Although it is not required, the SQL statement terminator (;) can be included in the statement text.

IDS

- A SELECT or INSERT statement cannot contain a Collection-Derived Table clause.

  EXECUTE IMMEDIATE cannot process input host variables, which are required for a collection variable. Use the EXECUTE statement or a cursor to process prepared accesses to collection variables. ♦

Examples of the EXECUTE IMMEDIATE Statement

The following examples show EXECUTE IMMEDIATE statements in ESQL/C. Both examples use host variables that contain a CREATE DATABASE statement. The first example uses the SQL statement terminator (;) inside the quoted string.

```
sprintf(cdb_text1, "create database %s;", usr_db_id);
EXEC SQL execute immediate :cdb_text;

sprintf(cdb_text2, "create database %s", usr_db_id);
EXEC SQL execute immediate :cdb_text;
```

Related Information

Related statements: EXECUTE, FREE, and PREPARE

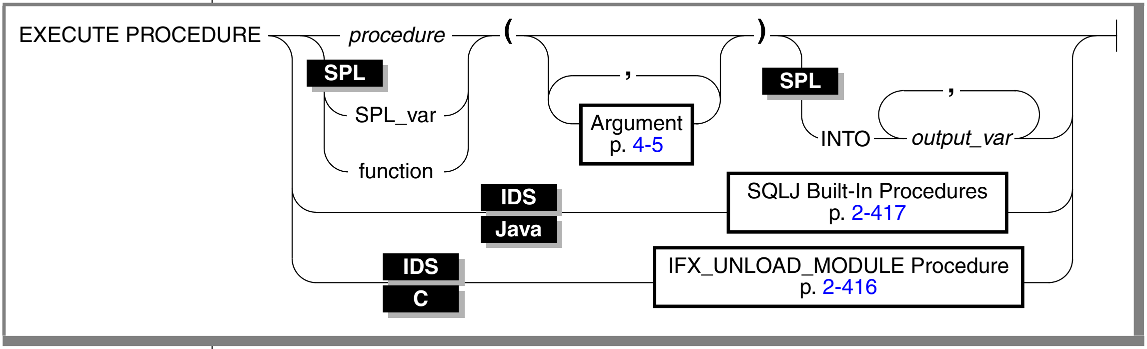
For a discussion of quick execution, see the *IBM Informix Guide to SQL: Tutorial*.

+

EXECUTE PROCEDURE

Use the EXECUTE PROCEDURE statement to invoke a user-defined procedure.

Syntax



Element	Purpose	Restrictions	Syntax
function	SPL function to execute	Must exist.	Database Object Name, p. 4-46
output_var	Host variable or program variable that receives the returned value from UDR	In the context of a CREATE TRIGGER statement, must contain column names in the triggering table or in another table.	Language specific
procedure	User-defined procedure to execute	Must exist.	Database Object Name, p. 4-46
SPL_var	Variable that contains the name of the SPL routine to execute	Must be a character data type that contains the non-NULL name of an SPL routine.	Identifier, p. 4-189

Usage

The EXECUTE PROCEDURE statement invokes the named user-defined procedure and specifies its arguments.

IDS

In Dynamic Server, for backward compatibility, you can use the EXECUTE PROCEDURE statement to execute an SPL function that you created with the CREATE PROCEDURE statement. ♦

XPS

In Extended Parallel Server, use the EXECUTE PROCEDURE statement to execute any SPL routine. Extended Parallel Server does not support the EXECUTE FUNCTION statement. ♦

E/C

In ESQL/C, if the EXECUTE PROCEDURE statement returns more than one row, it must be enclosed within an SPL FOREACH loop or accessed through a cursor. ♦

Causes of Errors

EXECUTE PROCEDURE returns an error under the following conditions:

- It has more arguments than the called procedure expects.
- One or more arguments are missing and do not have default values. In this case the arguments are initialized to the value of UNDEFINED.
- The fully qualified procedure name or the signature is not unique.
- No procedure with the specified name or signature is found.

SPL

Using the INTO Clause

Use the INTO clause to specify where to store the values that the SPL function returns.

If an SPL function returns more than one value, the values are returned into the list of variables in the order in which you specify them. If an SPL function returns more than one row or a collection data type, you must access the rows or collection elements with a cursor.

You cannot prepare an EXECUTE PROCEDURE statement that has an INTO clause. For more information, see "Alternatives to PREPARE ... EXECUTE FUNCTION ... INTO" on page 2-408.

SPL

Dynamic Routine-Name Specification of SPL Procedures

Dynamic routine-name specification simplifies the writing of an SPL routine that calls another SPL routine whose name is not known until runtime. To specify the name of an SPL routine in the EXECUTE PROCEDURE statement, instead of listing the explicit name of an SPL routine, you can use an SPL variable to hold the routine name.

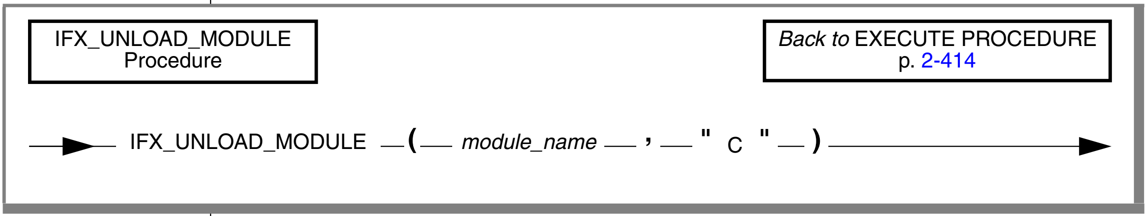
If the SPL variable names an SPL routine that returns a value (an SPL function), include the INTO clause of EXECUTE PROCEDURE to specify a *receiving variable* (or variables) to hold the value (or values) that the SPL function returns. For more information on how to execute SPL procedures dynamically, see the *IBM Informix Guide to SQL: Tutorial*.

IDS

C

IFX_UNLOAD_MODULE Procedure

The **IFX_UNLOAD_MODULE** procedure unloads a shared-object file from memory.


IFX_UNLOAD_MODULE Procedure
Back to EXECUTE PROCEDURE p. 2-414
IFX_UNLOAD_MODULE
(
module_name
,
" C "
)

Element	Purpose	Restrictions	Syntax
module_name	Full pathname of file to unload	Shared-object file must exist and be unused. Pathname can be up to 255 characters long.	Quoted String, p. 4-243

The **IFX_UNLOAD_MODULE** procedure can only unload an unused shared-object file; that is, when no executing SQL statements (in any database) are using any UDRs in the specified shared-object file. If any UDR in the shared-object file is currently in use, then **IFX_UNLOAD_MODULE** raises an error.

UNIX

For example, suppose you want to unload the **circle.so** shared library, which contains C UDRs. If this library resides in the **/usr/apps/opaque_types** directory, you can use the following EXECUTE PROCEDURE statement to execute the **IFX_UNLOAD_MODULE** procedure:

```
EXECUTE PROCEDURE ifx_unload_module(
   “/usr/apps/opaque_types/circle.so”, “C”);
```

♦

Windows

For example, suppose you want to unload the **circle.dll** dynamic link library, which contains C UDRs. If this library is in the **C:\usr\apps\opaque_types** directory, you can use the following EXECUTE PROCEDURE statement to execute the **IFX_UNLOAD_MODULE** procedure:

```
EXECUTE PROCEDURE ifx_unload_module(
   "C:\usr\apps\opaque_types\circle.dll", "C");
```

♦

For more information on how to use **IFX_UNLOAD_MODULE** to unload a shared-object file, see the chapter on how to design a UDR in *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to use the **IFX_REPLACE_MODULE** function, see "IFX_REPLACE_MODULE Function" on page 4-132.

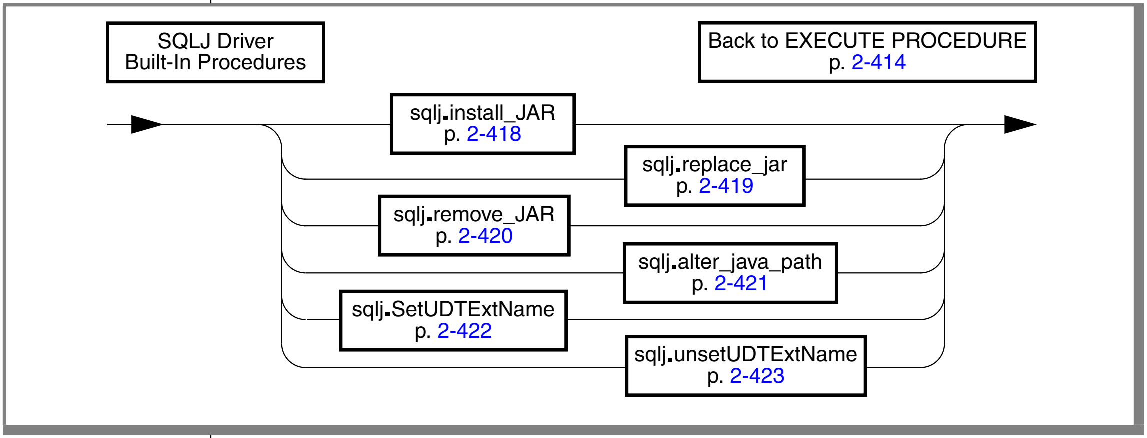
IDS

Java

SQLJ Driver Built-In Procedures

Use the SQLJ Driver built-in procedures for one of the following tasks:

- To install, replace, or remove a set of Java classes
- To specify a path for Java class resolution for Java classes that are included in a JAR file
- To map or remove the mapping between a user-defined type and the Java type to which it corresponds

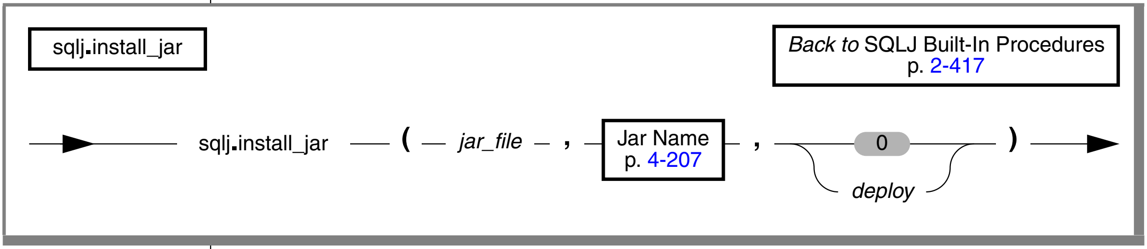


The SQLJ built-in procedures are stored in the **sysprocedures** system catalog table. They are grouped under the **sqlj** schema.

Tip: *For any Java static method, the first built-in procedure that you execute must be the **sqlj.install_jar( )** procedure. You must install the jar file before you can create a UDR or map a user-defined data type to a Java type. Similarly, you cannot use any of the other SQLJ built-in procedures until you have used **sqlj.install_jar( )**.*

sqlj.install_jar

Use the **sqlj.install_jar( )** procedure to install a Java jar file in the current database and assign it a jar identifier.



Element	Purpose	Restrictions	Syntax
deploy	Integer that causes the procedure to search for deployment descriptor files in the jar file	None.	Literal Number, p. 4-216
jar_file	URL of the jar file that contains the UDR written in Java	Maximum length of the URL is 255 bytes.	Quoted String, p. 4-243

For example, consider a Java class **Chemistry** that contains the following static method **explosiveReaction( )**:

```
public static int explosiveReaction(int ingredient);
```

Here the **Chemistry** class resides in this jar file on the server computer:

```
/students/data/Courses.jar
```

You can install all classes in the **Courses.jar** jar file in the current database with the following call to the **sqlj.install_jar( )** procedure:

```
EXECUTE PROCEDURE
   sqlj.install_jar("file://students/data/Courses.jar",
   "course_jar")
```

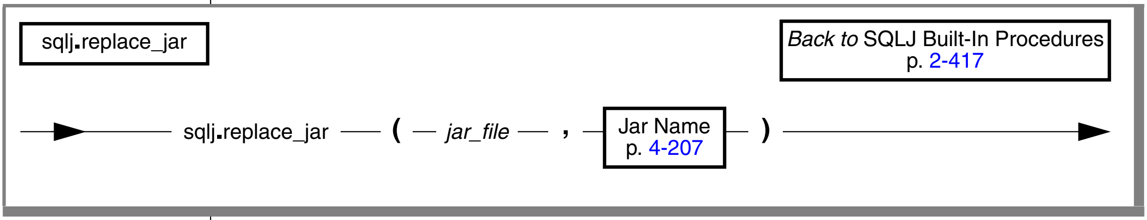
The **sqlj.install_jar( )** procedure assigns the jar ID, **course_jar**, to the **Courses.jar** file that it has installed in the current database.

After you define a jar ID in the database, you can use that jar ID when you create and execute a UDR written in Java.

When you specify a nonzero number for the third argument, the database server searches through any included deployment descriptor files. For example, you might want to include descriptor files that include SQL statements to register and grant privileges on UDRs in the jar file.

sqlj.replace_jar

Use the **sqlj.replace_jar( )** procedure to replace a previously installed jar file with a new version. When you use this syntax, you provide only the new jar file and assign it to the jar ID for which you want to replace the file.



Element	Purpose	Restrictions	Syntax
jar_file	URL of the jar file that contains the UDR written in Java	The maximum length of the URL is 255 bytes.	Quoted String, p. 4-243

If you attempt to replace a jar file that is referenced by one or more UDRs, the database server generates an error. You must drop the referencing UDRs before replacing the jar file.

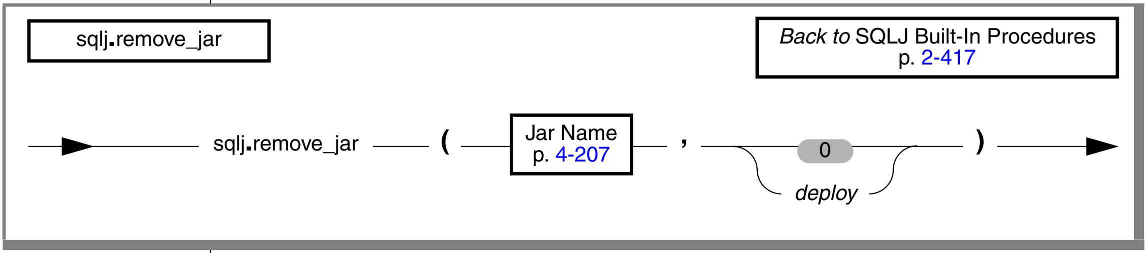
For example, the following call replaces the **Courses.jar** file, which had previously been installed for the **course_jar** identifier, with the **Subjects.jar** file:

```
EXECUTE PROCEDURE
   sqlj.replace_jar("file://students/data/Subjects.jar",
   "course_jar")
```

Before you replace the **Course.jar** file, you must drop the user-defined function **sql_explosive_reaction( )** with the DROP FUNCTION (or DROP ROUTINE) statement.

sqlj.remove_jar

Use the **sqlj.remove_jar( )** procedure to remove a previously installed jar file from the current database.



Element	Purpose	Restrictions	Syntax
deploy	Integer that causes the procedure to search for deployment descriptor files in the jar file	None.	Literal Number, p. 4-216

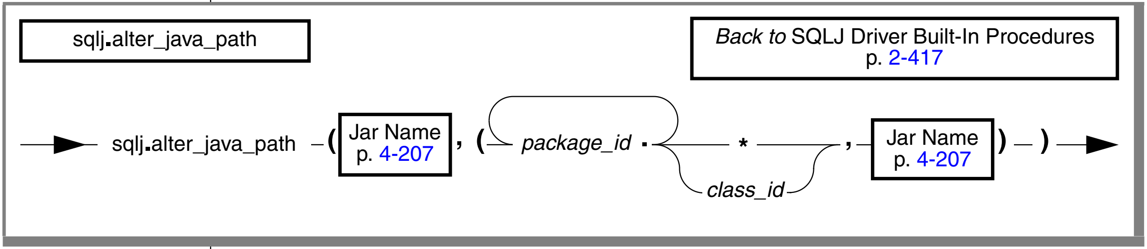
If you attempt to remove a jar file that is referenced by one or more UDRs, the database server generates an error. You must drop the referencing UDRs before you replace the jar file. For example, the following SQL statements remove the jar file associated with the **course_jar** jar id:

```
DROP FUNCTION sql_explosive_reaction;
EXECUTE PROCEDURE sqlj.remove_jar("course_jar")
```

When you specify a nonzero number for the second argument, the database server searches through any included deployment descriptor files. For example, you might want to include descriptor files that include SQL statements that revoke privileges on the UDRs in the associated jar file and to drop them from the database.

sqlj.alter_java_path

Use the **sqlj.alter_java_path( )** procedure to specify the *jar-file path* to use when the routine manager resolves related Java classes for the jar file of a UDR written in Java.



Element	Purpose	Restrictions	Syntax
class_id	Java class that contains method to implement the UDR	The Java class must exist in the jar file that *jar_id* identifies. Identifier must not exceed 255 bytes.	Language specific
package_id	Name of the package that contains the Java class	The fully qualified identifier of *package_id.class_id* must not exceed 255 bytes.	Language specific

The jar IDs that you specify, namely the jar ID for which you are altering the jar-file path and the resolution jar ID, both must have been installed with the **sqlj.install_jar** procedure. When you invoke a UDR written in the Java language, the routine manager attempts to load the Java class in which the UDR resides. At this time, it must resolve the references that this Java class makes to other Java classes.

The three types of such class references are these:

1. References to Java classes that the JVPCLASSPATH configuration parameter specifies (such as Java system classes like **java.util.Vector**)
2. References to classes that are in the same jar file as the UDR
3. References to classes that are outside the jar file that contains the UDR

The routine manager implicitly resolves classes of type 1 and 2 in the preceding list. To resolve type 3 references, it examines all the jar files in the jar-file path that the latest call to **sqlj.alter_java_path( )** specified.

The routine manager throws an exception if it cannot resolve a class reference. The routine manager checks the jar-file path for class references *after* it performs the implicit type 1 and type 2 resolutions.

If you want a Java class to be loaded from the jar file that the jar-file path specifies, make sure the Java class is *not* present in the JVPCLASSPATH configuration parameter. Otherwise, the system loader picks up that Java class first, which might result in a different class being loaded than what you expect.

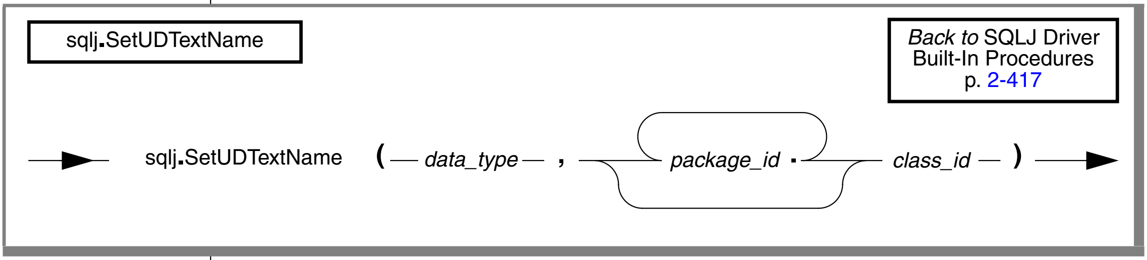
Suppose that the **sqlj.install_jar( )** procedure and CREATE FUNCTION have been executed as the preceding sections describe. The following SQL statement invokes **sql_explosive_reaction( )** function in the **course_jar** jar file:

```
EXECUTE PROCEDURE alter_java_path("course_jar",
   "(professor/*, prof_jar)");
EXECUTE FUNCTION sql_explosive_reaction(10000)
```

The routine manager attempts to load the class **Chemistry**. It uses the path that the call to **sqlj.alter_java_path( )** specifies to resolve any class references. Therefore, it checks the classes that are in the **professor** package of the jar file that **prof_jar** identifies.

sqlj.setUDTExtName

Use the **sqlj.setUDTExtName( )** procedure to define the mapping between a user-defined data type and a Java class.



Element	Purpose	Restrictions	Syntax
class_id	Java class that contains the Java data type	Qualified name *package_id.class_id* must not exceed 255 bytes.	Language-specific rules for Java identifiers
data_type	User-defined type for which to create a mapping	Name must not exceed 255 bytes.	Identifier, p. 4-189
package_id	Name of package that contains the *class_id* Java class	Same length restrictions as *class_id.*	Language-specific rules for Java identifiers

You must have registered the user-defined data type in the CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, or CREATE ROW TYPE statement.

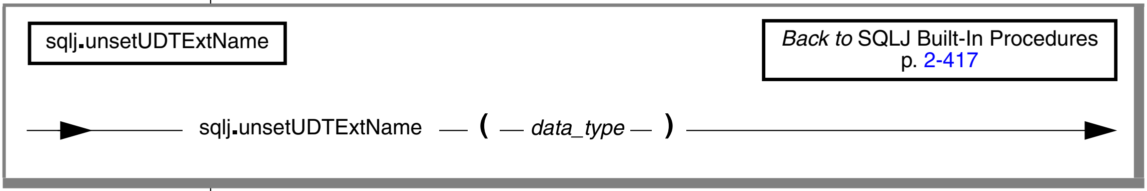
To look up the Java class for a user-defined data type, the database server searches in the jar-file path, which the **sqlj.alter_java_path( )** procedure has specified. For more information on the jar-file path, see "sqlj.alter_java_path" on page 2-421.

On the client side, the driver looks into the **CLASSPATH** path on the client environment before it asks the database server for the name of the Java class.

The **setUDTExtName** procedure is an extension to the *SQLJ:SQL Routines using the Java Programming Language* specification.

sqlj.unsetUDTExtName

Use the **sqlj.unsetUDTExtName( )** procedure to remove the mapping from a user-defined data type to a Java class.



Element	Purpose	Restrictions	Syntax
data_type	User-defined data type for which to remove the mapping	Must exist.	Identifier, p. 4-189

This procedure removes the SQL-to-Java mapping, and consequently removes any cached copy of the Java class from database server shared memory.

The **unsetUDTExtName** procedure is an extension to the *SQLJ:SQL Routines Using the Java Programming Language* specification.

Related Information

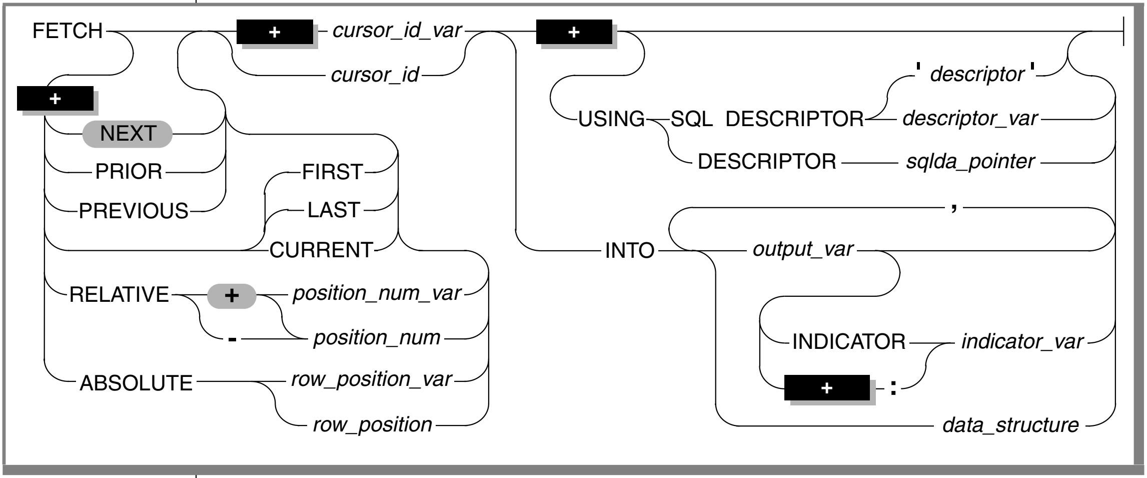
Related statements: CREATE FUNCTION, CREATE PROCEDURE, EXECUTE FUNCTION, GRANT, CALL, FOREACH, and LET

E/C

FETCH

Use the FETCH statement to move a cursor to a new row in the active set and to retrieve the row values from memory. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Cursor to retrieve rows	Must be declared and open	Identifier, p. 4-189
cursor_id_var	Host variable storing *cursor_id*	Must be character data type	Language specific
data_structure	Structure as a host variable	Must store fetched values	Language specific
descriptor	System-descriptor area	Must have been allocated	Quoted String, p. 4-243
descriptor_var	Host variable storing *descriptor*	Must be allocated.	Language specific
indicator_var	Host variable for return code if *output_var* can be NULL value	See "Using Indicator Variables" on page 428.	Language specific
output_var	Host variable for fetched value	Must store value from row	Language specific
position_num	Position relative to current row	Value 0 fetches current row	Literal Number, p. 4-216
position_num_var	Host variable (= *position_num*)	Value 0 fetches current row	Language specific
row_position	Ordinal position in active set	Must be an integer >1	Literal Number, p. 4-216
row_position_var	Host variable (= *row_ position*)	Must be 1 or greater	Language specific
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with **$** nor **:**	See ESQL/C manual.

Usage

How the database server creates, stores, and fetches members of the active set rows depends on whether the cursor is a sequential cursor or a scroll cursor.

X/O

In X/Open mode, if a cursor-direction value (such as NEXT or RELATIVE) is specified, a warning message is issued, indicating that the statement does not conform to X/Open standards. ♦

FETCH with a Sequential Cursor

A sequential cursor can fetch only the next row in sequence from the active set. The sole cursor-position option that is available to a sequential cursor is the default value, NEXT. A sequential cursor can read through a table only once each time it is opened. The following ESQL/C example illustrates the FETCH statement with a sequential cursor:

```
EXEC SQL FETCH seq_curs into :fname, :lname;
EXEC SQL FETCH NEXT seq_curs into ;fname, :lname;
```

When the program opens a sequential cursor, the database server processes the query to the point of locating or constructing the first row of data. The goal of the database server is to tie up as few resources as possible.

Because the sequential cursor can retrieve only the next row, the database server can frequently create the active set one row at a time.

On each FETCH operation, the database server returns the contents of the current row and locates the next row. This one-row-at-a-time strategy is not possible if the database server must create the entire active set to determine which row is the first row (as would be the case if the SELECT statement included an ORDER BY clause).

FETCH with a Scroll Cursor

These ESQL/C examples illustrate the FETCH statement with a scroll cursor:

```
EXEC SQL fetch previous q_curs into :orders;
EXEC SQL fetch last q_curs into :orders;
EXEC SQL fetch relative -10 q_curs into :orders;
printf("Which row? ");
scanf("%d",row_num);
EXEC SQL fetch absolute :row_num q_curs into :orders;
```

A scroll cursor can fetch any row in the active set, either by specifying an absolute row position or a relative offset. Use the following cursor-position options to specify a particular row that you want to retrieve.

Keyword	Purpose
NEXT	Retrieves next row in active set
PREVIOUS	Retrieves previous row in active set
PRIOR	Retrieves previous row in active set (Synonymous with PREVIOUS.)
FIRST	Retrieves the first row in active set
LAST	Retrieves the last row in active set
CURRENT	Retrieves the current row in active set (the same row as returned by the previous FETCH statement from the scroll cursor)
RELATIVE	Retrieves *n*th row, relative to the current cursor position in the active set, where *position_num* (or *position_num_var)* supplies *n.* A negative value indicates the nth row prior to the current cursor position. If *position_num* = `0`, the current row is fetched.
ABSOLUTE	Retrieves *n*th row in active set, where *row_position_var* (or *row_position*) = *n* . Absolute row positions are numbered from `1`.

Tip: *Do not confuse row-position values with* ***rowid*** *values. A* ***rowid*** *value is based on the position of a row in its table and remains valid until the table is rebuilt. A row-position value (a value that the ABSOLUTE keyword introduced) is based on the position of the row in the current active set of the cursor; the next time the cursor is opened, different rows might be selected.*

How the Database Server Stores Rows

The database server must retain all the rows in the active set for a scroll cursor until the cursor closes, because it cannot anticipate which row the program will ask for next. When a scroll cursor opens, the database server implements the active set as a temporary table, although it might not populate this table immediately.

The first time a row is fetched, the database server copies it into the temporary table as well as returning it to the program.

When a row is fetched for the second time, it can be taken from the temporary table. This scheme uses the fewest resources in case the program abandons the query before it fetches all the rows. Rows that are never fetched are usually not created or are saved in a temporary table.

Specifying Where Values Go in Memory

Each value from the select list of the query or the output of the executed user-defined function must be returned into a memory location. You can specify these destinations in one of the following ways:

- Use the INTO clause of a SELECT statement.
- Use the INTO clause of an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
- Use the INTO clause of a FETCH statement.
- Use a system-descriptor area.
- Use an **sqlda** structure.

Using the INTO Clause

If you associate a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a function cursor, the statement can contain an INTO clause to specify variables to receive the returned values. You can use this method only when you write the SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statement as part of the cursor declaration; see "DECLARE" on page 2-323. In this case, the FETCH statement cannot contain an INTO clause.

The following example uses the INTO clause of the SELECT statement to specify program variables in ESQL/C:

```
EXEC SQL declare ord_date cursor for
   select order_num, order_date, po_num
      into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

If you prepare a SELECT statement, the SELECT *cannot* include the INTO clause so you must use the INTO clause of the FETCH statement.

When you create a SELECT statement dynamically, you cannot use an INTO clause because you cannot name host variables in a prepared statement.

If you are certain of the number and data type of values in the select list, you can use an INTO clause in the FETCH statement. If user input generated the query, however, you might not be certain of the number and data type of values that are being selected. In this case, you must use a system descriptor or **sqlda** pointer structure.

Using Indicator Variables

Use an indicator variable if the returned data might be NULL.

The *indicator_var* parameter is optional, but use an indicator variable if the possibility exists that the value of *output_var* is NULL.

If you specify the indicator variable without the INDICATOR keyword, you cannot put a blank space between *output_var* and *indicator_var*.

For information about rules for placing a prefix before the *indicator_var*, see the *IBM Informix ESQL/C Programmer's Manual*.

The host variable cannot be a DATETIME or INTERVAL data type.

Using the INTO Clause of FETCH

When SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) omits the INTO clause, you must specify a data destination when a row is fetched.

For example, to dynamically execute a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) cannot include its INTO clause in the PREPARE statement. Therefore, the FETCH statement must include an INTO clause to retrieve data into a set of variables. This method lets you store different rows in different memory locations.

You can fetch into a program-array element only by using an INTO clause in the FETCH statement. If you use a program array, you must list both the array name and a specific element of the array in *data_structure.* When you are declaring a cursor, do not refer to an array element within the SQL statement.

Tip: *If you are certain of the number and data type of values in the select list of the Projection clause, you can use an INTO clause in the FETCH statement.*

In the following ESQL/C example, a series of complete rows is fetched into a program array. The INTO clause of each FETCH statement specifies an array element as well as the array name.

```
EXEC SQL BEGIN DECLARE SECTION;
   char wanted_state[2];
   short int row_count = 0;
   struct customer_t{
   {
      int    c_no;
      char   fname[15];
      char   lname[15];
   } cust_rec[100];
EXEC SQL END DECLARE SECTION;

main()
{
   EXEC SQL connect to'stores_demo';
   printf("Enter 2-letter state code: ");
   scanf ("%s", wanted_state);
   EXEC SQL declare cust cursor for
      select * from customer where state = :wanted_state;
   EXEC SQL open cust;
   EXEC SQL fetch cust into :cust_rec[row_count];
   while (SQLCODE == 0)
   {
      printf("\n%s %s", cust_rec[row_count].fname,
         cust_rec[row_count].lname);
      row_count++;
      EXEC SQL fetch cust into :cust_rec[row_count];
   }
   printf ("\n");
   EXEC SQL close cust;
   EXEC SQL free cust;
}
```

X/O

Using a System-Descriptor Area

You can use a system-descriptor area to store output values when you do not know the number of return values or their data types that a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns at runtime. A system-descriptor area describes the data type and memory location of one or more return values.

The keywords USING SQL DESCRIPTOR introduce the name of the system-descriptor area into which you fetch the contents of a row or the return values of a user-defined function. You can then use the GET DESCRIPTOR statement to transfer the values that the FETCH statement returns from the system-descriptor area into host variables.

This example shows a valid FETCH...USING SQL DESCRIPTOR statement:

```
EXEC SQL allocate descriptor 'desc';
   ...
EXEC SQL declare selcurs cursor for
   select * from customer where state = 'CA';
EXEC SQL describe selcurs using sql descriptor 'desc';
EXEC SQL open selcurs;
while (1)
   {
   EXEC SQL fetch selcurs using sql descriptor 'desc';
```

You can also use an **sqlda** structure to dynamically supply parameters. A system-descriptor area conforms to the X/Open standards.

Using sqlda Structures

You can use a pointer to an **sqlda** structure to stores output values when you do not know the number of return values or their data types that a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns.

This structure contains data descriptors that specify the data type and memory location for one selected value. The keywords USING DESCRIPTOR introduce the name of the **sqlda** pointer structure.

Tip: *If you are certain of the number and data type of values in the select list, you can use an INTO clause in the FETCH statement. For more information, see "Using the INTO Clause of FETCH" on page 2-428.*

To specify an sqlda structure as the location of parameters

1. Declare an **sqlda** pointer variable.
2. Use the DESCRIBE statement to fill in the **sqlda** structure.
3. Allocate memory to hold the data values.
4. Use the USING DESCRIPTOR clause of FETCH to specify the **sqlda** structure as the location into which you fetch the returned values.

The following example shows a FETCH USING DESCRIPTOR statement:

```
struct sqlda *sqlda_ptr;
...
EXEC SQL declare selcurs2 cursor for
   select * from customer where state = 'CA';
EXEC SQL describe selcurs2 into sqlda_ptr;
...
EXEC SQL open selcurs2;
while (1)
   {
   EXEC SQL fetch selcurs2 using descriptor sqlda_ptr;
   ...
```

The **sqld** value specifies the number of output values that are described in occurrences of the **sqlvar** structures of the **sqlda** structure. This number must correspond to the number of values returned from the prepared statement.

Fetching a Row for Update

The FETCH statement does not ordinarily lock a row that is fetched. Thus, another process can modify (update or delete) the fetched row immediately after your program receives it. A fetched row is locked in the following cases:

- When you set the isolation level to Repeatable Read, each row you fetch is locked with a read lock until the cursor closes or the current transaction ends. Other programs can also read the locked rows.
- When you set the isolation level to Cursor Stability, the current row is locked.
- ANSI In an ANSI-compliant database, an isolation level of Repeatable Read is the default; you can set it to something else. ♦
- When you are fetching through an update cursor (one that is declared FOR UPDATE), each row you fetch is locked with a promotable lock. Other programs can read the locked row, but no other program can place a promotable or write lock; therefore, the row is unchanged if another user tries to modify it using the WHERE CURRENT OF clause of an UPDATE or DELETE statement.

When you modify a row, the lock is upgraded to a write lock and remains until the cursor is closed or the transaction ends. If you do not modify the row, the behavior of the database server depends on the isolation level you have set. The database server releases the lock on an unchanged row as soon as another row is fetched, unless you are using Repeatable Read isolation (see "SET ISOLATION" on page 2-691).

Important: *You can hold locks on additional rows even when Repeatable Read isolation is not in use or is unavailable. Update the row with unchanged data to hold it locked while your program is reading other rows. You must evaluate the effect of this technique on performance in the context of your application, and you must be aware of the increased potential for deadlock.*

When you use explicit transactions, be sure that a row is both fetched and modified within a single transaction; that is, both the FETCH statement and the subsequent UPDATE or DELETE statement must fall between a BEGIN WORK statement and the next COMMIT WORK statement.

IDS

Fetching from a Collection Cursor

A collection cursor allows you to access the individual elements of an ESQL/C collection variable. To declare a collection cursor, use the DECLARE statement and include the Collection-Derived-Table segment in the SELECT statement that you associate with the cursor. After you open the collection cursor with the OPEN statement, the cursor allows you to access the elements of the collection variable.

To fetch elements, one at a time, from a collection cursor, use the FETCH statement and the INTO clause. The FETCH statement identifies the collection cursor that is associated with the collection variable. The INTO clause identifies the host variable that holds the element value that is fetched from the collection cursor. The data type of the host variable in the INTO clause must match the element type of the collection.

Suppose you have a table called **children** with the following structure:

```
CREATE TABLE children
(
   age      SMALLINT,
   name     VARCHAR(30),
   fav_colorsSET(VARCHAR(20) NOT NULL),
)
```

The following ESQL/C code fragment shows how to fetch elements from the **child_colors** collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection child_colors;
   varchar one_favorite[21];
   char child_name[31] = "marybeth";
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate collection :child_colors;
/* Get structure of fav_colors column for untyped
 * child_colors collection variable */
EXEC SQL select fav_colors into :child_colors
   from children
   where name = :child_name;
/* Declare select cursor for child_colors collection
 * variable */
EXEC SQL declare colors_curs cursor for
   select * from table(:child_colors);
EXEC SQL open colors_curs;
do
{
   EXEC SQL fetch colors_curs into :one_favorite;
   ...
} while (SQLCODE == 0)
EXEC SQL close colors_curs;
EXEC SQL free colors_curs;
EXEC SQL deallocate collection :child_colors;
```

After you fetch a collection element, you can modify the element with the UPDATE or DELETE statements. For more information, see the UPDATE and DELETE statements in this manual. You can also insert new elements into the collection variable with an INSERT statement. For more information, see the INSERT statement.

Checking the Result of FETCH

You can use the **SQLSTATE** variable to check the result of each FETCH statement. The database server sets the **SQLSTATE** variable after each SQL statement. If a row is returned successfully, the **SQLSTATE** variable contains the value `00000`. If no row is found, the database server sets the **SQLSTATE** code to `02000`, which indicates `no data found`, and the current row is unchanged. The following conditions set the **SQLSTATE** code to `02000`, indicating `no data found`:

- The active set contains no rows.
- You issue a FETCH NEXT statement when the cursor points to the last row in the active set or points past it.

- You issue a FETCH PRIOR or FETCH PREVIOUS statement when the cursor points to the first row in the active set.
- You issue a FETCH RELATIVE *n* statement when no *n*th row exists in the active set.
- You issue a FETCH ABSOLUTE *n* statement when no *n*th row exists in the active set.

The database server copies the **SQLSTATE** code from the **RETURNED_SQLSTATE** field of the system-diagnostics area. You can use the GET DIAGNOSTICS statement to examine the **RETURNED_SQLSTATE** field directly. The system-diagnostics area can also contain additional error information.

You can also use SQLCODE of **sqlca** to determine the same results.

Related Information

Related statements: ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, SET DEFERRED_PREPARE, and SET DESCRIPTOR

For a task-oriented discussion of the FETCH statement, see the *IBM Informix Guide to SQL: Tutorial*.

For more information about concepts that relate to the FETCH statement, see the *IBM Informix ESQL/C Programmer's Manual*.

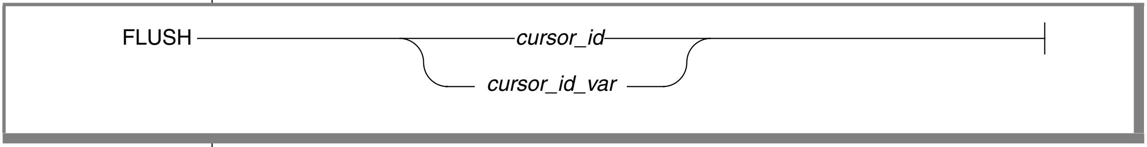
+

E/C

FLUSH

Use the FLUSH statement to force rows that a PUT statement buffered to be written to the database. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared.	Identifier, p. 4-189
cursor_id_var	Host variable that holds the value of *cursor_id*	Host variable must be a character data type.	Language specific

Usage

The PUT statement adds a row to a buffer, and the buffer is written to the database when it is full. Use the FLUSH statement to force the insertion when the buffer is not full.

If the program terminates without closing the cursor, the buffer is left unflushed. Rows placed into the buffer since the last flush are lost. Do not expect the end of the program to close the cursor and flush the buffer automatically. The following example shows a FLUSH statement:

```
FLUSH icurs
```

Error Checking FLUSH Statements

The **sqlca** structure contains information on the success of each FLUSH statement and the number of rows that are inserted successfully. The result of each FLUSH statement is contained in the fields of the sqlca: **sqlca.sqlcode**, **SQLCODE**, and **sqlca.sqlerrd[2]**.

When you use data buffering with an insert cursor, you do not discover errors until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set either to an error code or to zero (0) if no error occurs. The third element of the **SQLERRD** array is set to the number of rows that are successfully inserted into the database:

- If a block of rows is successfully inserted into the database, **SQLCODE** is set to zero (0) and **SQLERRD** to the count of rows.
- If an error occurs while the FLUSH statement is inserting a block of rows, **SQLCODE** shows which error, and **SQLERRD** contains the number of rows that were successfully inserted. (Uninserted rows are discarded from the buffer.)

Tip: *When you encounter an* **SQLCODE** *error, a corresponding* **SQLSTATE** *error value also exists. For information about how to get the message text, check the GET DIAGNOSTICS statement.*

To count the number of rows actually inserted into the database as well as the number not yet inserted

1. Prepare two integer variables, for example, **total** and **pending**.
2. When the cursor opens, set both variables to `0`.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a FLUSH statement executes or the cursor is closed, subtract the third field of the **SQLERRD** array from **pending**.

Related Information

Related statements: CLOSE, DECLARE, OPEN, and PREPARE

For a task-oriented discussion of FLUSH, see the *IBM Informix Guide to SQL: Tutorial*.

For information about the **sqlca** structure, see the *IBM Informix ESQL/C Programmer's Manual*.

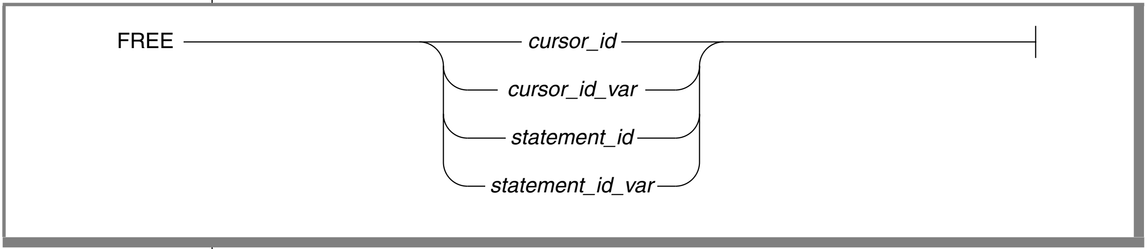
+

E/C

FREE

Use the FREE statement to release resources that are allocated to a prepared statement or to a cursor. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared.	Identifier, p. 4-189
cursor_id_var	Host variable that holds the value of cursor_id	Must be a character data type.	Language specific
statement_id	String that identifies an SQL statement	Must be defined in a previous PREPARE statement.	PREPARE, p. 2-527
statement_id_var	Host variable that identifies an SQL statement	Same restrictions as *statement_id*. Must be a character data type.	PREPARE, p. 2-527

Usage

FREE releases the resources that the database server and application-development tool allocated for a prepared statement or for a declared cursor.

If you declared a cursor for a prepared statement, FREE *statement_id* (or *statement_id_var*) releases only the resources in the application development tool; the cursor can still be used. The resources in the database server are released only when you free the cursor.

If you prepared a statement (but did not declare a cursor for it), FREE *statement_id* (or FREE *statement_id_var*) releases the resources in both the application development tool and the database server.

After you free a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following ESQL/C example shows the sequence of statements that is used to free an implicitly prepared statement:

```
EXEC SQL prepare sel_stmt from 'select * from orders';
...
EXEC SQL free sel_stmt;
```

The following ESQL/C example shows the sequence of statements that are used to release the resources of an explicitly prepared statement. The first FREE statement in this example frees the cursor. The second FREE statement in this example frees the prepared statement.

```
sprintf(demoselect, "%s %s",
   "select * from customer ",
   "where customer_num between 100 and 200");
EXEC SQL prepare sel_stmt from :demoselect;
EXEC SQL declare sel_curs cursor for sel_stmt;
EXEC SQL open sel_curs;
...
EXEC SQL close sel_curs;
EXEC SQL free sel_curs;
EXEC SQL free sel_stmt;
```

If you declared a cursor for a prepared statement, freeing the cursor releases only the resources in the database server. To release the resources for the statement in the application-development tool, use FREE *statement_id* (or FREE *statement_id_var*).

If a cursor is not declared for a prepared statement, freeing it releases the resources in both the application-development tool and the database server.

After a cursor is freed, it cannot be opened until it is declared again. The cursor should be explicitly closed before it is freed.

For an example of a FREE statement that frees a cursor, see the previous example.

Related Information

Related statements: CLOSE, DECLARE, EXECUTE, EXECUTE IMMEDIATE, OPEN, PREPARE, and SET AUTOFREE

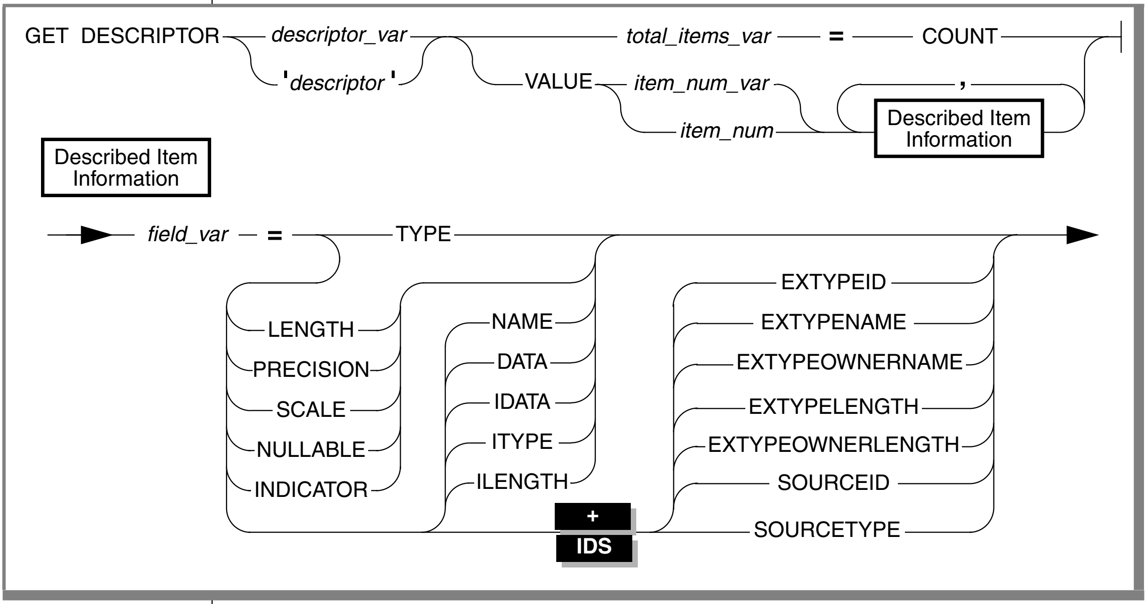
For a task-oriented discussion of the FREE statement, see the *IBM Informix Guide to SQL: Tutorial*.

+
E/C

GET DESCRIPTOR

Use the GET DESCRIPTOR statement to read from a system descriptor area. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area (SDA)	System-descriptor area must already have been allocated	Quoted String, p. 4-243
descriptor_var	Variable that stores *descriptor* value	Same restrictions as *descriptor*	Language specific
field_var	Host variable to receive the contents of a field from an SDA	Must be of type that can receive value of a specified SDA field.	Language specific
item_num	Unsigned ordinal number of an item described in the SDA	0 ≤ *item_num* ≤ (number of item descriptors in the SDA)	Literal Number, p. 4-216
item_num_ var	Host variable storing *item_num*	Must be an integer data type	Language specific
total_items_var	Host variable storing the number of items described in the SDA	Must be an integer data type	Language specific

Usage

Use GET DESCRIPTOR to accomplish any of the following tasks:

- Determine how many items are described in a system-descriptor area
- Determine the characteristics of each column or expression that is described in the system-descriptor area
- Copy a value from the system-descriptor area into a host variable after a FETCH statement

IDS

Use the GET DESCRIPTOR statement after you describe EXECUTE FUNCTION, INSERT, SELECT, or UPDATE statements with the DESCRIBE...USING SQL DESCRIPTOR statement. ♦

XPS

Use the GET DESCRIPTOR statement after you describe EXECUTE PROCEDURE, INSERT, or SELECT statements with the DESCRIBE...USING SQL DESCRIPTOR statement. ♦

The host variables that are used in the GET DESCRIPTOR statement must be declared in the BEGIN DECLARE SECTION of a program.

If an error occurs during the assignment to any identified host variable, the contents of the host variable are undefined.

Using the COUNT Keyword

Use the COUNT keyword to determine how many items are described in the system-descriptor area.

The following ESQL/C example shows how to use a GET DESCRIPTOR statement with a host variable to determine how many items are described in the system-descriptor area called **desc1**:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
int h_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc1' with max 20;

/* This section of program would prepare a SELECT or INSERT
 * statement into the s_id statement id.
*/
EXEC SQL describe s_id using sql descriptor 'desc1';

EXEC SQL get descriptor 'desc1' :h_count = count;
...
}
```

Using the VALUE Clause

Use the VALUE clause to obtain information about a described column or expression or to retrieve values that the database server returns in a system descriptor area.

The *item_num* must be greater than zero (0) and less than the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

Using the VALUE Clause After a DESCRIBE

After you describe a SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE), INSERT, or UPDATE statement, the characteristics of each column or expression in the select list of the SELECT statement, the characteristics of the values returned by the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, or the characteristics of each column in a INSERT or UPDATE statement are returned to the system-descriptor area. Each value in the system-descriptor area describes the characteristics of one returned column or expression.

The following ESQL/C example shows how to use a GET DESCRIPTOR statement to obtain data type information from the **demodesc** system-descriptor area:

```
EXEC SQL get descriptor 'demodesc' value :index
       :type = TYPE,
       :len = LENGTH,
       :name = NAME;
printf("Column %d: type = %d, len = %d, name = %s\n",
      index, type, len, name);
```

The value that the database server returns into the TYPE field is a defined integer. To evaluate the data type that is returned, test for a specific integer value. For additional information about integer data type values, see "Setting the TYPE or ITYPE Field" on page 2-673.

X/O

In X/Open mode, the X/Open code is returned to the TYPE field. You cannot mix the two modes because errors can result. For example, if a particular data type is not defined under X/Open mode but is defined for IBM Informix products, executing a GET DESCRIPTOR statement can result in an error.

In X/Open mode, a warning message appears if ILENGTH, IDATA, or ITYPE is used. It indicates that these fields are not standard X/Open fields for a system-descriptor area. ♦

If the TYPE of a fetched value is DECIMAL or MONEY, the database server returns the precision and scale information for a column into the PRECISION and SCALE fields after a DESCRIBE statement is executed. If the TYPE is *not* DECIMAL or MONEY, the SCALE and PRECISION fields are undefined.

Using the VALUE Clause After a FETCH

Each time your program fetches a row, it must copy the fetched value into host variables so that the data can be used. To accomplish this task, use a GET DESCRIPTOR statement after each fetch of each value in the select list. If three values exist in the select list, you need to use three GET DESCRIPTOR statements after each fetch (assuming you want to read all three values). The item numbers for each of the three GET DESCRIPTOR statements are `1`, `2`, and `3`.

The following ESQL/C example shows how you can copy data from the DATA field into a host variable (**result**) after a fetch. For this example, it is predetermined that all returned values are the same data type.

```
EXEC SQL get descriptor 'demodesc' :desc_count = count;
.. .
EXEC SQL fetch democursor using sql descriptor 'demodesc';
for (i = 1; i <= desc_count; i++)
   {
   if (sqlca.sqlcode != 0) break;
   EXEC SQL get descriptor 'demodesc' value :i :result = DATA;
   printf("%s ", result);
   }
printf("\n");
```

Fetching a NULL Value

When you use GET DESCRIPTOR after a fetch, and the fetched value is NULL, the INDICATOR field is set to `-1` (NULL). The value of DATA is undefined if INDICATOR indicates a NULL value. The host variable into which DATA is copied has an unpredictable value.

Using LENGTH or ILENGTH

If your DATA or IDATA field contains a character string, you must specify a value for LENGTH. If you specify LENGTH=`0`, LENGTH is automatically set to the maximum length of the string. The DATA or IDATA field might contain a literal character string or a character string that is derived from a character variable of CHAR or VARCHAR data type. This provides a method to determine the length of a string in the DATA or IDATA field dynamically.

If a DESCRIBE statement precedes a GET DESCRIPTOR statement, LENGTH is automatically set to the maximum length of the character field that is specified in your table.

This information is identical for ILENGTH. Use ILENGTH when you create a dynamic program that does not comply with the X/Open standard.

IDS

Describing an Opaque-Type Column

The DESCRIBE statement sets the following item-descriptor fields when the column to fetch has an opaque type as its data type:

- The EXTYPEID field stores the extended ID for the opaque type.
  This integer corresponds to a value in the **extended_id** column of the **sysxtdtypes** system catalog table.
- The EXTYPENAME field stores the name of the opaque type.
  This character value corresponds to a value in the **name** column of the row with the matching **extended_id** value in the **sysxtdtypes** system catalog table.
- The EXTYPELENGTH field stores the length of the opaque-type name.
  This integer is the length of the data type name (in bytes).
- The EXTYPEOWNERNAME field stores the name of the opaque-type owner. This character value corresponds to a value in the **owner** column of the row with the matching **extended_id** value in the **sysxtdtypes** system catalog table.
- The EXTYPEOWNERLENGTH field stores the length of the value in the EXTTYPEOWNERNAME field. This integer is the length, in bytes, of the name of the owner of the opaque type.

Use these field names with the GET DESCRIPTOR statement to obtain information about an opaque column.

IDS

Describing a Distinct-Type Column

The DESCRIBE statement sets the following item-descriptor fields when the column to fetch has a distinct type as its data type:

- The SOURCEID field stores the extended identifier for the source data type.

  This integer value corresponds to a value in the **source** column for the row of the **sysxtdtypes** system catalog table whose **extended_id** value matches that of the distinct data type. This field is only set if the source data type is an opaque data type.
- The SOURCETYPE field stores the data type constant for the source data type.

  This value is the data type constant (from the **sqlstypes.h** file) for the data type of the source type for the distinct type. The codes for the SOURCETYPE field are listed in the description of the TYPE field in the SET DESCRIPTOR statement. (For more information, see "Setting the TYPE or ITYPE Field" on page 2-673). This integer value must correspond to the value in the **type** column for the row of the **sysxtdtypes** system catalog table whose **extended_id** value matches that of the distinct data type.

Use these field names with the GET DESCRIPTOR statement to obtain information about a distinct-type column.

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, OPEN, PREPARE, PUT, and SET DESCRIPTOR

For more information on concepts that relate to the GET DESCRIPTOR statement, see the *IBM Informix ESQL/C Programmer's Manual*.

For more information on the **sysxtdtypes** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

+

E/C

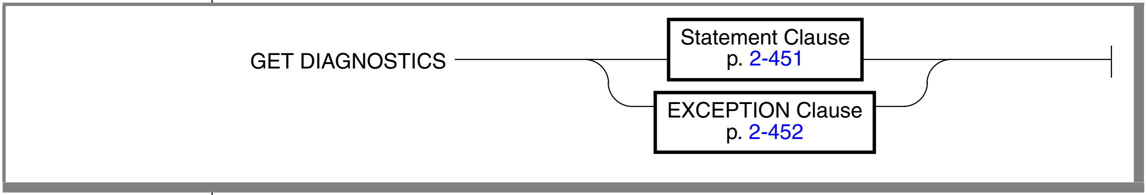
GET DIAGNOSTICS

Use the GET DIAGNOSTICS statement to return diagnostic information about executing an SQL statement. The GET DIAGNOSTICS statement uses one of the following two clauses:

- The Statement clause returns count and overflow information about errors and warnings that the most recent SQL statement generates.
- The EXCEPTION clause provides specific information about errors and warnings that the most recent SQL statement generates.

Use this statement with ESQL/C.

Syntax



Usage

The GET DIAGNOSTICS statement retrieves specified status information from the SQL diagnostics area (**SQLDA**) and retrieves either count and overflow information or other specified information on an exception. Using GET DIAGNOSTICS does not change the contents of the diagnostics area.

Using the SQLSTATE Error Status Code

When an SQL statement executes, an error status code is automatically generated. This code represents `success`, `failure`, `warning`, or `no data found`. This error status code is stored in a variable called SQLSTATE.

Class and Subclass Codes

The SQLSTATE status code is a five-character string that can contain only digits and uppercase letters.

The first two characters of the SQLSTATE status code indicate a class. The last three characters of the SQLSTATE code indicate a subclass. Figure 2-1 shows the structure of the SQLSTATE code. This example uses the value `08001`, where `08` is the class code and `001` is the subclass code. The value `08001` represents the error `unable to connect with database environment`.

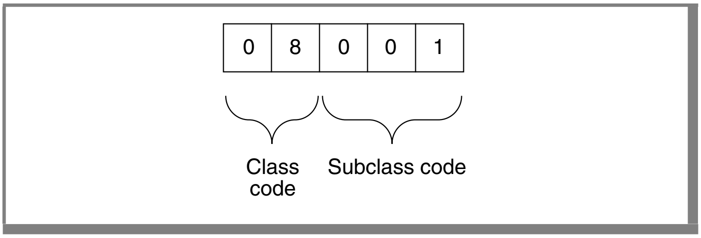


Figure 2-1
Structure of the SQLSTATE Code

The following table is a quick reference for interpreting class code values.

SQLSTATE Class Code Value	Outcome
00	Success
01	Success with warning
02	No data found
> 02	Error or warning

Support for the ANSI/ISO Standard for SQL

All status codes returned to the SQLSTATE variable are ANSI-compliant except in the following cases:

- SQLSTATE codes with a class code of `01` and a subclass code that begins with an `I` are Informix-specific warning messages.
- SQLSTATE codes with a class code of `IX` and any subclass code are Informix-specific error messages.
- SQLSTATE codes whose class code begins with a digit in the range 5 to 9 or with an uppercase letter in the range I to Z indicate conditions that are currently undefined by the ANSI/ISO standard for SQL. The only exception is that SQLSTATE codes whose class code is `IX` are Informix-specific error messages.

List of SQLSTATE Codes

This table describes the class codes, subclass codes, and the meaning of all valid warning and error codes associated with the SQLSTATE variable.

Class	Subclass	Meaning
00	000	Success
01	000	Success with warning
01	002	Disconnect error. Transaction rolled back
01	003	NULL value eliminated in set function
01	004	String data, right truncation
01	005	Insufficient item descriptor areas
01	006	Privilege not revoked
01	007	Privilege not granted
01	I01	Database has transactions
01	I03	ANSI-compliant database selected
01	I04	IBM Informix database server selected
01	I05	Float to decimal conversion was used
01	I06	Informix extension to ANSI-compliant standard syntax
01	I07	UPDATE or DELETE statement does not have a WHERE clause
01	I08	An ANSI keyword was used as a cursor name
01	I09	Number of Select list items is not equal to the number in INTO list
01	I10	Database server running in secondary mode
01	I11	Dataskip is turned on
02	000	No data found
07	000	Dynamic SQL error
07	001	USING clause does not match dynamic parameters
07	002	USING clause does not match target specifications
07	003	Cursor specification cannot be executed
07	004	USING clause is required for dynamic parameters
07	005	Prepared statement is not a cursor specification
07	006	Restricted data type attribute violation
07	008	Invalid descriptor count

(1 of 3)

Class	Subclass	Meaning
07	009	Invalid descriptor index
08	000	Connection exception
08	001	Database server rejected the connection
08	002	Connection name in use
08	003	Connection does not exist
08	004	Client unable to establish connection
08	006	Transaction rolled back
08	007	Transaction state unknown
08	S01	Communication failure
0A	000	Feature not supported
0A	001	Multiple server transactions
21	000	Cardinality violation
21	S01	Insert value list does not match column list
21	S02	Degree of derived table does not match column list
22	000	Data exception
22	001	String data, right truncation
22	002	NULL value, no indicator parameter
22	003	Numeric value out of range
22	005	Error in assignment
22	027	Data exception trim error
22	012	Division by zero (0)
22	019	Invalid escape character
22	024	Unterminated string
22	025	Invalid escape sequence
23	000	Integrity constraint violation
24	000	Invalid cursor state
25	000	Invalid transaction state
2B	000	Dependent privilege descriptors still exist
2D	000	Invalid transaction termination
26	000	Invalid SQL statement identifier

(2 of 3)

Class	Subclass	Meaning
2E	000	Invalid connection name
28	000	Invalid user-authorization specification
33	000	Invalid SQL descriptor name
34	000	Invalid cursor name
35	000	Invalid exception number
37	000	Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE
3C	000	Duplicate cursor name
40	000	Transaction rollback
40	003	Statement completion unknown
42	000	Syntax error or access violation
S0	000	Invalid name
S0	001	Base table or view table already exists
S0	002	Base table not found
S0	011	Index already exists
S0	021	Column already exists
S1	001	Memory allocation failure
IX	000	Informix reserved error message

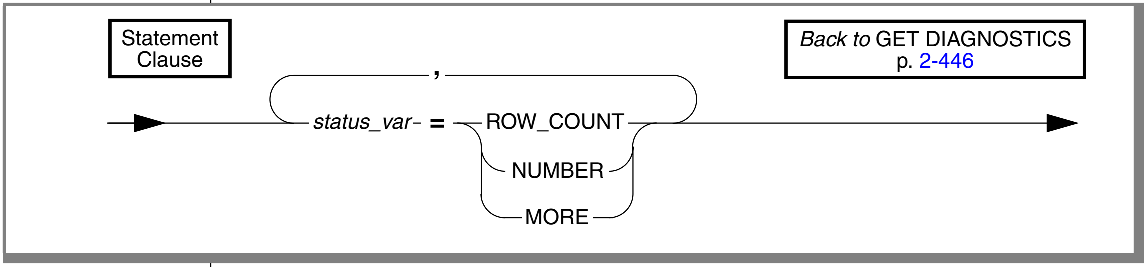
(3 of 3)

Using SQLSTATE in Applications

You can use a built-in variable, called SQLSTATE, that you do not have to declare in your program. SQLSTATE contains the status code, essential for error handling, that is generated every time your program executes an SQL statement. SQLSTATE is created automatically. You can examine the SQLSTATE variable to determine whether an SQL statement was successful. If the SQLSTATE variable indicates that the statement failed, you can execute a GET DIAGNOSTICS statement to obtain additional error information.

For an example of how to use an SQLSTATE variable in a program, see "Using GET DIAGNOSTICS for Error Checking" on page 2-457.

Statement Clause



Element	Purpose	Restrictions	Syntax
status_var	Host variable to receive status information about the most recent SQL statement for the specified status field name	Must match data type of the field.	Language specific

When retrieving count and overflow information, GET DIAGNOSTICS can deposit the values of the three statement fields into a corresponding host variable. The host-variable data type must be the same as that of the requested field. The following keywords represent these three fields.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type
MORE	Character	Y or N	char[2]
NUMBER	Integer	1 to 35,000	int
ROW_COUNT	Integer	0 to 999,999,999	int

Using the MORE Keyword

Use the MORE keyword to determine if the most recently executed SQL statement performed the following actions:

- Stored all the exceptions that it detected in the **SQLDA** diagnostics area. If so, GET DIAGNOSTICS returns a value of N.
- Detected more exceptions than it stored in the **SQLDA** diagnostics area. If so, GET DIAGNOSTICS returns a value of Y. (The value of MORE is always N.)

Using the ROW_COUNT Keyword

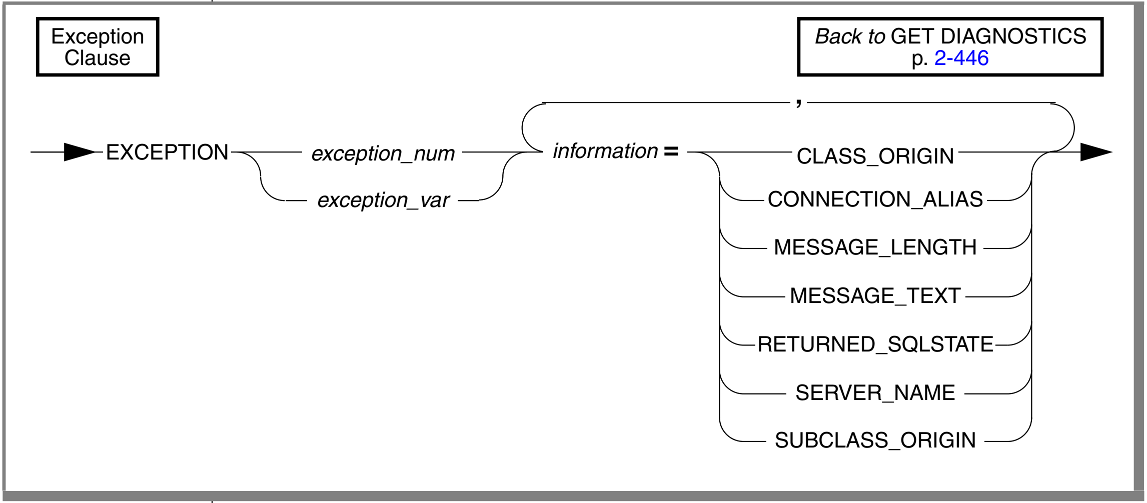
The ROW_COUNT keyword returns the number of rows the most recently executed DML statement processed. ROW_COUNT counts these rows:

- Inserted into a table
- Updated in a table
- Deleted from a table

Using the NUMBER Keyword

The NUMBER keyword returns the number of exceptions that the most recently executed SQL statement raised. The NUMBER field can hold a value from `1` to `35,000`, depending on how many exceptions are counted.

EXCEPTION Clause



Element	Purpose	Restrictions	Syntax
exception_num	Number of exceptions	Integer from `1` to `35,000`	Literal number, p. 4-216
exception_var	Variable storing *exception_num*	Must be SMALLINT or INT	Language specific
information	Host variable to receive the value of a specified exception field	Data type must match that of requested field	Language specific

The *exception_num* literal indicates one of the exception values from the number of exceptions that the NUMBER field in the Statement clause returns.

When retrieving exception information, GET DIAGNOSTICS writes the values of each of the seven fields into corresponding host variables. These fields are located in the diagnostics area and are derived from an exception raised by the most recent SQL statement.

The host-variable data type must be the same as that of the requested field. The following table describes the seven exception information fields.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type
RETURNED_SQLSTATE	Character	SQLSTATE value	char[6]
CLASS_ORIGIN	Character	String	char[255]
SUBCLASS_ORIGIN	Character	String	char[255]
MESSAGE_TEXT	Character	String	char[255]
MESSAGE_LENGTH	Integer	Numeric value	int
SERVER_NAME	Character	String	char[255]
CONNECTION_NAME	Character	String	char[255]

The application specifies the exception by number, using either an unsigned integer or an integer host variable (an exact numeric with a scale of 0). An exception with a value of `1` corresponds to the SQLSTATE value set by the most recent SQL statement other than GET DIAGNOSTICS. The association between other exception numbers and other exceptions raised by that SQL statement is undefined. Thus, no set order exists in which the diagnostic area can be filled with exception values. You always get at least one exception, even if the SQLSTATE value indicates success.

If an error occurs within the GET DIAGNOSTICS statement (that is, if an invalid exception number is requested), the Informix internal SQLCODE and SQLSTATE variables are set to the value of that exception. In addition, the GET DIAGNOSTICS fields are undefined.

Using the RETURNED_SQLSTATE Keyword

The RETURNED_SQLSTATE keyword returns the SQLSTATE value that describes the exception.

Using the CLASS_ORIGIN Keyword

Use the CLASS_ORIGIN keyword to retrieve the class portion of the RETURNED_SQLSTATE value. If the International Standards Organization (ISO) standard defines the class, the value of CLASS_ORIGIN is equal to ISO 9075. Otherwise, the value returned by CLASS_ORIGIN is defined by Informix and cannot be ISO 9075. ANSI SQL and ISO SQL are synonymous.

Using the SUBCLASS_ORIGIN Keyword

The SUBCLASS_ORIGIN keyword returns data on the RETURNED_SQLSTATE subclass. (This value is `ISO 9075` if the ISO standard defines the subclass.)

Using the MESSAGE_TEXT Keyword

The MESSAGE_TEXT keyword returns the message text of the exception (for example, an error message).

Using the MESSAGE_LENGTH Keyword

The MESSAGE_LENGTH keyword returns the length in bytes of the current message text string.

Using the SERVER_NAME Keyword

The SERVER_NAME keyword returns ne the name of the database server associated with a CONNECT or DATABASE statement. GET DIAGNOSTICS updates the SERVER_NAME field when any of the following events occur:

- A CONNECT statement successfully executes
- A SET CONNECTION statement successfully executes
- A DISCONNECT statement successfully terminates the current connection
- A DISCONNECT ALL statement fails

The SERVER_NAME field is not updated, however, after these events:

- A CONNECT statement fails
- A DISCONNECT statement fails (but this does not include the DISCONNECT ALL statement)
- A SET CONNECTION statement fails

The SERVER_NAME field retains the value set in the previous SQL statement. If any of the preceding conditions occur on the first SQL statement that executes, the **SERVER_NAME** field is blank.

The Contents of the SERVER_NAME Field

The SERVER_NAME field contains different information after you execute the following statements.

Executed Statement	SERVER_NAME Field Contents
CONNECT	Contains the name of the database server to which you connect or fail to connect. Field is blank if you do not have a current connection or if you make a default connection.
SET CONNECTION	Contains the name of the database server to which you switch or fail to switch.
DISCONNECT	Contains the name of the database server from which you disconnect or fail to disconnect. If you disconnect and then you execute a DISCONNECT statement for a connection that is not current, the SERVER_NAME field remains unchanged.
DISCONNECT ALL	Sets the field to blank if the statement executes successfully. If the statement fails, SERVER_NAME contains the names of all the database servers from which you did not disconnect. (This information does not mean that the connection still exists.)

If CONNECT succeeds, SERVER_NAME is set to one of the following values:

- The **INFORMIXSERVER** value (if the connection is to a default database server; that is, if CONNECT specified no database server).
- The name of the database server (if the connection is to a specific database server).

The DATABASE Statement

When you execute a DATABASE statement, the SERVER_NAME field contains the name of the database server on which the database resides.

Using the CONNECTION_NAME Keyword

Use the CONNECTION_NAME keyword to specify a name for the connection used in your CONNECT or DATABASE statement.

When the CONNECTION_NAME Keyword Is Updated

GET DIAGNOSTICS updates the CONNECTION_NAME field when the following situations occur:

- A CONNECT statement successfully executes
- A SET CONNECTION statement successfully executes
- A DISCONNECT statement successfully executes at the current connection

  GET DIAGNOSTICS fills the CONNECTION_NAME field with blanks because no current connection exists.
- A DISCONNECT ALL statement fails

When the CONNECTION_NAME Is Not Updated

The CONNECTION_NAME field is not updated in the following cases:

- A CONNECT statement fails
- A DISCONNECT statement fails (but this does not include the DISCONNECT ALL statement)
- A SET CONNECTION statement fails

The CONNECTION_NAME field retains the value set in the previous SQL statement. If any of the preceding conditions occur on the first SQL statement that executes, the CONNECTION_NAME field is blank.

DATABASE Statement

After a DATABASE statement, the CONNECTION_NAME field is blank.

The Contents of the CONNECTION_NAME Field

The CONNECTION_NAME field contains different information after you execute the following statements.

Executed Statement	CONNECTION_NAME Field Contents
CONNECT	Contains connection name specified in the CONNECT statement, to which you connect or fail to connect. The field is blank for no current connection or a default connection.
SET CONNECTION	Contains the connection name specified in the CONNECT statement, to which you switch or fail to switch
DISCONNECT	Contains the connection name specified in the CONNECT statement, from which you disconnect or fail to disconnect. If you disconnect, and then execute a DISCONNECT statement for a connection that is not current, the CONNECTION_NAME field remains unchanged.
DISCONNECT ALL	Contains no information if the statement executes successfully. If the statement does not execute successfully, the CONNECTION_NAME field contains the names of all the connections, specified in your CONNECT statement, from which you did not disconnect. However, this information does not mean that the connection still exists.

If CONNECT is successful, CONNECTION_NAME takes one of these values:

- The name of the database environment as specified in the CONNECT statement if the CONNECT does not include the AS clause
- The name of the connection (identifier after the AS keyword) if the CONNECT includes the AS clause

Using GET DIAGNOSTICS for Error Checking

GET DIAGNOSTICS returns values from **SQLDA** fields. For each field that you want to access, you must supply a host variable with a compatible data type.

The following example illustrates how to use the GET DIAGNOSTICS statement to display error information. The example shows an ESQL/C error display routine called **disp_sqlstate_err( )**.

```
void disp_sqlstate_err()
{
int j;
EXEC SQL BEGIN DECLARE SECTION;
    int exception_count;
    char overflow[2];
    int exception_num=1;
    char class_id[255];
    char subclass_id[255];
    char message[255];
    int messlen;
    char sqlstate_code[6];
    int i;
EXEC SQL END DECLARE SECTION;
    printf("---------------------------------");
    printf("-------------------------\n");
    printf("SQLSTATE: %s\n",SQLSTATE);
    printf("SQLCODE: %d\n", SQLCODE);
    printf("\n");
    EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
    printf("EXCEPTIONS:  Number=%d\t", exception_count);
    printf("More? %s\n", overflow);
    for (i = 1; i <= exception_count; i++)
    {
        EXEC SQL get diagnostics  exception :i
            :sqlstate_code = RETURNED_SQLSTATE,
            :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
            :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
        printf("- - - - - - - - - - - - - - - - - - - - - -\n");
        printf("EXCEPTION %d: SQLSTATE=%s\n", i,
            sqlstate_code);
        message[messlen-1] ='\0';
        printf("MESSAGE TEXT: %s\n", message);
        j = stleng(class_id);
        while((class_id[j] == '\0') ||
              (class_id[j] == ' '))
            j--;
        class_id[j+1] = '\0';
        printf("CLASS ORIGIN: %s\n",class_id);
        j = stleng(subclass_id);
        while((subclass_id[j] == '\0') ||
              (subclass_id[j] == ' '))
            j--;
        subclass_id[j+1] = '\0';
        printf("SUBCLASS ORIGIN: %s\n",subclass_id);
    }
    printf("---------------------------------");
    printf("-------------------------\n");
}
```

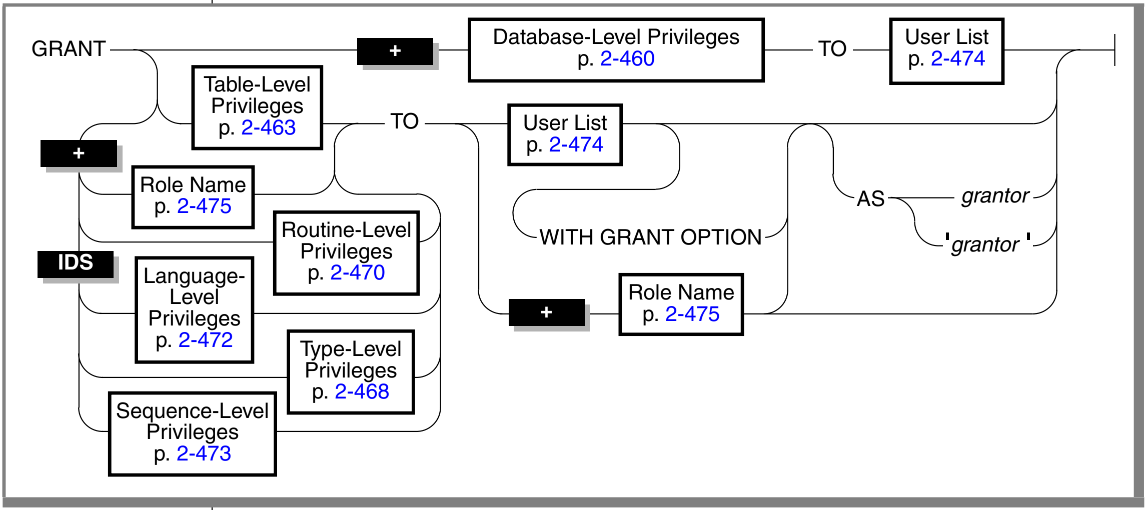
Related Information

For a task-oriented discussion of error handling and the SQLSTATE variable, see the *IBM Informix Guide to SQL: Tutorial*. For a discussion of concepts related to the GET DIAGNOSTICS statement and the SQLSTATE variable, see the *IBM Informix ESQL/C Programmer's Manual*.

GRANT

The GRANT statement can assign privileges to users of the database.

Syntax


GRANT
+
Database-Level Privileges p. 2-460
TO
User List p. 2-474
Table-Level Privileges p. 2-463
TO
User List p. 2-474
+
Role Name p. 2-475
WITH GRANT OPTION
AS
grantor
'grantor'
IDS
Routine-Level Privileges p. 2-470
Language-Level Privileges p. 2-472
+
Role Name p. 2-475
Type-Level Privileges p. 2-468
Sequence-Level Privileges p. 2-473

Element	Purpose	Restrictions	Syntax
grantor	Name of user who can use REVOKE to undo the effects of this GRANT statement. If the AS clause is omitted, the default is the login name of the user who executes the GRANT statement.	Must be a valid authorization identifier.	Owner Name, p. 4-234.

Usage

The GRANT statement extends privileges to other users that would normally accrue only to the DBA or to the creator of an object. Subsequent GRANT statements do not affect privileges that have already been granted to a user.

You can use the GRANT statement for operations like the following:

- Authorize others to use or administrate a database that you create
- Allow others to view, alter, or drop a table, synonym, view or (for Dynamic Server only) a sequence object that you create

- Allow others to use a data type or the SPL language, or (for Dymanic Server only) to execute a user-defined routine (UDR) that you create
- Give a role name and its privileges to one or more users

You can grant privileges to a previously created role. You can grant a role to individual users or to another role.

Privileges that you grant remain in effect until you cancel them with a REVOKE statement. Only the grantor of a privilege can revoke that privilege. The grantor is the person who issues the GRANT statement, unless the AS *grantor* clause transfers the right to revoke to e another user.

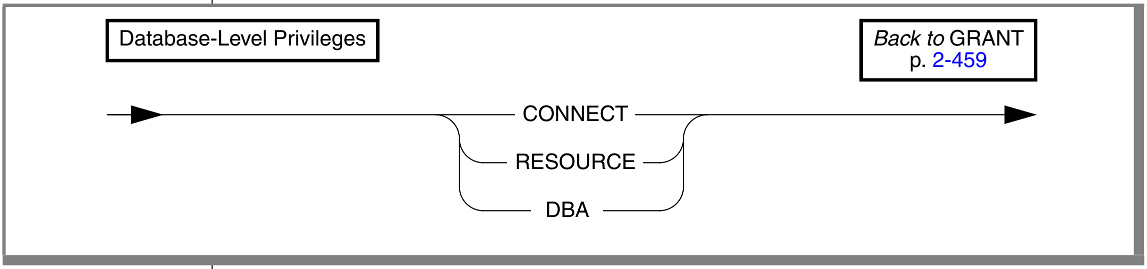
Only the owner of an object or a user who has been explicitly granted permission WITH GRANT OPTION can grant permissions on an object. Having DBA privileges is not sufficient. As DBA, however, one can grant a privilege on behalf of another user by using AS clause. For database objects whose owner is not a user recognized by the operating system (for example, the **informix** authorization identifier), the AS clause is useful.

The keyword PUBLIC extends a GRANT to all users. If you want to restrict to a particular user the privileges that PUBLIC already holds, you must first revoke the right of PUBLIC to those privileges.

When database-level privileges conflict with table-level privileges, the more restrictive privileges take precedence.

To grant privileges on one or more fragments of a table that has been fragmented by expression, see "GRANT FRAGMENT" on page 2-480.

Database-Level Privileges



When you create a database with the CREATE DATABASE statement, you are the owner. and automatically receive all database-level privileges.

The database remains inaccessible to other users until you, as DBA, grant database privileges.

As database owner, you also automatically receive table-level privileges on all tables in the database. For more information about table-level privileges, see "Table-Level Privileges" on page 2-463.

Warning: *Although user **informix** and DBAs can modify most system catalog tables (only user **informix** can modify **systables**), it is strongly recommended that you do not update, delete, or alter any rows in them. Modifying the system catalog tables can destroy the integrity of the database. The use of the ALTER TABLE statement to modify the size of the next extent of system catalog tables is not supported.*

Database access levels are, from lowest to highest, Connect, Resource, and DBA. Use the corresponding keyword to grant a level of access privilege.

Privilege	Functions
CONNECT	Lets you query and modify data You can modify the database schema if you own the database object you want to modify. Any user with the Connect privilege can perform the following functions: ■ Connect to the database with the CONNECT statement or another connection statement ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided the user has the necessary table-level privileges ■ Create views, provided the user has the Select privilege on the underlying tables ■ Create synonyms ■ Create temporary tables and create indexes on the temporary tables ■ Alter or drop a table or an index, provided the user owns the table or index (or has Alter, Index, or References privileges on the table) ■ Grant privileges on a table or view, provided the user owns the table (or was given privileges on the table with the WITH GRANT OPTION keyword)

(1 of 2)

Privilege	Functions
RESOURCE	Lets you extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following functions: ■ Create new tables ■ Create new indexes ■ Create new UDRs ■ Create new data types
DBA	Has all the capabilities of the Resource privilege and can perform the following additional operations: ■ Grant any database-level privilege, including the DBA privilege, to another user ■ Grant any table-level privilege to another user or to a role ■ Grant a role to a user or to another role ■ Revoke a privilege whose grantor you specify as the *revoker* in the AS clause of the REVOKE statement. ■ Restrict the Execute privilege to DBAs when registering a UDR ■ Execute the SET SESSION AUTHORIZATION statement ■ Use the NEXT SIZE keywords to alter extent sizes in the system catalog tables ■ Create any database object ■ Create tables, views, and indexes, designating another user as owner of these objects ■ Alter, drop, or rename database objects, regardless of who owns them ■ Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement ■ Execute DROP DATABASE and RENAME DATABASE statements ■ Insert, delete, or update rows of any system catalog table except **systables**

(2 of 2)

User **informix** has the privilege required to alter tables in the system catalog, including the **systables** table.

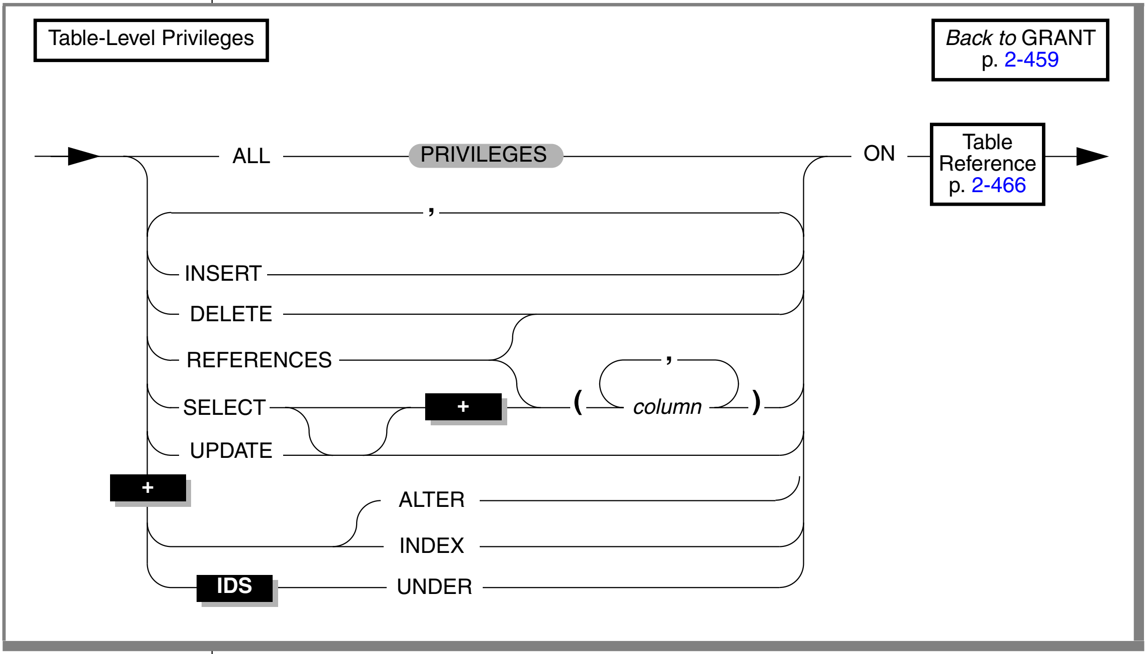
The following example uses the PUBLIC keyword to grant the Connect privilege on the currently active database to all users:

```
GRANT CONNECT TO PUBLIC
```

Table-Level Privileges

When you create a table with the CREATE TABLE statement, you are the table owner and automatically receive all table-level privileges. You cannot transfer ownership to another user, but you can grant table-level privileges to another user or to a role. (See, however, "RENAME TABLE" on page 2-554, which can change both the name and the ownership of a table.)

A person with the database-level DBA privilege automatically receives all table-level privileges on every table in that database.



Element	Purpose	Restrictions	Syntax
column	Column on which a Select, Update, or References privilege is granted. If you omit *column*, the default scope is all columns in the specified table.	Must exist.	Identifier, p. 4-189

The table that follows lists keywords for granting table-level privileges.

Privilege	Purpose
ALTER	Lets you add or delete columns, modify column data types, add or delete constraints, change the locking mode of the table from PAGE to ROW, or add or drop a corresponding ROW type name for your table. It also lets you set the database object mode of unique indexes and constraints to the enabled, disabled, or filtering mode. In addition, this privilege lets you set the database object mode of nonunique indexes and triggers to the enabled or disabled modes. You must have the Resource privilege to use the Alter privilege. In addition, you also need the Usage privilege for any user-defined data type affected by the ALTER TABLE statement.
INSERT	Lets you insert rows
DELETE	Lets you delete rows
SELECT	Lets you name any column in SELECT statements. You can restrict the Select privilege to one or more columns by listing them.
UPDATE	Lets you name any column in UPDATE statements. You can restrict this privilege to one or more columns by listing them.
REFERENCES	Lets you reference columns in referential constraints. You must have the Resource privilege to take advantage of the References privilege. (You can add, however, a referential constraint during an ALTER TABLE statement. This action does not require the Resource privilege on the database.) You can restrict the References privilege to one or more columns by listing the columns explicitly. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to place cascading deletes on a table.
INDEX	Lets you create permanent indexes. You must have the Resource privilege to use the Index privilege. (Any user with the Connect privilege can create an index on temporary tables.)
UNDER (IDS only)	Lets you create subtables under a typed table
ALL	Provides all privileges. The PRIVILEGES keyword is optional.

You can narrow the scope of a Select, Update, or References privilege by specifying the columns to which the privilege applies.

Specify the keyword PUBLIC as *user* if you want a GRANT statement to apply to all users.

Some simple examples that follow can help illustrate how table-level privileges are granted with the GRANT statement.

The following statement grants the privilege to delete and select values in any column in the table **customer** to users **mary** and **john**. It also grants the Update privilege, but only for columns **customer_num**, **fname**, and **lname**.

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
   ON customer TO mary, john
```

To grant the same privileges as those above to all authorized users, use the keyword PUBLIC as the following example shows:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
   ON customer TO PUBLIC
```

IDS

Suppose a user named **mary** has created a typed table named **tab1**. By default, only user **mary** can create subtables under the **tab1** table. If **mary** wants to grant the ability to create subtables under the **tab1** table to a user named **john**, **mary** must enter the following GRANT statement:

```
GRANT UNDER ON tab1 TO john
```

After receiving the Under privilege on table **tab1**, user **john** can create one or more subtables under **tab1**. ♦

Effect of the ALL Keyword

The ALL keyword grants all possible table-level privileges to the specified user. If any or all of the table-level privileges do not exist for the grantor, the GRANT statement with the ALL keyword succeeds (in the sense of SQLCODE being set to zero, even if the "possible" privileges are an empty set for the grantor on the table). In this case, however, the following SQLSTATE warning is returned:

```
01007 - Privilege not granted.
```

For example, assume that user **ted** has the Select and Insert privileges on the **customer** table with the authority to grant those privileges to other users.

User **ted** wants to grant all table-level privileges to user **tania**. So user **ted** issues the following GRANT statement:

```
GRANT ALL ON customer TO tania
```

This statement executes successfully but returns SQLSTATE code `01007` for the following reasons:

- The statement succeeds in granting the Select and Insert privileges to user **tania** because user **ted** has those privileges and the right to grant those privileges to other users.
- The other privileges implied by the ALL keyword were not grantable by user **ted** and, therefore, were not granted to user **tania**.

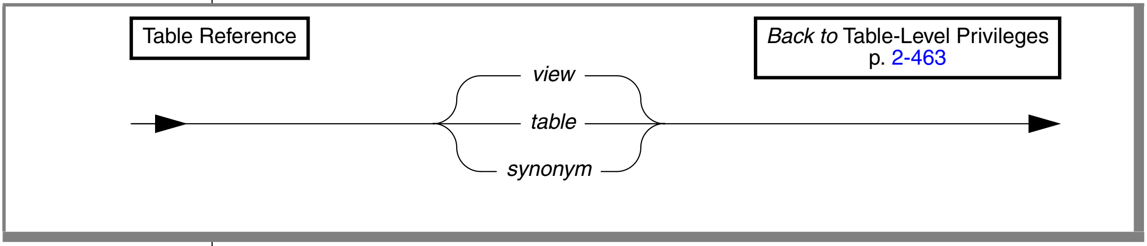
IDS

With Dynamic Server, if you grant all table-level privileges with the ALL keyword, the privileges includes the Under privilege only if the table is a typed table. The grant of ALL privileges does not include the Under privilege if the table is not based on a ROW type.

If the table owner grants ALL privileges on a traditional relational table and later changes that table to a typed table, the table owner must explicitly grant the Under privilege to allow other users to create subtables under it. ♦

Table Reference

You grant table-level privileges directly by referencing the table name or an existing synonym. You can also grant table-level privileges on a view.



Element	Purpose	Restrictions	Syntax
synonym, table, view	Synonym, table, or view on which privileges are granted	The table, view, or synonym must exist.	Database Object Name, p. 4-46

The object on which you grant privileges must reside in the current database.

Privileges on Table Name and Synonym Name

Normally, when you create a table in a database that is *not* ANSI compliant, **public** receives Select, Insert, Delete, Under, and Update privileges for that table and its synonyms. (The **NODEFDAC** environment variable, when set to `yes`, prevents **public** from automatically receiving table-level privileges.)

To allow access to only certain users, explicitly revoke those privileges **public** automatically receives and then grant only those you want, as the following example shows:

```
REVOKE ALL ON customer FROM PUBLIC;
GRANT ALL ON customer TO john, mary;
GRANT SELECT (fname, lname, company, city)
   ON customer TO PUBLIC
```

ANSI

In an ANSI-compliant database, if you create a table, only you, its owner, have any table-level privileges until you explicitly grant them to others. ♦

As explained in the next section, "Privileges on a View," **public** does *not* automatically receive any privileges for a view that you create.

Privileges on a View

You must have at least the Select privilege on a table or columns to create a view on that table.

For views that reference only tables in the current database, if the owner of a view loses the Select privilege on any table underlying the view, the view is dropped.

You have the same privileges for the view that you have for the table or tables contributing data to the view. For example, if you create a view from a table to which you have only Select privileges, you can select data from your view but you cannot delete or update data.

For detailed information on how to create a view, see "CREATE VIEW" on page 2-310.

When you create a view, only you have access to table data through that view. Even users who have privileges on the base table of the view do not automatically receive privileges for the view.

You can grant (or revoke) privileges on a view only if you are the owner of the underlying tables or if you received these privileges on the table with the right to grant them (the WITH GRANT OPTION keyword). You must explicitly grant those privileges within your authority; **public** does not automatically receive privileges on a view.

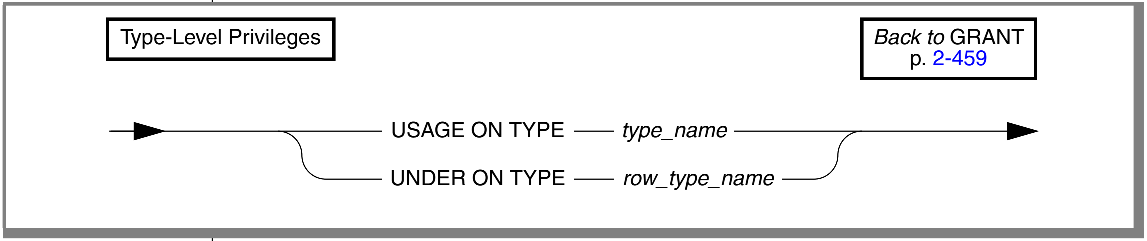
The creator of a view can explicitly grant Select, Insert, Delete, and Update privileges for the view to other users or to a role name. You cannot grant Index, Alter, Under, or References privileges on a view (or the All privilege because All includes Index, References, and Alter).

IDS

Type-Level Privileges

You can specify two privileges on data types:

- The Usage privilege on a user-defined data type
- The Under privilege on a named ROW type.



Element	Purpose	Restrictions	Syntax
row_type_name	Named row type on which the Under privilege is granted	Named ROW data type must exist.	Data Type, p. 4-49
type_name	User-defined data type on which the Usage privilege is granted	User-defined data type must exist.	Data Type, p. 4-49

To find out what privileges exist on a data type, check the **sysxtdtypes** system catalog table for the owner name and the **sysxtdtypeauth** system catalog table for additional type privileges that might have been granted. For more information on system catalog tables, see the *IBM Informix Guide to SQL: Reference*.

USAGE Privilege

You own any user-defined data type (UDT) that you create. As owner, you automatically receive the Usage privilege on that data type and can grant the Usage privilege to others so that they can reference the type name or reference data of that type in SQL statements. DBAs can also grant the Usage privilege for user-defined data types.

If you grant the Usage privilege to a user or role that has Alter privileges, that person can add a column to the table that contains values of your UDT.

Without privileges from the GRANT statement, any user can issue SQL statements that reference built-in data types. In contrast, a user must receive an explicit Usage privilege from a GRANT statement to use a distinct data type, even if the distinct type is based on a built-in type.

For more information about user-defined types, see "CREATE OPAQUE TYPE" on page 2-169, "CREATE DISTINCT TYPE" on page 2-115, the discussion of data types in the *IBM Informix Guide to SQL: Reference* and the *IBM Informix Database Design and Implementation Guide*.

UNDER Privilege

You own any named ROW type that you create. If you want other users to be able to create subtypes under this named ROW type, you must grant these users the Under privilege on your named ROW type.

For example, suppose that you create a ROW type named **rtype1**:

```
CREATE ROW TYPE rtype1 (cola INT, colb INT)
```

If you want another user named **kathy** to be able to create a subtype under this named ROW type, you must grant the Under privilege on this named ROW type to user **kathy**:

```
GRANT UNDER on rtype1 to kathy
```

Now user **kathy** can create another ROW type under the **rtype1** ROW type even though **kathy** is not the owner of the **rtype1** ROW type:

```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1
```

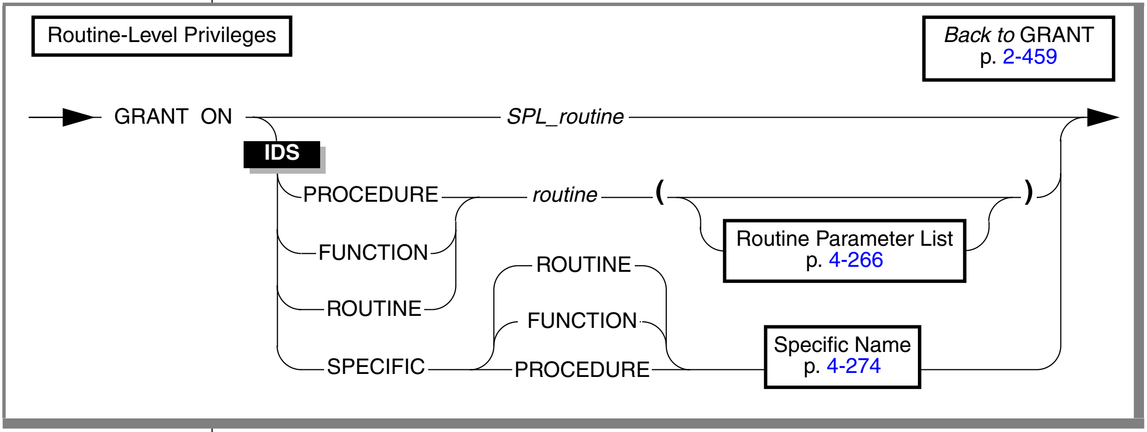
For more about named ROW types, see "CREATE ROW TYPE" on page 2-198, and the discussion of data types in the *IBM Informix Guide to SQL: Reference* and the *IBM Informix Database Design and Implementation Guide*.

Routine-Level Privileges

When you create a user-defined routine (UDR) with the CREATE FUNCTION or CREATE PROCEDURE statement, you own, and automatically receive the Execute privilege on that UDR.

The Execute privilege allows you to invoke the UDR with an EXECUTE FUNCTION or EXECUTE PROCEDURE statement, whichever is appropriate, or with a CALL statement in an SPL routine. The Execute privilege also allows you to use a user-defined function in an expression, as in this example:

```
SELECT * FROM table WHERE in_stock(partnum) < 20
```



Element	Purpose	Restrictions	Syntax
routine	A user-defined routine	Must exist.	Database Object Name, p. 4-46
SPL_routine	An SPL routine	Must be unique in the database.	Database Object Name, p. 4-46

Whether you must grant the Execute privilege explicitly depends on the following conditions:

- If you have DBA-level privileges, you can use the DBA keyword of CREATE FUNCTION or CREATE PROCEDURE to restrict the default Execute privilege to users with the DBA database-level privilege. You must explicitly grant the Execute privilege on that UDR to users who do not have the DBA privilege.
- If you have the Resource database-level privilege but not the DBA privilege, you cannot use the DBA keyword when you create a UDR:
  - When you create a UDR in a database that is *not* ANSI compliant, **public** can execute that UDR. You do not need to issue a GRANT statement for the Execute privilege.
  - The **NODEFDAC** environment variable, when set to `yes`, prevents **public** from executing your UDR until you explicitly grant the Execute privilege.
- ANSI In an ANSI-compliant database, the creator of a UDR must explicitly grant the Execute privilege on that UDR. ♦

IDS

Because of routine overloading, you can grant the Execute privilege on more than one UDR at a time. The following table explains the purpose of the keywords that you specify.

Keyword	Purpose
SPECIFIC	Grants the Execute privilege for the UDR that *specific name* identifies
FUNCTION	Grants the Execute privilege for all user-defined functions with the specified *routine name* (and parameter types that match *routine parameter list,* if supplied)
PROCEDURE	Grants the Execute privilege for all user-defined procedures with the specified *routine name (*and parameter types that match *routine parameter list,* if supplied)
ROUTINE	Grants the Execute privilege for all user-defined functions and all user-defined procedures with the specified *routine name (*and parameter types that match *routine parameter list,* if supplied)

If both a user-defined function and a user-defined procedure have the same name and list of parameter types, you can grant the Execute privilege to both with the keyword ROUTINE.

To limit the Execute privilege to one version of the same routine name, use the FUNCTION, PROCEDURE, or SPECIFIC keyword.

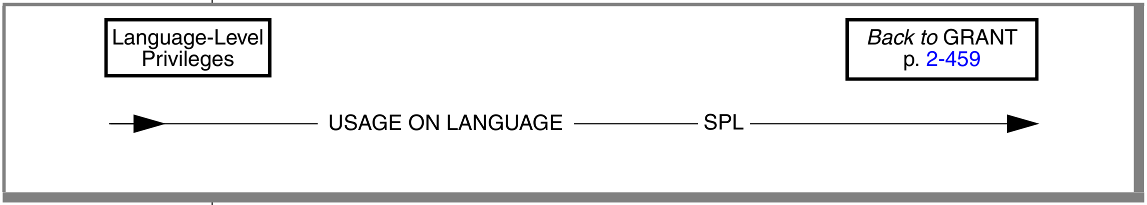
To limit the Execute privilege to a UDR that accepts certain data types as arguments, include the routine parameter list or use the SPECIFIC keyword to introduce the specific name of a UDR.

Tip: *If an external function has a negator function, you must grant the Execute privilege on both the external function and its negator function before users can execute the external function.* ♦

IDS

Language-Level Privileges

A user must have the Usage privilege on a language to register a user-defined routine (UDR) that is written in that language.



In this release of Dynamic Server, only the SPL keyword is supported within the USAGE ON LANGUAGE clause.

When a user executes a CREATE FUNCTION or CREATE PROCEDURE statement to register a UDR that is written in SPL, the database server verifies that the user has the Usage privilege on the language in which the UDR is written. (In this release of Dynamic Server, the C language and the Java language do not require Usage privilege.) For information on other privileges that these statements require, see "CREATE FUNCTION" on page 2-133 and "CREATE PROCEDURE" on page 2-182.

Usage Privilege in Stored Procedure Language

The Usage privilege on SPL is granted to PUBLIC by default. Only user **informix**, the DBA, or a user who was granted the Usage privilege WITH GRANT OPTION can grant the Usage privilege in SPL to another user.

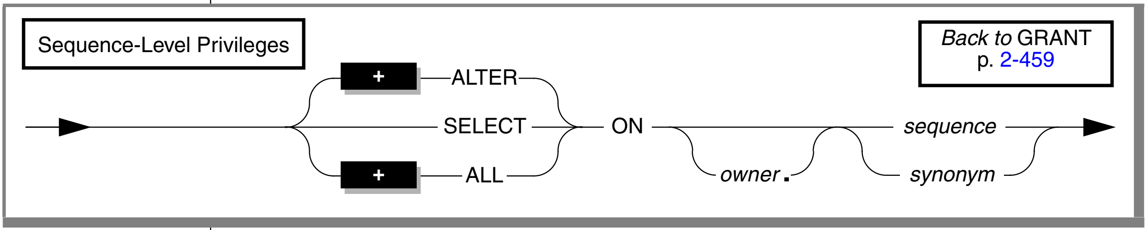
In the following example, assume that the default Usage privilege in SPL was revoked from PUBLIC and the DBA wants to grant the Usage privilege in SPL to the role named **developers**:

```
GRANT USAGE ON LANGUAGE SPL TO developers
```

IDS

Sequence-Level Privileges

Although a sequence is implemented as a table, only a subset of the table privileges (page 2-463) can be granted on a sequence. You can grant the Select or Alter privileges (or both) on a sequence:



Element	Purpose	Restrictions	Syntax
owner	Owner of the sequence	Must own *sequence*	Owner Name, p. 4-234
sequence	Sequence on which to grant privileges	Must exist	Identifier, p. 4-189
synonym	Synonym for sequence on which to grant privileges	Must exist	Identifier, p. 4-189

The sequence must reside in the current database. You can qualify the *sequence* or *synonym* identifier with a valid *owner* name, but the name of a remote *database* (or *database@server*) is not valid as a qualifier.

You can also use the WITH GRANT option when you specify ALTER, SELECT, or ALL to specify privileges that you are granting on a sequence object.

Alter Privilege

You can grant the Alter privilege on a sequence to another user or to a role. The Alter privilege enables a specified user or role to modify the definition of a sequence with the ALTER SEQUENCE statement or to rename the sequence with the RENAME SEQUENCE statement.

Select Privilege

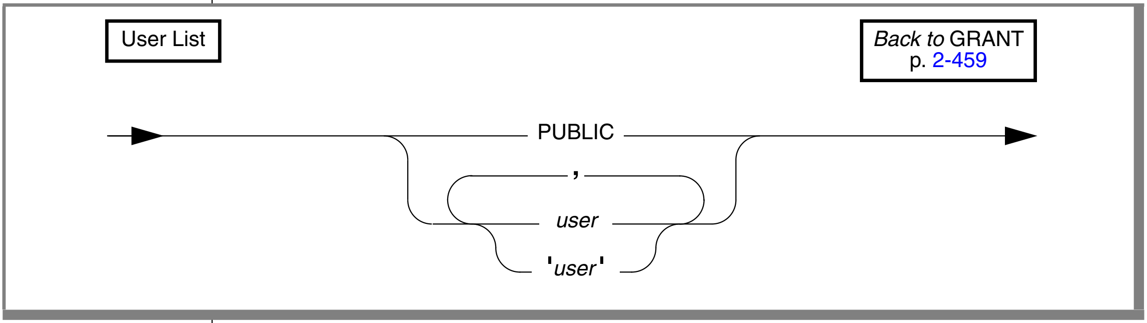
You can grant the Select privilege on a sequence to another user or to a role. The Select privilege enables a specified user or role to use *sequence*.CURRVAL and *sequence*.NEXTVAL expressions in SQL statements to read and to increment (respectively) the values of a sequence.

ALL Keyword

You can use the ALL keyword to grant both Alter and Select privileges on a sequence object to another user or to a role.

User List

You can grant privileges to an individual user or to a list of users. You can also specify the PUBLIC keyword to grant privileges to all users.



Element	Purpose	Restrictions	Syntax
user	Login name of the user who receives the role or privilege	Put quotes around *user* to preserve the lettercase of the name of the user. The single keyword PUBLIC grants a role or a privilege to all authorized users.	Owner Name, p. 4-234

The following example grants the table-level privilege Insert on **table1** to the user named **mary** in a database that is not ANSI-compliant:

```
GRANT INSERT ON table1 TO mary
```

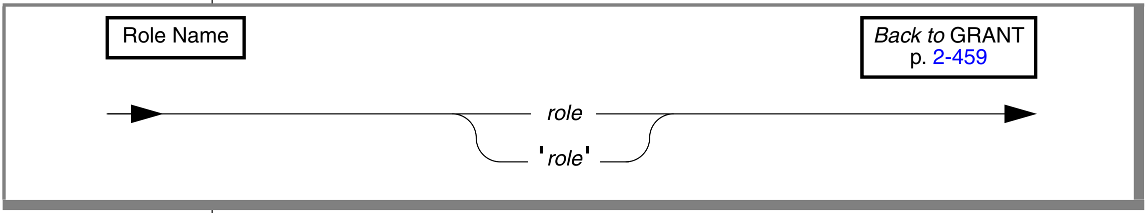
ANSI

In an ANSI-compliant database, if you do not use quotes around *user*, the name of the user is stored in uppercase letters. ♦

Role Name

You can identify one or more users by a name that describes their function, or *role*. You create the role, then grant the role to one or more users. You can also grant a role to another role.

After you create and grant a role, you can grant certain privileges to the one or more users associated with that role name.



Element	Purpose	Restrictions	Syntax
role	Name of a role that is granted, or to which a privilege or another role is granted	Must exist. If enclosed between quotation marks, *role* is case sensitive.	Identifier, p. 4-189

Granting a Role to a User or Another Role

You must add a role to the database before the role can be used in a GRANT statement. For more information, see "CREATE ROLE" on page 2-194.

A DBA has the authority to grant a new role to another user. If a user receives a role WITH GRANT OPTION, that user can grant the role to other users or to another role. Users keep a role granted to them until a REVOKE statement breaks the association between their login names and the role name.

Important: *CREATE ROLE and GRANT do not activate the role. A role has no effect until SET ROLE enables it. A role grantor or a role grantee can issue the SET ROLE.*

The following example shows the actions required to grant and activate the role **payables** to a group of employees who perform account payables functions. First the DBA creates role **payables**, then grants it to **maryf**.

```
CREATE ROLE payables;
GRANT payables TO maryf WITH GRANT OPTION
```

The DBA or **maryf** can activate the role with the following statement:

```
SET ROLE payables
```

User **maryf** has the WITH GRANT OPTION authorization to grant **payables** to other employees who pay accounts.

```
GRANT payables TO charly, gene, marvin, raoul
```

If you grant privileges for one role to another role, the recipient role has a combined set of privileges. The following example grants the role **petty_cash** to the role **payables**:

```
CREATE ROLE petty_cash;
SET ROLE petty_cash;
GRANT petty_cash TO payables
```

If you attempt to grant a role to itself, either directly or indirectly, the database server generates an error.

Granting a Privilege to a Role

You can grant table- and routine-level privileges to a role if you have the authority to grant these same privileges to login names or PUBLIC. A role cannot have database-level privileges.

IDS

You can also grant type-level privileges to a role. ♦

When you grant a privilege to a role:

- You can specify the AS *grantor* clause.

  In this way, whomever has the role can revoke these same privileges. For more information, see "AS grantor Clause" on page 2-478.

- You cannot include the WITH GRANT OPTION clause.

  A role cannot, in turn, grant the same table-, type-, or routine-level privileges to another user.

The following example grants the table-level privilege Insert on the **supplier** table to the role **payables**:

```
GRANT INSERT ON supplier TO payables
```

Anyone granted the role of **payables** can now insert into **supplier**.

WITH GRANT OPTION Keywords

The WITH GRANT OPTION keywords convey the specified privilege to *user* with the right to grant the same privileges to other users. You create a chain of privileges that begins with you and extends to *user* as well as to whomever *user* subsequently conveys the right to grant privileges. If you use WITH GRANT OPTION, you can no longer control the dissemination of privileges.

If you revoke from *user* the privilege that you granted using the WITH GRANT OPTION keyword, you sever the chain of privileges. That is, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created (unless *user*, or the users who received privileges from *user*, were granted the same set of privileges by someone else).

The following examples illustrate this situation. You, as the owner of the table **items**, issue the following statements to grant access to user **mary**:

```
REVOKE ALL ON items FROM PUBLIC;
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION
```

User **mary** uses her privilege to grant users **cathy** and **paul** access to the table.

```
GRANT SELECT, UPDATE ON items TO cathy;
GRANT SELECT ON items TO paul
```

Later you issue the following statement to cancel access privileges for user **mary** on the **items** table:

```
REVOKE SELECT, UPDATE ON items FROM mary
```

This single statement effectively revokes all privileges on the **items** table from users **mary**, **cathy**, and **paul**. If you want to create a chain of privileges with another user as the source of the privilege, use the AS *grantor* clause.

AS *grantor* Clause

When you grant privileges, by default, you are the one who can revoke those privileges. The AS *grantor* clause lets you establish another user as the source of the privileges you are granting. When you use this clause, the login provided in the AS grantor clause replaces your login in the appropriate system catalog table.

You can use this clause only if you have the DBA privilege on the database.

After you use this clause, only the specified *grantor* can REVOKE the effects of the current GRANT. Even a DBA cannot revoke a privilege unless that DBA is listed in the system catalog table as the source who granted the privilege.

The following example illustrates this situation. You are the DBA and you grant all privileges on the **items** table to user **tom** with the right to grant all privileges:

```
REVOKE ALL ON items FROM PUBLIC;
GRANT ALL ON items TO tom WITH GRANT OPTION
```

The following example illustrates a different situation. You also grant Select and Update privileges to user **jim**, but you specify that the grant is made as user **tom**. (The records of the database server show that user **tom** is the grantor of the grant in the **systabauth** system catalog table, rather than you.)

```
GRANT SELECT, UPDATE ON items TO jim AS tom
```

Later, you decide to revoke privileges on the **items** table from user **tom**, so you issue the following statement:

```
REVOKE ALL ON items FROM tom
```

When you try to revoke privileges from user **jim** with a similar statement, however, the database server returns an error, as the next example shows:

```
REVOKE SELECT, UPDATE ON items FROM jim

580: Cannot revoke permission.
```

You get an error because the database server record shows the original grantor as user **tom**, and you cannot revoke the privilege. Although you are the DBA, you cannot revoke a privilege that another user granted.

Related Information

Related statements: GRANT FRAGMENT, REVOKE, and REVOKE FRAGMENT

For information about roles, see the following statements: CREATE ROLE, DROP ROLE, and SET ROLE.

In the *IBM Informix Database Design and Implementation Guide*, see the discussion of privileges.

For a discussion of how to embed GRANT and REVOKE statements in programs, see the *IBM Informix Guide to SQL: Tutorial*.

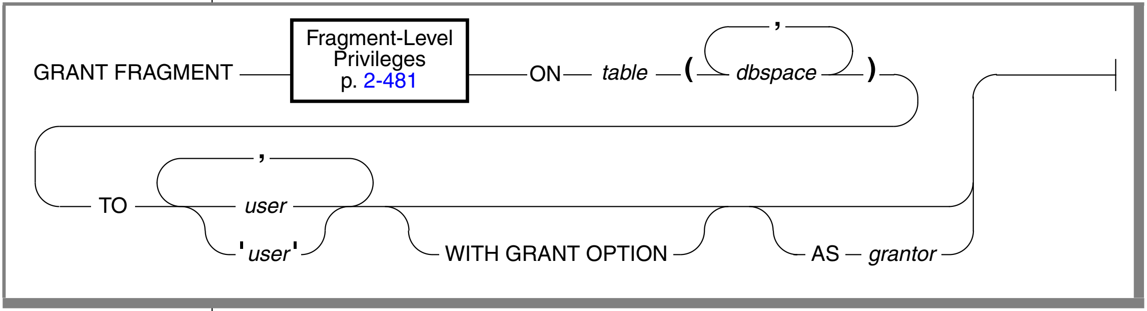
+

IDS

GRANT FRAGMENT

Use the GRANT FRAGMENT statement to grant Insert, Update, and Delete privileges to users on individual fragments of a fragmented table that has been fragmented by expression.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	dbspace that stores the fragment(s) on which privileges are to be granted	Must exist. You must specify at least one.	Identifier, p. 4-189
grantor	User who can revoke the privileges	Same as for *user*..	Same as for *user*.
table	Table that contains fragment(s) on which privileges are granted	Must exist and must be fragmented by expression.	Database Object Name, p. 4-46
user	User(s) to whom the specified privileges are to be granted	Must be a valid authorization identifier.	Must conform to the rules of your operating system.

Usage

The GRANT FRAGMENT statement is similar to the GRANT statement. Both statements grant privileges to users, but GRANT ASSIGNS privileges on a table, but GRANT FRAGMENT assigns privileges on table fragments. GRANT FRAGMENT is valid only for tables that are fragmented according to an expression-based distribution scheme. For an explanation of expression-based fragmentation, see "Expression Distribution Scheme" on page 2-25.

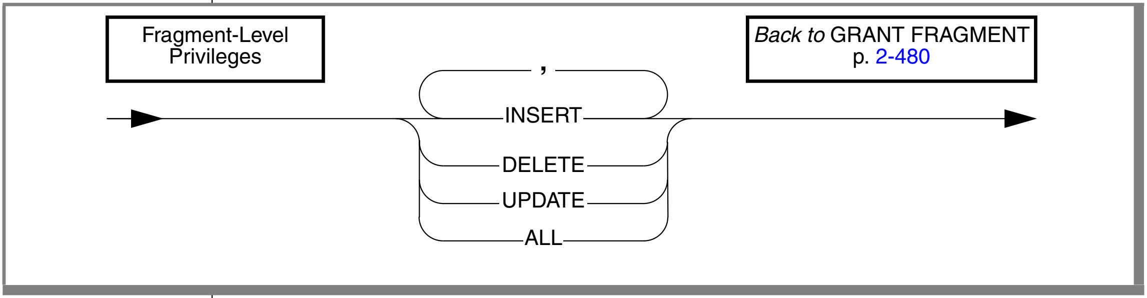
In the TO clause, if you enclose *user* in quotation marks, the name of the user designated as grantee is case sensitive and is stored exactly as you typed it.

ANSI

In an ANSI-compliant database, if you do not use quotes around *user*, the name of the user is stored in uppercase letters. ♦

The user specified in the AS clause is listed as the *grantor* of the privileges in the **grantor** column of the **sysfragauth** system catalog table. If the AS clause is omitted, the user who issues the GRANT FRAGMENT statement is the default *grantor* of the privileges.

Fragment-Level Privileges



The following table defines each of the fragment-level privileges.

Privilege	Effect
ALL	Provides Insert, Delete, and Update privileges on a fragment
INSERT	Can insert rows in the fragment
DELETE	Can delete rows in the fragment
UPDATE	Can update rows in the fragment and in any column in UPDATE actions.

Definition of Fragment-Level Authorization

ANSI

In an ANSI-compliant database, the owner implicitly receives all table-level privileges on a newly created table, but no other users receive privileges. ♦

A user who has table privileges on a fragmented table has the privileges implicitly on all fragments of the table. These privileges are not recorded in the **sysfragauth** system catalog table.

When a fragmented table is created in a database that is not ANSI compliant, the table owner implicitly receives all table-level privileges on the table, and other users (that is, PUBLIC) receive the following default set of privileges on the table: Select, Update, Insert, Delete, and Index. The privileges granted to PUBLIC are explicitly recorded in the **systabauth** system catalog table.

Whether or not the database is ANSI compliant, you can use the GRANT FRAGMENT statement to grant explicit Insert, Update, and Delete privileges on one or more fragments of a table that is fragmented by expression. The privileges that the GRANT FRAGMENT statement grants are explicitly recorded in the **sysfragauth** system catalog table.

The Insert, Update, and Delete privileges that are conferred on table fragments by the GRANT FRAGMENT statement are collectively known as fragment-level privileges or fragment-level authority.

Role of Fragment-Level Authority in Command Validation

Fragment-level privilege enables users to execute INSERT, DELETE, and UPDATE statements on table fragments even if they lack Insert, Update, and Delete privileges on the table as a whole. Users who lack privileges at the table level can insert, delete, and update rows in authorized fragments because of the algorithm by which the database server validates commands. This algorithm consists of the following checks:

1. When a user executes an INSERT, DELETE, or UPDATE statement, the database server first checks whether the user has the table privileges necessary for the operation attempted. If the table privileges exist, the statement continues processing.
2. If the table privileges do not exist, the database server checks whether the table is fragmented by expression. If the table is not fragmented by expression, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the command.
3. If the table is fragmented by expression, the database server checks whether the user has the fragment privileges necessary for the operation attempted. If the fragment privileges exists, the command continues processing. If the fragment privileges do not exist, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the command.

Duration of Fragment-Level Privileges

The duration of fragment-level privileges is tied to the duration of the fragmentation strategy for the table as a whole.

If you drop a fragmentation strategy by means of a DROP TABLE statement or the INIT, DROP, or DETACH clauses of an ALTER FRAGMENT statement, you also drop any privileges that exist for the affected fragments. Similarly, if you drop a dbspace, you also drop any privileges that exist for the fragment that resides in that dbspace.

Tables that are created as a result of a DETACH or INIT clause of an ALTER FRAGMENT statement do not keep the privileges that the former fragment or fragments had when they were part of the fragmented table. Instead, such tables assume the default table privileges.

If a table with fragment privileges defined on it is changed to a table with a round-robin strategy or some other expression strategy, the fragment privileges are also dropped, and the table assumes the default table privileges.

You can grant fragment-level privileges on one fragment of a table or on a list of fragments.

Granting Privileges on One Fragment

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp1** to user **larry**:

```
GRANT FRAGMENT ALL ON customer (dbsp1) TO larry
```

Granting Privileges on More Than One Fragment

The following statement grants the Insert, Update, and Delete privileges on the fragments of the **customer** table in **dbsp1** and **dbsp2** to user **millie**:

```
GRANT FRAGMENT ALL ON customer (dbsp1, dbsp2) TO millie
```

Granting Privileges on All Fragments of a Table

To grant privileges on all fragments of a table to the same user or users, you can use the GRANT statement instead of the GRANT FRAGMENT statement. You can also use the GRANT FRAGMENT statement for this purpose.

Assume that the **customer** table is fragmented by expression into three fragments, and these fragments reside in the dbspaces named **dbsp1**, **dbsp2**, and **dbsp3**. You can use either of the following statements to grant the Insert privilege on all fragments of the table to user **helen**:

```
GRANT FRAGMENT INSERT ON customer (dbsp1, dbsp2, dbsp3)
TO helen;

GRANT INSERT ON customer TO helen;
```

Granting Privileges to One User or a List of Users

You can grant fragment-level privileges to a single user or to a list of users.

Granting Privileges to One User

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp3** to user **oswald**:

```
GRANT FRAGMENT ALL ON customer (dbsp3) TO oswald
```

Granting Privileges to a List of Users

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp3** to users **jerome** and **hilda**:

```
GRANT FRAGMENT ALL ON customer (dbsp3) TO jerome, hilda
```

Granting One Privilege or a List of Privileges

When you specify fragment-level privileges in a GRANT FRAGMENT statement, you can specify one privilege, a list of privileges, or all privileges.

Granting One Privilege

The following statement grants the Update privilege on the fragment of the **customer** table in **dbsp1** to user **ed**:

```
GRANT FRAGMENT UPDATE ON customer (dbsp1) TO ed
```

Granting a List of Privileges

The following statement grants the Update and Insert privileges on the fragment of the **customer** table in **dbsp1** to user **susan**:

```
GRANT FRAGMENT UPDATE, INSERT ON customer (dbsp1) TO susan
```

Granting All Privileges

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp1** to user **harry**:

```
GRANT FRAGMENT ALL ON customer (dbsp1) TO harry
```

WITH GRANT OPTION Clause

By including the WITH GRANT OPTION clause in the GRANT FRAGMENT statement, you convey the specified fragment-level privileges to a user and the right to grant those same privileges to other users.

The following statement grants the Update privilege on the fragment of the **customer** table in **dbsp3** to user **george** and gives this user the right to grant the Update privilege on the same fragment to other users:

```
GRANT FRAGMENT UPDATE ON customer (dbsp3) TO george
   WITH GRANT OPTION
```

AS grantor Clause

The AS *grantor* clause is optional in a GRANT FRAGMENT statement. Use this clause to specify the grantor of the privilege.

When you include the AS *grantor* clause in the GRANT FRAGMENT statement, you specify that the user who is named as *grantor* is listed as the grantor of the privilege in the **grantor** column of the **sysfragauth** system catalog table.

In the next example, the DBA grants the Delete privilege on the fragment of the **customer** table in **dbsp3** to user **martha**. In the GRANT FRAGMENT statement, the DBA uses the AS *grantor* clause to specify that user **jack** is listed as the grantor of the privilege in the **sysfragauth** system catalog table.

```
GRANT FRAGMENT DELETE ON customer (dbsp3) TO martha AS jack
```

Omitting the AS grantor Clause

When GRANT FRAGMENT does not include the AS *grantor* clause, the user who issues the statement is the default grantor of the specified privileges.

In the next example, the user grants the Update privilege on the fragment of the **customer** table in **dbsp3** to user **fred**. Because this statement does not specify the AS *grantor* clause, the user who issues the statement is listed by default as the grantor of the privilege in the **sysfragauth** system catalog table.

```
GRANT FRAGMENT UPDATE ON customer (dbsp3) TO fred
```

If you omit the AS *grantor* clause of GRANT FRAGMENT, or if you specify your own login name as the *grantor*, you can later use the REVOKE FRAGMENT statement to revoke the privilege that you granted to the specified user.

XPS

In Extended Parallel Server, however, if you specify someone other than yourself as the grantor of the specified privilege to the specified user, only that grantor can revoke the privilege from the user. ♦

For example, if you grant the Delete privilege on the fragment of the **customer** table in **dbsp3** to user **martha** but specify user **jack** as the grantor of the privilege, user **jack** can revoke that privilege from user **martha**, but you cannot revoke that privilege from user **martha**.

For Dynamic Server, the DBA, or the owner of the fragment, can use the AS clause of GRANT FRAGMENT to revoke privileges on the fragment.

Related Information

Related statements: GRANT and REVOKE FRAGMENT

For a discussion of fragment-level and table-level privileges, see the *IBM Informix Database Design and Implementation Guide*.

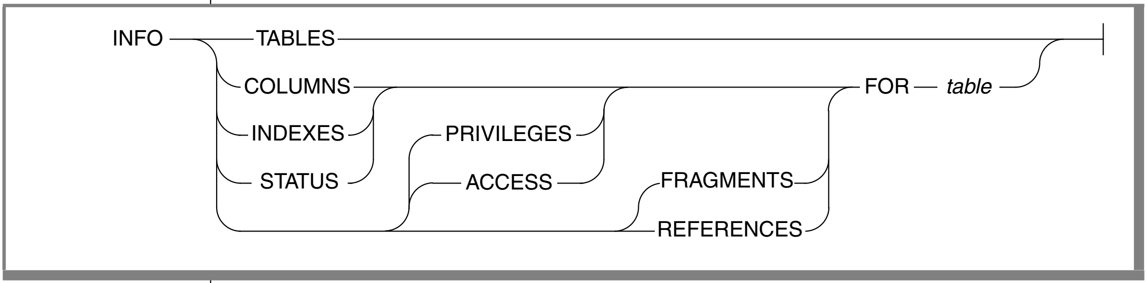
+

DB

INFO

Use the INFO statement to display information about databases tables. Use this statement with DB-Access.

Syntax



Element	Purpose	Restrictions	Syntax
table	Table about which you seek information	Must exist.	Database Object Name, p. 4-46

Usage

Keywords of the INFO statement can display the following information.

INFO Keyword	Information Displayed
TABLES	Table names in the current database
COLUMNS	Column information for a specified table
INDEXES	Index information for a specified table
FRAGMENTS	Fragmentation strategy for a specified table
ACCESS *or* **PRIVILEGES**	Access privileges for a specified table. (The ACCESS and PRIVILEGES keywords are synonyms.)
REFERENCES	References privileges for columns of a specified table
STATUS	Status information for a specified table

- **TABLES Keyword**

  Use TABLES to display a list of the tables in the current database, not including system catalog table s, in one of the following formats:

  - If you are the owner of the **cust_calls** table, it appears as **cust_calls**.
  - If you are *not* the owner of the **cust_calls** table, the name of the owner precedes the table name, such as **'june'.cust_calls**.

- **COLUMNS Keyword**

  Use COLUMNS to display the names and data types of the columns in a specified table and whether NULL values are allowed.

- **INDEXES Keyword**

  Use INDEXES to display the name, owner, and type of each index in a specified table, the clustered status, and the indexed columns.

- **FRAGMENTS Keyword**

  Use FRAGMENTS to display the names of dbspaces storing fragments of a table. If the table is fragmented with an expression-based distribution scheme, the INFO statement also shows the expressions.

- **ACCESS or PRIVILEGES Keyword**

  Use ACCESS or PRIVILEGES to display user-access privileges for a specified table. (These two keywords are synonyms in this context.)

- **REFERENCES Keyword**

  Use REFERENCES to display the References privilege for users for the columns of a specified table. For database-level privileges, use a SELECT statement to query the **sysusers** system catalog table.

- **STATUS Keyword**

  Use STATUS to display information about the owner, row length, number of rows and columns, creation date, and status of audit trails for a specified table.

Related Information

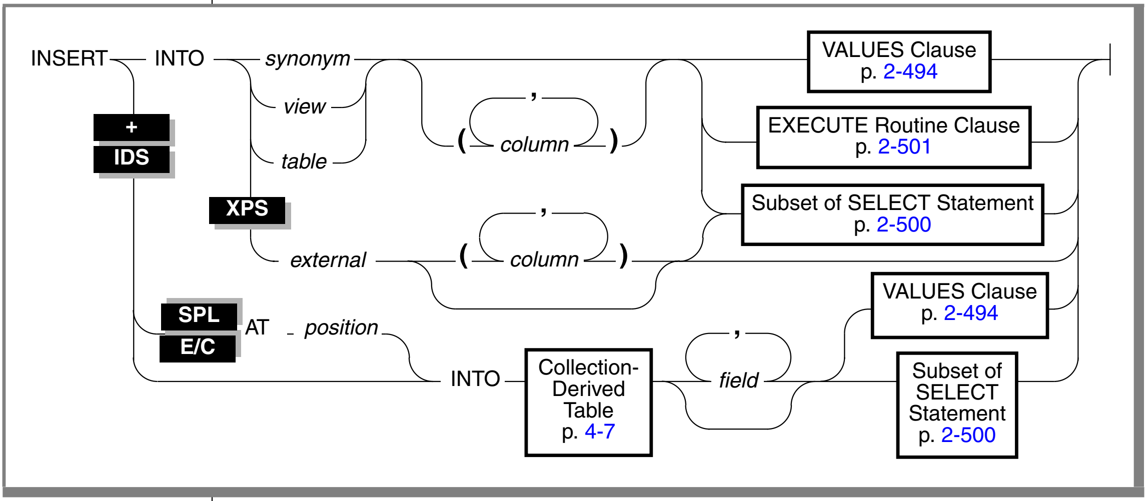
Related statements: GRANT and REVOKE

For a description of the **Info** option on the SQL menu or the TABLE menu in DB-Access, see the *IBM Informix DB-Access User's Guide*.

INSERT

Use the INSERT statement to insert one or more new rows into a table or view or one or more elements into an SPL or ESQL/C collection variable.

Syntax


INSERT
INTO
synonym
view
table
+
IDS
XPS
external
(
column
)
,
SPL
E/C
AT
position
INTO
Collection-Derived Table p. 4-7
field
VALUES Clause p. 2-494
EXECUTE Routine Clause p. 2-501
Subset of SELECT Statement p. 2-500
VALUES Clause p. 2-494
Subset of SELECT Statement p. 2-500

Element	Purpose	Restrictions	Syntax
column	Column to receive new value	See "Specifying Columns" on page 2-490.	Identifier, p. 4-189
external	External table into which to insert data	Must exist.	Database Object Name, p. 4-46
field	Field of a named or unnamed row type	Must already be defined in the database.	"Field Definition" on page 2-201
position	Position at which to insert an element of a LIST data type	Literal number or an INT or SMALLINT type SPL variable.	Literal Number, p. 4-216
synonym, table, view	Table, view, or synonym in which to insert data	Synonym and the table to which it points must exist.	Database Object Name, p. 4-46

Usage

To insert data into a table, you must either own the table or have the Insert privilege for the table (see "GRANT" on page 2-459). To insert data into a view, you must have the required Insert privilege, and the view must meet the requirements explained in "Inserting Rows Through a View" on page 2-491.

If you insert data into a table that has data integrity constraints associated with it, the inserted data must meet the constraint criteria. If it does not, the database server returns an error.

If you are using effective checking, and the checking mode is set to IMMEDIATE, all specified constraints are checked at the end of each INSERT statement. If the checking mode is set to DEFERRED, all specified constraints are *not* checked until the transaction is committed.

Specifying Columns

If you do not explicitly specify one or more columns, data is inserted into columns using the column order that was established when the table was created or last altered. The column order is listed in the **syscolumns** system catalog table.

E/C

In ESQL/C, you can use the DESCRIBE statement with an INSERT statement to determine the column order and the data type of the columns in a table. ♦

The number of columns specified in the INSERT INTO clause must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. If you specify a column list, the columns receive data in the order in which you list the columns. The first value following the VALUES keyword is inserted into the first column listed, the second value is inserted into the second column listed, and so on.

If you omit a column from the column list, and the column does not have a default value associated with it, the database server places a NULL value in the column when the INSERT statement is executed.

E/C

IDS

SPL

Using the AT Clause

Use the AT clause to insert LIST elements at a specified position in a collection variable. By default, Dynamic Server adds a new element at the end of a LIST collection.

If you specify a position greater than the number of elements in the list, the database server adds the element to the end of the list. You must specify a position value of at least 1 because the first element in the list is at position 1.

The following SPL example inserts a value at a specific position in a list:

```
CREATE PROCEDURE test3()
   DEFINE a_list LIST(SMALLINT NOT NULL);
   SELECT list_col INTO a_list FROM table1 WHERE id = 201;
   INSERT AT 3 INTO TABLE(a_list) VALUES( 9 );
   UPDATE table1 VALUES list_col = a_list WHERE id = 201;
END PROCEDURE;
```

Suppose that before this INSERT, **a_list** contained the elements {1,8,4,5,2}. After this INSERT, **a_list** contains the elements {1,8,9,4,5,2}. The new element 9 was inserted at position 3 in the list. For more information on inserting values into collection variables, see "Collection-Derived Table" on page 4-7.

Inserting Rows Through a View

You can insert data through a *single-table* view if you have the Insert privilege on the view. To do this, the defining SELECT statement can select from only one table, and it cannot contain any of the following components:

- DISTINCT keyword
- GROUP BY clause
- Derived value (also referred to as a virtual column)
- Aggregate value

Columns in the underlying table that are unspecified in the view receive either a default value or a NULL value if no default is specified. If one of these columns does not specify a default value, and a NULL value is not allowed, the insert fails.

You can use data-integrity constraints to prevent users from inserting values into the underlying table that do not fit the view-defining SELECT statement.

For further information, see "WITH CHECK OPTION Keywords" on page 2-314.

If several users are entering sensitive information into a single table, the built-in USER function can limit their view to only the specific rows that each user inserted. The following example contains a view and an INSERT statement that achieve this effect:

```
CREATE VIEW salary_view AS
   SELECT lname, fname, current_salary
      FROM salary
      WHERE  entered_by = USER

INSERT INTO salary
   VALUES ('Smith', 'Pat', 75000, USER)
```

E/C

Inserting Rows with a Cursor

In ESQL/C, if you associate a cursor with an INSERT statement, you must use the OPEN, PUT, and CLOSE statements to carry out the INSERT operation. For databases that have transactions but are not ANSI-compliant, you must issue these statements within a transaction.

If you are using a cursor that is associated with an INSERT statement, the rows are buffered before they are written to the disk. The insert buffer is flushed under the following conditions:

- The buffer becomes full.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- In a database that is not ANSI-compliant, an OPEN statement implicitly closes and then reopens the cursor.
- A COMMIT WORK statement ends the transaction.

When the insert buffer is flushed, the client processor performs appropriate data conversion before it sends the rows to the database server. When the database server receives the buffer, it converts any user-defined data types and then begins to insert the rows one at a time into the database. If an error is encountered while the database server inserts the buffered rows into the database, any buffered rows that follow the last successfully inserted rows are discarded.

Inserting Rows into a Database Without Transactions

If you are inserting rows into a database with no transaction logging, you must take explicit action to restore inserted rows if the operation fails. For example, if INSERT fails after entering some rows, the successfully inserted rows remain in the table. You cannot recover automatically from a failed INSERT into a database for which no transaction log exists

Inserting Rows into a Database with Transactions

If you are inserting rows into a database with transactions, and you are using explicit transactions, use the ROLLBACK WORK statement to undo the insertion. If you do not execute BEGIN WORK before the insert, and the insert fails, the database server automatically rolls back any database modifications made since the beginning of the insert.

ANSI

If you are inserting rows into an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an INSERT statement fails, use the ROLLBACK WORK statement to undo the insertions.

If you are using an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

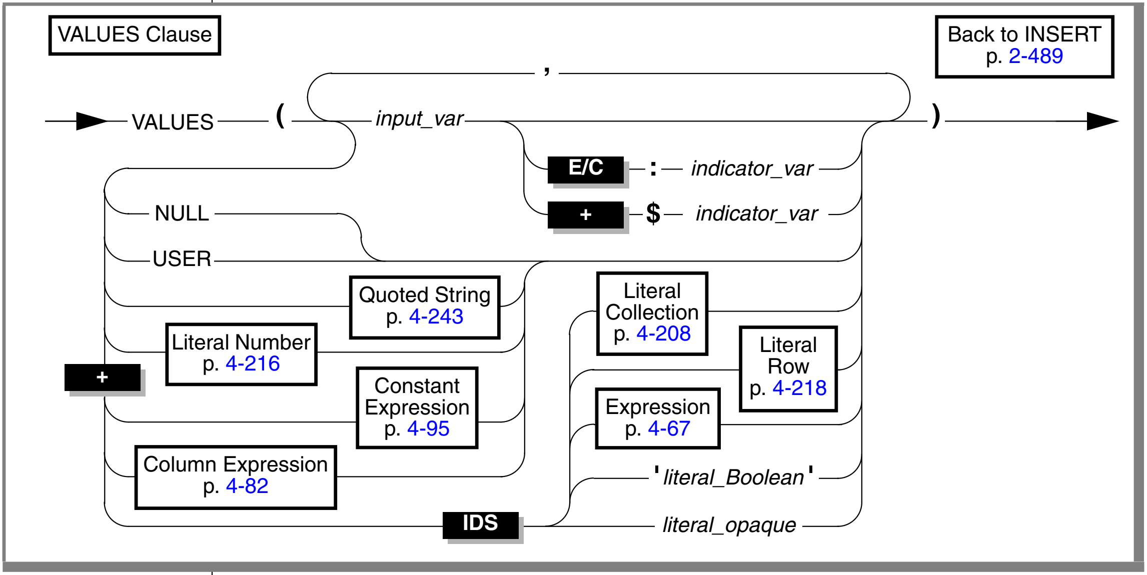
Tables that you create with the RAW logging type are never logged. Thus, raw tables are not recoverable, even if the database uses logging. For information about raw tables, refer to the *IBM Informix Guide to SQL: Reference*.

Rows that you insert with a transaction remain locked until the end of the transaction. The end of a transaction is either a COMMIT WORK statement, where all modifications are made to the database, or a ROLLBACK WORK statement, where none of the modifications are made to the database. If many rows are affected by a *single* INSERT statement, you can exceed the maximum number of simultaneous locks permitted. To prevent this situation, either insert fewer rows per transaction or lock the page (or the entire table) before you execute the INSERT statement.

VALUES Clause

The VALUES clause can specify values to insert into one or more columns. When you use the VALUES clause, you can insert only one row at a time.

Each value that follows the VALUES keyword is assigned to the corresponding column listed in the INSERT INTO clause (or in column order, if a list of columns is not specified). If you are inserting a quoted string into a column, the maximum length that can be inserted without error is 256 bytes.



Element	Purpose	Restrictions	Syntax
indicator_var	Variable to show if SQL statement returns NULL to *input_var*	See the *IBM Informix ESQL/C Programmer's Manual*.	Language specific
input_var	Variable that holds value to insert. Can be a COLLECTION variable.	Can contain any value option of VALUES clause.	Language specific
literal_opaque	Literal representation for an opaque data type	Must be recognized by the input support function of opaque type.	See documentation for the opaque type.
literal_Boolean	Literal representation of a Boolean value as a single character	Either `'t'` (TRUE) or `'f'` (FALSE)	Quoted String, p. 4-243

E/C

In ESQL/C, if you use an *input_var* variable to specify the value, you can insert character strings longer than 256 bytes into a table. ♦

For the keywords and the types of literal values that are valid in the VALUES clause, refer to "Constant Expressions" on page 4-95.

Considering Data Types

The value that the INSERT statement puts into a column does not need to be of the same data type as the column that receives it. These two data types, however, must be compatible. Two data types are *compatible* if the database server has some way to cast one data type to another. A *cast* is the mechanism by which the database server converts one data type to another.

The database server makes every effort to perform data conversion. If the data cannot be converted, the INSERT operation fails. Data conversion also fails if the target data type cannot hold the value that is specified. For example, you cannot insert the integer `123456` into a column defined as a SMALLINT data type because this data type cannot hold a number that large.

For a summary of the casting that the database server provides, see the *IBM Informix Guide to SQL: Reference*. For information on how to create a user-defined cast, see the CREATE CAST statement in this manual and *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Inserting Values into Serial Columns

You can insert consecutive numbers, explicit values, or explicit values that reset the serial sequence value in a serial column:

- To insert a consecutive serial value. Specify a zero (0) for the serial column in the INSERT statement. In this case, the database server assigns the next highest value.
- To insert an explicit value. Specify the nonzero value after first verifying that it does not duplicate one already in the table. If the serial column is uniquely indexed or has a unique constraint, and you r value duplicates one already in the table, an error results. If the value is greater than the current maximum value, you will create a gap in the series.

- To create a gap in the series (reset the serial value). Specify a positive value that is greater than the current maximum value in the column.

  Alternatively, you can use the MODIFY clause of the ALTER TABLE statement to reset the next value of a serial column.

For more information, see "Altering the Next Serial Number" on page 2-67.

NULL values are not allowed in serial columns.

IDS

Inserting a serial value into a table that is part of a table hierarchy updates all tables in the hierarchy that contain the serial counter with the value that you insert (either a zero (0) for the next highest value or a specific number). ♦

IDS

Inserting Values into Opaque-Type Columns

Some opaque data types require special processing when they are inserted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This is accomplished by calling a user-defined support function called **assign( )**. When you execute INSERT on a table whose rows contains one of these opaque types, the database server automatically invokes the **assign( )** function for the type. The **assign( )** function can make the decision of how to store the data. For more information about the **assign( )** support function, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

IDS

Inserting Values into Collection Columns

You can use the VALUES clause to insert values into a collection column. For more information, see "Collection Constructors" on page 4-108.

For example, suppose you define the **tab1** table as follows:

```
CREATE TABLE tab1
(
   int1 INTEGER,
   list1 LIST(ROW(a INTEGER, b CHAR(5)) NOT NULL),
   dec1 DECIMAL(5,2)
)
```

The following INSERT statement inserts a row into **tab1**:

```
INSERT INTO tab1 VALUES
   (
   10,
   LIST{ROW(1,'abcde'),
      ROW(POW(3,3), '=27'),
      ROW(ROUND(ROOT(126)), '=11')},
   100
   )
```

The collection column, **list1**, in this example has three elements. Each element is an unnamed row type with an INTEGER field and a CHAR(5) field. The first element is composed of two literal values, an integer (1) and a quoted string (`abcde`). The second and third elements also use a quoted string to indicate the second field, but specify the value for the first field with an expression.

Regardless of what method you use to insert values into a collection column, you cannot insert NULL elements into the column. Thus expressions that you use cannot evaluate to NULL. If the collection that you are attempting to insert contains a NULL element, the database server returns an error.

E/C

SPL

You can also use a collection variable to insert the values of one or more collection elements into a collection column. For more information, see "Collection-Derived Table" on page 4-7. ♦

IDS

Inserting Values into ROW-Type Columns

The VALUES clause to insert literal and nonliteral values in a named or unnamed ROW type column, as in the following example:

```
CREATE ROW TYPE address_t
(
   street CHAR(20),
   city CHAR(15),
   state CHAR(2),
   zipcode CHAR(9)
);
CREATE TABLE employee
(
   name ROW ( fname CHAR(20), lname CHAR(20)),
   address address_t
);
```

The next example inserts literal values in the **name** and **address** columns:

```
INSERT INTO employee VALUES
    (
        ROW('John', 'Williams'),
        ROW('103 Baker St', 'Tracy','CA', 94060)::address_t
    )
```

INSERT uses ROW constructors to generate values for the **name** column (an unnamed ROW data type) and the **address** column (a named ROW data type). When you specify a value for a named ROW data type, you must use the CAST AS keywords or the double colon (**::**) operator, with the name of the ROW data type, to cast the value to the named ROW data type.

For the syntax for ROW constructors, see "Constructor Expressions" on page 4-106 in the Expression segment. For information on literal values for named r and unnamed ROW data types, see "Literal Row" on page 4-218.

E/C

You can use ESQL/C host variables to insert *nonliteral* values in two ways:

- An entire ROW type into a column. Use a **row** variable in the VALUES clause to insert values for all fields in a ROW column at one time.
- Individual fields of a ROW type. To insert nonliteral values in a ROW-type column, insert the elements into a **row** variable and then specify the **collection** variable in the SET clause of an UPDATE statement.

When you use a ROW variable in the VALUES clause, the ROW variable must contain values for each field value. For information on how to insert values in a ROW variable, see "Inserting into a Row Variable" on page 2-502. ♦

IDS

Using Expressions in the VALUES Clause

With IBM Informix Dynamic Server, you can insert any type of expression except a column expression into a column. For example, you can insert built-in functions that return the current date, date and time, login name of the current user, or database server name where the current database resides.

The TODAY keyword returns the system date. The CURRENT keyword returns the system date and time. The USER keyword returns a string that contains the login account name of the current user. The SITENAME or DBSERVERNAME keyword returns the database server name where the current database resides. This example uses built-in functions to insert data:

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
                  call_code, call_descr)
   VALUES (212, CURRENT, USER, 'L', '2 days')
```

For more information, see "Expression" on page 4-67.

Inserting NULL Values

When you execute the INSERT statement, it inserts a NULL value i into any column for which you provide no value, as well as for all columns that have no associated default values and that are not listed explicitly.

You also can specify the NULL keyword in the VALUES clause to indicate that a column should be assigned a NULL value.

The following example inserts values into three columns of the **orders** table:

```
INSERT INTO orders (orders_num, order_date, customer_num)
   VALUES (0, NULL, 123)
```

In this example, a NULL value is explicitly entered in the **order_date** column, and all other columns of the **orders** table that are *not* explicitly listed in the INSERT INTO clause are also filled with NULL values.

Truncated CHAR Values

If you assign a value to a CHAR(*n*) column or variable and the length of that value exceeds *n* characters, the database server truncates the last characters without raising an error. For example, suppose that you define this table:

```
CREATE TABLE tab1 (col_one CHAR(2)
```

The database server truncates the data values in the following INSERT statements to `"jo"` and `"sa"` respectively, but does not return a warning:

```
INSERT INTO tab1 VALUES ("john");
INSERT INTO tab1 VALUES ("sally");
```

Thus the semantic integrity of data for a CHAR(*n*) column or variable is not enforced when the value inserted or updated exceeds length *n*.

Subset of SELECT Statement

As indicated in the diagram for "INSERT" on page 2-489, not all clauses and options of the SELECT statement are available for you to use in an INSERT statement. The following SELECT clauses and options are not supported:

- FIRST and INTO TEMP
- IDS ORDER BY and UNION ♦

ANSI

If this statement has a WHERE clause that does not return rows, **sqlca** returns SQLNOTFOUND (100) for ANSI-compliant databases. ♦

If an insert that is part of a multistatement prepare inserts no rows, **sqlca** returns SQLNOTFOUND (100) for both ANSI-compliant databases and databases that are not ANSI-compliant. In databases that are not ANSI-compliant, **sqlca** returns zero (0) if no rows satisfy the WHERE clause.

IDS

If you are inserting values into a supertable in a table hierarchy, the subquery can reference a subtable. If you are inserting values into a subtable in a table hierarchy, the subquery can reference the supertable if it references only the supertable. That is, the subquery must use the SELECT…FROM ONLY (*supertable*)…syntax. ♦

XPS

Using External Tables

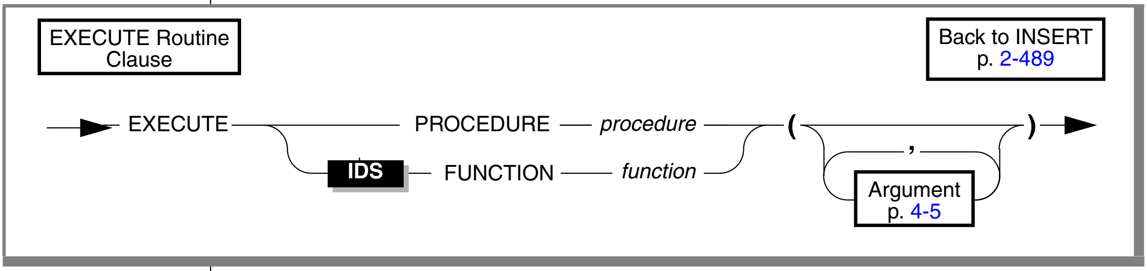
In Extended Parallel Server, a SELECT statement that is a part of a load or unload operation involving an external table is subject to these restrictions:

- Only one external table is allowed in the FROM clause.
- The SELECT subquery cannot contain an INTO clause, but it can include any valid SQL expression.

When you move data from a database into an external table, the SELECT statement must define all columns in the external table. The SELECT statement must not contain a FIRST, FOR UPDATE, INTO, INTO SCRATCH, or INTO TEMP clause. You can use an ORDER BY clause, however, to produce files that are ordered within themselves.

EXECUTE Routine Clause

You can specify the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement to insert values that a user-defined function returns.



Element	Purpose	Restrictions	Syntax
function, procedure	User-defined function or procedure to insert the data	Must exist.	Database Object Name, p. 4-46

When you use a user-defined function to insert column values, the return values of the function must have a one-to-one correspondence with the listed columns. That is, each value that the function returns must be of the data type expected by the corresponding column in the column list.

IDS

For backward compatibility, you can use the EXECUTE PROCEDURE keywords to execute an SPL function that was created with the CREATE PROCEDURE statement. ♦

If the called SPL routine scans or updates the target table of the insert, the database returns an error. That is, the SPL routine cannot select data from the table into which you are inserting rows.

If a called SPL routine contains certain SQL statements, the database server returns an error. For information on which SQL statements cannot be used in an SPL routine that is called within a data manipulation statement, see "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279.

Number of Allowed Return Values

SPL

An SPL function can return one or more values. Make sure that the number of values that the function returns matches the number of columns in the table or the number of columns that you list in the column list of the INSERT statement. The columns into which you insert the values must have compatible data types with the values that the SPL function returns. ♦

Ext

An external function can only return *one* value. Make sure that you specify only one column in the column list of the INSERT statement. This column must have a compatible data type with the value that the external function returns.The external function can be an iterator function. ♦

The following example shows how to insert data into a temporary table called **result_tmp** in order to output to a file the results of a user-defined function (**f_one**) that returns multiple rows.

```
CREATE TEMP TABLE result_tmp( ... );
INSERT INTO result_tmp EXECUTE FUNCTION f_one();
UNLOAD TO 'file' SELECT * FROM foo_tmp;
```

E/C
IDS
SPL

Inserting into a Row Variable

The INSERT statement does not support a row variable in the Collection-Derived-Table segment. You can use the UPDATE statement, however, to insert new field values into a row variable. For example, the following ESQL/C code fragment inserts a new row into the **rectangles** table (which "Inserting Values into ROW-Type Columns" on page 2-497 defines):

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
EXEC SQL END DECLARE SECTION;

...
EXEC SQL update table(:myrect)
   set x=7, y=3, length=6, width=2;
EXEC SQL insert into rectangles values (12, :myrect);
```

For more information, see "Updating a Row Variable" on page 2-775.

E/C

Using INSERT as a Dynamic Management Statement

In ESQL/C, you can use the INSERT statement to handle situations where you need to write code that can insert data whose structure is unknown at the time you compile. For more information, refer to the dynamic management section of the *IBM Informix ESQL/C Programmer's Manual*.

Related Information

Related statements: CLOSE, CREATE EXTERNAL TABLE, DECLARE, DESCRIBE, EXECUTE, FLUSH, FOREACH, OPEN, PREPARE, PUT, and SELECT

For a task-oriented discussion of inserting data into tables and for information on how to access row and collections with SPL variables, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion of the GLS aspects of the INSERT statement, see the *IBM Informix GLS User's Guide*.

For information on how to access row and collections with ESQL/C host variables, see the chapter on complex types in the *IBM Informix ESQL/C Programmer's Manual*.

+

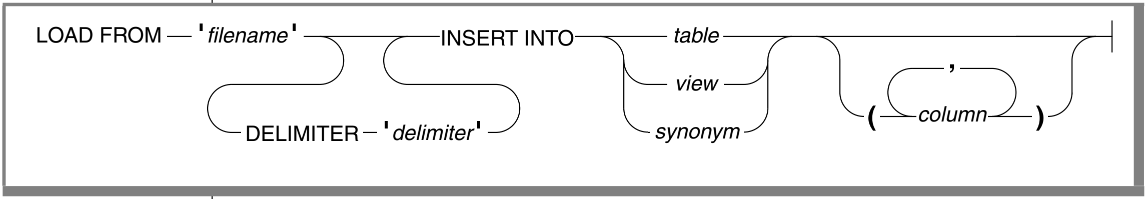
DB

SQLE

LOAD

Use the LOAD statement to insert data from an operating-system file into an existing table or view. Use this statement with DB-Access and the SQL Editor.

Syntax



Element	Purpose	Restrictions	Syntax
column	Column to receive data values from *filename*	See "INSERT INTO Clause" on page 2-511.	Identifier, p. 4-189
delimiter	Character to separate data values in each line of the load file. Default *delimiter* is the pipe (|) symbol.	See "DELIMITER Clause" on page 2-511.	Quoted String, p. 4-243
filename	Path and filename of file to read. Default pathname is current directory	See "LOAD FROM File" on page 2-505.	Conform to operating system rules
synonym, *table*, *view*	Synonym for the table in which to insert data from *filename*	*Synonym* and *table* or *view* to which it points must exist.	Database Object Name, p. 4-46

Usage

The LOAD statement appends new rows to the table. It does not overwrite existing data. You cannot add a row that has the same key as an existing row.

To use the LOAD statement, you must have Insert privileges for the table where you want to insert data. For information on database-level and table-level privileges, see the GRANT statement.

LOAD FROM File

The LOAD FROM file contains the data to be loaded into the specified table or view. The default pathname for the load file is the current directory.

You can use the file that the UNLOAD statement creates as the LOAD FROM file. (See "UNLOAD TO File" on page 2-754 for a description of how values of various data types are represented within the UNLOAD TO file.)

If you do not include a list of columns in the INSERT INTO clause, the fields in the file must match the columns that are specified for the table in number, order, and data type.

Each line of the file must have the same number of fields. You must define field lengths that are less than or equal to the length that is specified for the corresponding column. Specify only values that can convert to the data type of the corresponding column. The following table indicates how the database server expects you to represent the data types in the LOAD FROM file (when you use the default locale, U.S. English).

IDS

Type of Data	Input Format
blank	One or more blank characters between delimiters. You can include leading blanks in fields that do not correspond to character columns.
Boolean	A `'t'` or `'T'` indicates a TRUE value, and an `'f'` or `'F'` indicates a FALSE value. ♦
collections	Collection must have its values surrounded by braces ({}) and a field delimiter separating each element. For more information, see "Loading Complex Data Types" on page 2-510.
DATE	Character string in the following format: *mm/dd/year* You must state the month as a two-digit number. You can use a two-digit number for the year if the year is in the 20th century. (You can specify another century algorithm with the **DBCENTURY** environment variable.) The value must be an actual date; for example, February 30 is illegal. You can use a different date format if you indicate this format with the **GL_DATE** or **DBDATE** environment variable. For more information about environment variables, see the *IBM Informix Guide to SQL: Reference* and the *IBM Informix GLS User's Guide*.

(1 of 2)

Type of Data	Input Format
DECIMAL, MONEY, FLOAT	Value that can include a leading and/or trailing currency symbol and thousands and decimal separators. Your locale files or the **DBMONEY** environment variable can specify a currency format.
NULL	Nothing between the delimiters
ROW types (named or unnamed)	Row type must have its values surrounded by parentheses and a field delimiter that separates each element. For more information, see "Loading Complex Data Types" on page 2-510.
Simple large objects (TEXT, BYTE)	TEXT and BYTE columns are loaded directly from the LOAD TO file. For more information, see "Loading Simple Large Objects" on page 2-508.
Smart large objects (CLOB, BLOB)	CLOB and BLOB columns are loaded from a separate operating-system file. The field for the CLOB or BLOB column in the LOAD FROM file contains the name of this separate file. For more information, see "Loading Smart Large Objects" on page 2-509.
Time	Character string in *year-month-day hour:minute:second.fraction* format. You cannot use data type keywords or qualifiers for DATETIME or INTERVAL values. The year must be a 4-digit number, and the month must be a 2-digit number. The **DBTIME** or **GL_DATETIME** environment variable can specify other formats.
User-defined data formats (opaque types)	Associated opaque type must have an import support function defined if special processing is required to copy the data in the LOAD FROM file to the internal format of the opaque type. An import binary support function might also be required for data iin binary format. The LOAD FROM file data must be in the format that the import or import binary support function expects. The associated opaque type must have an assign support function if special processing is required before writing the data is in the database. See "Loading Opaque-Type Columns" on page 2-510.

(2 of 2)

For more information on **DB*** environment variables, refer to the *IBM Informix Guide to SQL: Reference*. For more information on **GL*** environment variables, refer to the *IBM Informix GLS User's Guide*.

GLS

If you are using a nondefault locale, the formats of DATE, DATETIME, MONEY, and numeric column values in the LOAD FROM file must be compatible with the formats that the locale supports for these data types. For more information, see the *IBM Informix GLS User's Guide*. ♦

The following example shows the contents of an input file named **new_custs**:

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo
Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo
Alto|CA|94301|(415)323-6440
```

This data file conveys the following information:

- Indicates a serial field by specifying a zero (0)
- Uses the pipe (|), the default delimiter
- Assigns NULL values to the **phone** field for the first row and the **address2** field for the second row. The NULL values are shown by two delimiters with nothing between them.

The following statement loads the values from the **new_custs** file into the **customer** table that **jason** owns:

```
LOAD FROM 'new_custs' INSERT INTO jason.customer
```

If you include any of the following special characters as part of the value of a field, you must precede the character with a backslash (\ ) escape symbol:

- Backslash
- Delimiter
- Newline character anywhere in the value of a VARCHAR or NVARCHAR column
- Newline character at end of a value for a TEXT value

Do not use the backslash character (\ ) as a field separator. It serves as an escape character to inform the LOAD statement that the next character is to be interpreted as part of the data, rather than as having special significance.

Fields that correspond to character columns can contain more characters than the defined maximum allows for the field. The extra characters are ignored.

If you are loading files that contain VARCHAR data types, note the following information:

- If you give the LOAD statement data in which the character fields (including VARCHAR) are longer than the column size, the excess characters are disregarded.
- Use the backslash (\ ) to escape embedded delimiter and backslash characters in all character fields, including VARCHAR.
- Do not use the following characters as delimiting characters in the LOAD FROM file: 0 to 9, a to f, A to F, backslash, newline character.

Loading Simple Large Objects

The database server loads simple large objects (BYTE and TEXT columns) directly from the LOAD FROM file. Keep the following restrictions in mind when you load BYTE and TEXT data:

- You cannot have leading and trailing blanks in BYTE fields.
- Use the backslash (\) to escape the special significance of literal delimiter and backslash characters in TEXT fields.
- Data being loaded into a BYTE column must be in ASCII-hexadecimal form. BYTE columns cannot contain preceding blanks.
- Do not use the following characters as delimiting characters in the LOAD FROM file: 0 to 9, a to f, A to F, backslash, newline character.

GLS

For TEXT columns, the database server handles any required code-set conversions for the data. See also the *IBM Informix GLS User's Guide*. ♦

If you are unloading files that contain BYTE or TEXT data types, objects smaller than 10 kilobytes are stored temporarily in memory. You can adjust the 10-kilobyte setting to a larger setting with the **DBBLOBBUF** environment variable. Simple large objects that are larger than the default or the setting of **DBBLOBBUF** are stored in a temporary file. For more information about the **DBBLOBBUF** environment variable, see the *IBM Informix Guide to SQL: Reference*.

IDS

Loading Smart Large Objects

The database server loads smart large objects (BLOB and CLOB columns) from a separate operating-system file on the client computer. For information on the structure of this file, see "Unloading Smart Large Objects" on page 2-756.

In a LOAD FROM file, a CLOB or BLOB column value appears as follows:

```
start_off,length,client_path
```

In this format, *start_off* is the starting offset (in hexadecimal) of the smart-large-object value within the client file, *length* is the length (in hexadecimal) of the BLOB or CLOB value, and *client_path* is the pathname for the client file. No blank spaces can appear between these values.

For example, to load a CLOB value that is 512 bytes long and is at offset 256 in the **/usr/apps/clob9ce7.318** file, the database server expects the CLOB value to appear as follows in the LOAD FROM file:

```
|100,200,/usr/apps/clob9ce7.318|
```

If the whole client file is to be loaded, a CLOB or BLOB column value appears as follows in the LOAD FROM file:

```
client_path
```

For example, to load a CLOB value that occupies the entire file **/usr/apps/clob9ce7.318**, the database server expects the CLOB value to appear as follows in the LOAD FROM file:

```
|/usr/apps/clob9ce7.318|
```

GLS

For CLOB columns, the database server handles any required code-set conversions for the data. See also the *IBM Informix GLS User's Guide*. ♦

IDS

Loading Complex Data Types

In a LOAD FROM file, complex data types appear as follows:

- Collections are introduced with the appropriate constructor (SET, MULTISET, or LIST), and their elements are enclosed in braces ({ }) and separated with a comma, as follows:

  ```
  constructor{val1 , val2 , ... }
  ```

  For example, to load the SET values {1, 3, 4} into a column whose data type is SET(INTEGER NOT NULL), the corresponding field of the LOAD FROM file appears as:

  ```
  |SET{1 , 3 , 4}|
  ```

- Row types (named and unnamed) are introduced with the ROW constructor and their fields are enclosed with parentheses and separated with a comma, as follows:

  ```
  ROW(val1 , val2 , ... )
  ```

  For example, to load the ROW values (1, 'abc'), the corresponding field of the LOAD FROM file appears as:

  ```
  |ROW(1 , abc)|
  ```

IDS

Loading Opaque-Type Columns

Some opaque data types require special processing when they are inserted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign()**. When you execute the LOAD statement on a table whose rows contain one of these opaque types, the database server automatically invokes the **assign()** function for the type. The **assign()** function can make the decision of how to store the data. For more information about the **assign()** support function, see the *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

DELIMITER Clause

Use the DELIMITER clause to specify the delimiter that separates the data contained in each column in a row in the input file. You can specify TAB (CTRL-I) or a blank space (= ASCII 32) as the delimiter symbol. You cannot use the following items as the delimiter symbol:

- Backslash (\)
- Newline character (CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

If you omit this clause, the database server checks the **DBDELIMITER** environment variable. For information about how to set the **DBDELIMITER** environment variable, see the *IBM Informix Guide to SQL: Reference*.

If the **DBDELIMITER** environment variable has not been set, the default delimiter is the pipe (|).

The following example specifies the semicolon (**;**) as the delimiting character. The example uses Windows file-naming conventions.

```
LOAD FROM 'C:\data\loadfile' DELIMITER ';'
   INSERT INTO orders
```

INSERT INTO Clause

Use the INSERT INTO clause to specify the table, synonym, or view in which to load the new data. You must specify the column names only if one of the following conditions is true:

- You are not loading data into all columns.
- The input file does not match the default order of the columns (the order specified when the table was created).

The following example identifies the **price** and **discount** columns as the only columns in which to add data. The example uses Windows filenaming conventions.

```
LOAD FROM 'C:\tmp\prices' DELIMITER ','
   INSERT INTO norman.worktab(price,discount)
```

Related Information

Related statements: UNLOAD and INSERT

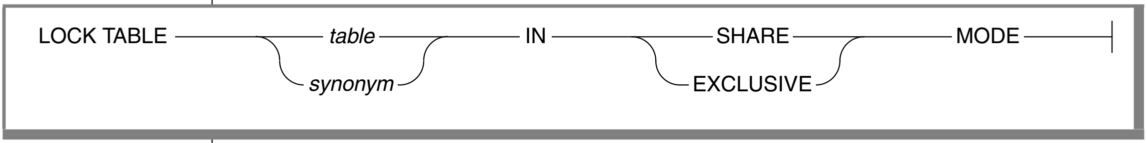
For a task-oriented discussion of the LOAD statement and other utilities for moving data, see the *IBM Informix Migration Guide*.

For a discussion of the GLS aspects of the LOAD statement, see the *IBM Informix GLS User's Guide*.

LOCK TABLE

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Synonym for the table to be locked	Synonym and the table to which it points must exist.	Database Object Name, p. 4-46
table	Table to be locked	See first paragraph of Usage.	Database Object Name, p. 4-46

Usage

You can use LOCK TABLE to lock a table if either of the following is true:

- You are the owner of the table.
- You have Select privilege on the table or on a column in the table, either from a direct grant or from a grant to PUBLIC.

The LOCK TABLE statement fails if the table is already locked in EXCLUSIVE mode by another process, or if you request an EXCLUSIVE lock while another user has locked the same table in SHARE mode.

The SHARE keyword locks a table in *shared mode*. Shared mode gives other processes *read* access to the table but denies *write* access. Other processes cannot update or delete data if a table is locked in shared mode.

The EXCLUSIVE keyword locks a table in *exclusive mode*. This mode denies other processes both read and write access to the table. Exclusive-mode locking automatically occurs during the ALTER INDEX, ALTER TABLE, CREATE INDEX, DROP INDEX, RENAME COLUMN, RENAME TABLE, START VIOLATIONS TABLE, and STOP VIOLATIONS TABLE statements.

Databases with Transactions

If your database was created with transactions, the LOCK TABLE statement succeeds only if it executes within a transaction. You must issue a BEGIN WORK statement before you can execute a LOCK TABLE statement.

ANSI

Transactions are implicit in an ANSI-compliant database. The LOCK TABLE statement succeeds whenever the specified table is not already locked by another process. ♦

The following guidelines apply to the use of the LOCK TABLE statement within transactions:

- You cannot lock system catalog tables.
- You cannot switch between shared and exclusive table locking within a transaction. For example, once you lock the table in shared mode, you cannot upgrade the lock mode to exclusive.
- If you issue a LOCK TABLE statement before you access a row in the table, and PDQ is not in effect, no row locks are set for the table. In this way, you can override row-level locking and avoid exceeding the maximum number of locks that are defined in the database server configuration. (But if PDQ is not in effect, you might run out of locks with error -134 unless the LOCKS parameter of your ONCONFIG file specifies a large enough number of locks.)
- All row and table locks release automatically after a transaction is completed. The UNLOCK TABLE statement fails in a database that uses transactions.
- The same user can explicitly use LOCK TABLE to lock up to 32 tables concurrently. (Use SET ISOLATION to specify an appropriate isolation level, such as Repeatable Read, if you need to lock rows from more than 32 tables during a single transaction.)

The following example shows how to change the locking mode of a table in a database that was created with transaction logging:

```
BEGIN WORK
LOCK TABLE orders IN EXCLUSIVE MODE
 ...
COMMIT WORK
BEGIN WORK
LOCK TABLE orders IN SHARE MODE
 ...
COMMIT WORK
```

Warning: *It is recommended that you not use nonlogging tables in a transaction. If you need to use a nonlogging table in a transaction, either lock the table in exclusive mode or set the isolation level to Repeatable Read to prevent concurrency problems.*

Databases Without Transactions

In a database that was created without transactions, table locks that were set by the LOCK TABLE statement are released after any of the following events:

- An UNLOCK TABLE statement executes.
- The user closes the database.
- The user exits the application.

To change the lock mode on a table, release the lock with the UNLOCK TABLE statement and then issue a new LOCK TABLE statement.

The following example shows how to change the lock mode of a table in a database that was created without transactions:

```
LOCK TABLE orders IN EXCLUSIVE MODE
   . . .
UNLOCK TABLE orders
   . . .
LOCK TABLE orders IN SHARE MODE
```

Related Information

Related statements: BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET ISOLATION, SET LOCK MODE, and UNLOCK TABLE

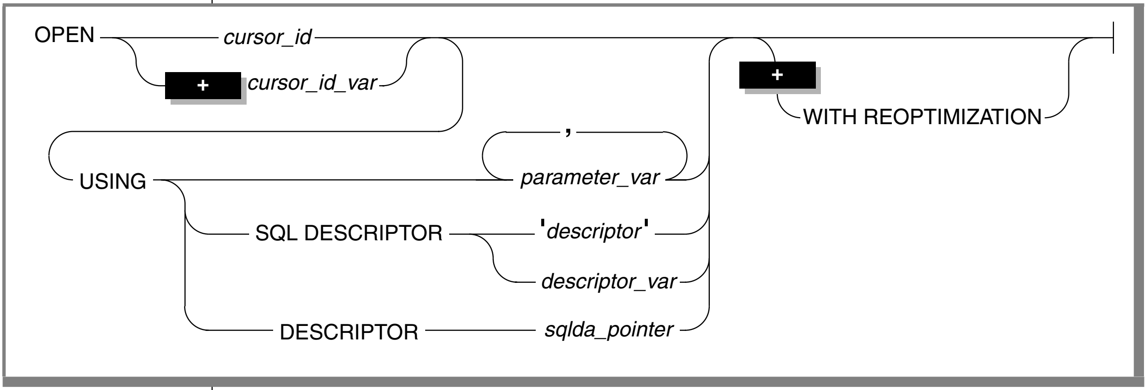
For a discussion of concurrency and locks, see the *IBM Informix Guide to SQL: Tutorial*.

E/C

OPEN

Use the OPEN statement to activate a cursor. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared	Identifier, p. 4-189
cursor_id_var	Host variable = *cursor_id*	Must be a character data type	Language specific
descriptor	Name of a system-descriptor area	Must have been allocated	Quoted String, p. 4-243
descriptor_var	Host variable that identifies the system-descriptor area	System-descriptor area must have been allocated	Quoted String, p. 4-243
parameter_var	Host variable whose contents replace a question (?) mark place-holder in a prepared statement	Must be a character or collection data type	Language specific
sqlda_pointer	Pointer to **sqlda** structure defining data type and memory location of values to replace question (?) marks in a prepared statement	Cannot begin with a dollar ($) sign nor with a colon (:). You must use an **sqlda** structure with dynamic SQL statements.	DESCRIBE, p. 2-351

Usage

A *cursor* is a database object that can contain an ordered set of values. The OPEN statement activates a cursor that the DECLARE statement created.

The database server supports the following types of cursors:

- A *select cursor*: a cursor that is associated with a SELECT statement
- A *function cursor*: a cursor that is associated with the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement
- An *insert cursor*: a cursor that is associated with the INSERT statement
- IDS A c*ollection curso*r: a select or insert cursor that operates on a collection variable ♦

The specific actions that the database server takes differ, depending on the statement with which the cursor is associated. When you associate one of the previous statements with a cursor directly (that is, you do not prepare the statement and associate the statement identifier with the cursor), the OPEN statement implicitly prepares the statement.

ANSI

In an ANSI-compliant database, you receive an error code if you try to open a cursor that is already open. ♦

Opening a Select Cursor

When you open either a select cursor or an update cursor that is created with the SELECT… FOR UPDATE syntax, the SELECT statement is passed to the database server with any values that are specified in the USING clause. The database server processes the query to the point of locating or constructing the first row of the active set. The following example illustrates a simple OPEN statement in ESQL/C:

```
EXEC SQL declare s_curs cursor for select * from orders;
EXEC SQL open s_curs;
```

Opening an Update Cursor Inside a Transaction

If you are working in a database with explicit transactions, you must open an update cursor within a transaction. This requirement is waived if you declared the cursor using the WITH HOLD option.

Opening a Function Cursor

When you open a function cursor, the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement is passed to the database server with any values that are specified in the USING clause.

The values in the USING clause are passed as arguments to the user-defined function. This user-defined function must be declared to accept values. (If the statement was previously prepared, the statement was passed to the database server when it was prepared.) The database server executes the function to the point where it returns the first set of values.

The following example illustrates a simple OPEN statement in ESQL/C:

```
EXEC SQL declare s_curs cursor for
   execute function new_func(arg1,arg2)
   into :ret_val1, :ret_val2;
EXEC SQL open s_curs;
```

XPS

In Extended Parallel Server, to re-create this example, use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

Reopening a Select or Function Cursor

The database server evaluates the values that are named in the USING clause of the OPEN statement only when it opens the select or function cursor. While the cursor is open, subsequent changes to program variables in the USING clause do not change the active set of the cursor.

ANSI

In a database that is ANSI-compliant, you receive an error code if you try to open a cursor that is already open. ♦

In a database that is not ANSI-compliant, a subsequent OPEN statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new active set, based on the current values of the variables in the USING clause. If the variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different active set.

Even if the values of the variables are unchanged, the values in the active set can be different, in the following situations:

- If the user-defined function takes a different execution path from the previous OPEN statement on a function cursor
- If data in the table was modified since the previous OPEN statement on a select cursor

The database server can process most queries dynamically, without pre-fetching all rows when it opens the select or function cursor. Therefore, if other users are modifying the table at the same time that the cursor is being processed, the active set might reflect the results of these actions.

For some queries, the database server evaluates the entire active set when it opens the cursor. These queries include those with the following features:

- Queries that require sorting: those with an ORDER BY clause or with the DISTINCT or UNIQUE keyword
- Queries that require hashing: those with a join or with the GROUP BY clause

For these queries, any changes that other users make to the table while the cursor is being processed are not reflected in the active set.

Errors Associated with Select and Function Cursors

Because the database server is seeing the query for the first time, it might detect errors. In this case, the database server does not actually return the first row of data, but it sets a return code in the **sqlca.sqlcode, SQLCODE** field of the sqlca. The return code value is either negative or zero, as the following table describes.

Return Code Value	Purpose
Negative	Shows an error is detected in the SELECT statement
Zero	Shows the SELECT statement is valid

If the SELECT, SELECT…FOR UPDATE, EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement is valid, but no rows match its criteria, the first FETCH statement returns a value of `100` (`SQLNOTFOUND`), which means no rows were found.

Tip: *When you encounter an* ***SQLCODE*** *error, a corresponding* ***SQLSTATE*** *error value also exists. For information about how to get the message text, check the GET DIAGNOSTICS statement.*

Opening an Insert Cursor

When you open an insert cursor, the cursor passes the INSERT statement to the database server, which checks the validity of the keywords and column names. The database server also allocates memory for an insert buffer to hold new data. (See "DECLARE" on page 2-323.)

An OPEN statement for a cursor that is associated with an INSERT statement cannot include a USING clause.

Example of Opening an Insert Cursor

The following ESQL/C example illustrates an OPEN statement with an insert cursor:

```
EXEC SQL prepare s1 from
   'insert into manufact values ('npr', 'napier')';
EXEC SQL declare in_curs cursor for s1;
EXEC SQL open in_curs;
EXEC SQL put in_curs;
EXEC SQL close in_curs;
```

Reopening an Insert Cursor

When you reopen an insert cursor that is already open, you effectively flush the insert buffer; any rows that are stored in the insert buffer are written into the database table. The database server first closes the cursor, which causes the flush and then reopens the cursor. For information about how to check errors and count inserted rows, see "Error Checking" on page 2-546.

ANSI

In an ANSI-compliant database, you receive an error code if you try to open a cursor that is already open. ♦

IDS

Opening a Collection Cursor

You can declare both select and insert cursors on collection variables. Such cursors are called collection cursors. You must use the OPEN statement to activate these cursors.

Use the name of a collection variable in the USING clause of the OPEN statement. For more information on the use of OPEN...USING with a collection variable, see "Fetching from a Collection Cursor" on page 2-432 and "Inserting into a Collection Cursor" on page 2-544.

USING Clause

The USING clause is required when the cursor is associated with a prepared statement that includes question-mark (?) placeholders, as follows:

- A SELECT statement with input parameters in its WHERE clause
- An EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with input parameters as arguments of its user-defined function
- An INSERT statement with input parameters in its VALUES clause

You can supply values for these parameters in one of the following ways:

- You can specify one or more host variables.
- You can specify a system-descriptor area.
- You can specify a pointer to an **sqlda** structure.

(For more information, see "PREPARE" on page 2-527.)

If you know the number of parameters to be supplied at runtime and their data types, you can define the parameters that are needed by the statement as host variables in your program. You pass parameters to the database server by opening the cursor with the USING keyword, followed by the names of the variables. These variables are matched with the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement question-mark (?) parameters in a one-to-one correspondence, from left to right.

You cannot include indicator variables in the list of variable names. To use an indicator variable, you must include the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement as part of the DECLARE statement.

You must supply one host variable name for each placeholder. The data type of each variable must be compatible with the corresponding type that the prepared statement requires. The following ESQL/C code fragment opens a select cursor and specifies host variables in the USING clause:

```
sprintf (select_1, "%s %s %s %s %s",
   "SELECT o.order_num, sum(total price)",
   "FROM orders o, items i",
   "WHERE o.order_date > ? AND o.customer_num = ?",
   "AND o.order_num = i.order_num",
   "GROUP BY o.order_num");
EXEC SQL prepare statement_1 from :select_1;
EXEC SQL declare q_curs cursor for statement_1;
EXEC SQL open q_curs using :o_date, :o.customer_num;
```

The following example illustrates the USING clause of the OPEN statement with an EXECUTE FUNCTION statement in an ESQL/C code fragment:

```
stcopy ("EXECUTE FUNCTION one_func(?, ?)", exfunc_stmt);
EXEC SQL prepare exfunc_id from :exfunc_stmt;
EXEC SQL declare func_curs cursor for exfunc_id;
EXEC SQL open func_curs using :arg1, :arg2;
```

XPS

In Extended Parallel Server, to re-create this example use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

Specifying a System Descriptor Area

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate input values from a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values to replace question-mark (?) placeholders.

X/O

A system-descriptor area conforms to the X/Open standards. ♦

Use the SQL DESCRIPTOR keywords to introduce the name of a system descriptor area as the location of the parameters.

The COUNT field in the system-descriptor area corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement. You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

The following example shows the OPEN...USING SQL DESCRIPTOR statement:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL open selcurs using sql descriptor 'desc1';
```

As the example indicates, the system descriptor area must be allocated before you reference it in the OPEN statement.

E/C

Specifying a Pointer to an sqlda Structure

If you do not know the number of parameters to be supplied at runtime, or their data types, you can associate input values from an **sqlda** structure. An **sqlda** structure lists the data type and memory location of one or more values to replace question-mark (?) placeholders.

Use the DESCRIPTOR keyword to introduce a pointer to the **sqlda** structure as the location of the parameters.

The **sqlda** value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement.

Example of Specifying a Pointer to an sqlda Structure

The following example shows an OPEN...USING DESCRIPTOR statement:

```
struct sqlda *sdp;
...
EXEC SQL open selcurs using descriptor sdp;
```

Using the WITH REOPTIMIZATION Option

Use the WITH REOPTIMIZATION keywords to reoptimize your query plan. When you prepare SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statements, the database server uses a query plan to optimize the query. If you later modify the data associated with the prepared statement, you can compromise the effectiveness of the query plan for that statement. In other words, if you change the data, you might deoptimize your query. To ensure optimization of your query, you can prepare the statement again, or open the cursor again using the WITH REOPTIMIZATION option.

You should generally use the WITH REOPTIMIZATION option, because it provides the following advantages over preparing a statement again:

- Rebuilds only the query plan, rather than the entire statement
- Uses fewer resources
- Reduces overhead
- Requires less time

The WITH REOPTIMIZATION option forces the database server to optimize the query-design plan before it processes the OPEN cursor statement.

The following example uses the WITH REOPTIMIZATION keywords:

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you execute a FREE *statement_id* or FREE *statement_id_var* statement, you can still open the cursor associated with the freed statement ID. If you release resources with a FREE *cursor_id* or FREE *cursor_id_var* statement, however, you cannot use the cursor unless you declare the cursor again.

Similarly, if you use the SET AUTOFREE statement for one or more cursors, when the program closes the specific cursor, the database server automatically frees the cursor-related resources. In this case, you cannot use the cursor unless you declare the cursor again.

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, CLOSE, DECLARE, EXECUTE, FETCH, FLUSH, FREE, GET DESCRIPTOR, PREPARE, PUT, SET AUTOFREE, SET DEFERRED_PREPARE, and SET DESCRIPTOR

For a task-oriented discussion of the OPEN statement, see the *IBM Informix Guide to SQL: Tutorial*.

For more information on system-descriptor areas and the **sqlda** structure, refer to the *IBM Informix ESQL/C Programmer's Manual*.

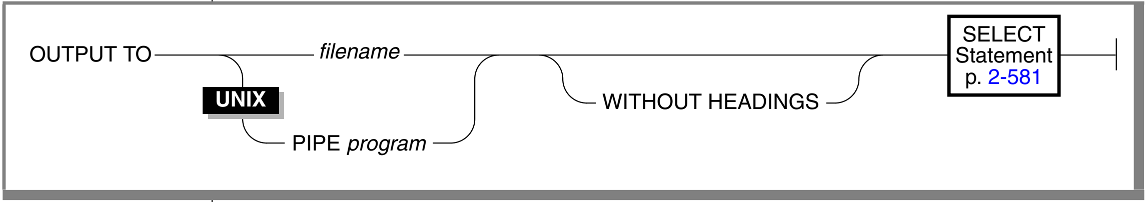
+

DB

OUTPUT

The OUTPUT statement writes query results in an operating-system file, or pipes query results to another program. Use this statement with DB-Access.

Syntax



Element	Purpose	Restrictions	Syntax
filename	Path and filename where query results are written. The default path is the current directory.	Can specify a new or existing file. If the file exists, the query results overwrite the current contents of the file.	Must conform to the rules of your operating system.
program	Name of a program to receive the query results as input	Program must exist, must be known to the operating system, and must be able to read the results of a query.	Must conform to the rules of your operating system.

Usage

You can use the OUTPUT statement to direct the results of a query to an operating-system file or to a program. You can also specify whether column headings should be omitted from the query output.

Sending Query Results to a File

To send the results of a query to an operating-system file, specify the full pathname for the file. If the file already exists, the output overwrites the current contents.

The following examples show how to send the result of a query to an operating-system file. The example uses UNIX filenaming conventions.

```
OUTPUT TO /usr/april/query1
   SELECT * FROM cust_calls WHERE call_code = 'L'
```

Displaying Query Results Without Column Headings

To display the results of a query without column headings, use the WITHOUT HEADINGS keywords.

UNIX

Sending Query Results to Another Program

In the UNIX environment, you can use the keyword PIPE to send the query results to another program, as the following example shows:

```
OUTPUT TO PIPE more
   SELECT customer_num, call_dtime, call_code
      FROM cust_calls
```

Related Information

Related statements: SELECT and UNLOAD

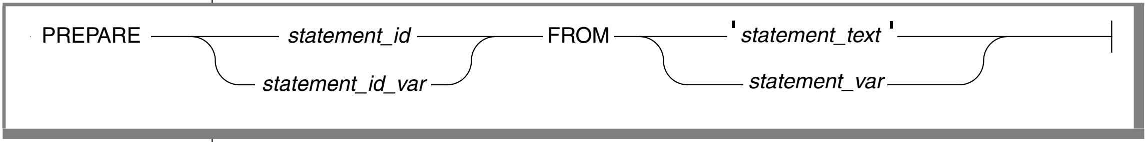
+

E/C

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements at runtime. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
statement_id	Identifier declared here for the prepared object	Must be unique in the database among names of cursors and prepared objects.	Identifier, p. 4-189
statement_id_var	Host variable storing *statement_id*	Must have been previously declared as a character data type	Language specific
statement_text	Text of the SQL statement(s) to prepare	See "Preparing Multiple SQL Statements" on page 2-536 and "Statement Text" on page 2-529	Quoted String, p. 4-243.
statement_var	Host variable storing the text of one or more SQL statements	Must be a character data type. Not valid if the SQL statement(s) contains the Collection-Derived-Table segment.	Language specific

Usage

The PREPARE statement enables your program to assemble the text of one or more SQL statements at runtime (creating a *prepared object*) and make it executable. This dynamic form of SQL is accomplished in three steps:

1. A PREPARE statement accepts statement text as input, either as a quoted string or stored within a character variable. Statement text can contain question-mark (?) placeholders to represent values that are to be defined when the statement is executed.
2. An EXECUTE or OPEN statement can supply the required input values and execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later using the FREE statement.

IDS

The collating order that is current when the PREPARE statement creates a prepared object is also used when that object is executed, even if the execution-time collation of the session (or of **DB_LOCALE**) is different. ♦

Restrictions

The number of prepared objects in a single program is limited by available memory. These include statement identifiers declared in PREPARE statements (*statement_id* or *statement_id_var*) and declared cursors. To avoid exceeding the limit, use the FREE statement to release some statements or cursors.

The maximum length of a PREPARE statement is 64 kilobytes.

For restrictions on the statements in the character string, see "Restricted Statements in Single-Statement Prepares" on page 2-531 and "Restricted Statements in Multistatement Prepared Objects" on page 2-537.

Using a Statement Identifier

PREPARE sends the statement text to the database server, which analyzes the statement text. If the text contains no syntax errors, the database server translates it to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The name of the structure is the value that is assigned to the statement identifier in the PREPARE statement. Subsequent SQL statements refer to the structure by using the same statement identifier that was used in the PREPARE statement.

A subsequent FREE statement releases the database server resources that were allocated to the statement. After you release these resources with FREE, you cannot use the statement identifier in a DECLARE statement or with the EXECUTE statement until you prepare the statement again.

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of a statement identifier is global to the program. Therefore, a statement identifier that is prepared in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of a statement identifier to the file in which it is prepared, preprocess all the files with the **-local** command-line option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or series of statements at a time. A new PREPARE statement can specify an existing statement identifier if you want to bind the t identifier to a different SQL statement text.

The PREPARE statement supports dynamic statement-identifier names, which allow you to prepare a statement identifier as an identifier or as a host variable of a data type that can contain a character string. The first example that follows shows a statement identifier that was specified as a host variable. The second specifies a statement identifier as a character-string.

```
stcopy ("query2", stmtid);
EXEC SQL prepare :stmtid from 'select * from customer';

EXEC SQL prepare query2 from 'select * from customer';
```

The variable must be a character data type. In C, it must be declared as `char`.

Statement Text

The PREPARE statement can take statement text either as a quoted string or as text that is stored in a program variable. The following restrictions apply to the statement text:

- The text can contain only SQL statements. It cannot contain statements or comments *from* the host-programming language.
- The text can contain comments that are preceded by a double dash (--) or enclosed in braces ({ }).

  These comment symbols represent SQL comments. For more information on SQL comment symbols, see "How to Enter SQL Comments" on page 1-6.
- The text can contain either a single SQL statement or a series of statements that are separated by semicolon (;) symbols.

  For a list of SQL statements that cannot be prepared, see "Restricted Statements in Single-Statement Prepares" on page 2-531. For more information on how to prepare multiple SQL statements, see "Preparing Multiple SQL Statements" on page 2-536.
- The text cannot include an embedded SQL statement prefix or terminator, such as a dollar sign ($) or the words EXEC SQL.

- Host-language variables are not recognized as such in prepared text.

  Therefore, you cannot prepare a SELECT (or EXECUTE FUNCTION or EXECUTE PROCEDURE) statement that includes an INTO clause, because the INTO clause requires a host-language variable.
- The only identifiers that you can use are names that are defined in the database, such as names of tables and columns. For more information on how to use identifiers in statement text, see "Preparing Statements with SQL Identifiers" on page 2-533.
- Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes, as in this ESQL/C example:

  ```
  EXEC SQL prepare new_cust from
     'insert into customer(fname,lname) values(?,?)';
  ```

  For more information on how to use question marks as placeholders, see "Preparing Statements That Receive Parameters" on page 2-532.

IDS

If the prepared statement contains the Collection-Derived-Table segment on an ESQL/C collection variable, some additional limitations exist on how you can assemble the text for the PREPARE statement. For information about dynamic SQL, see the *IBM Informix ESQL/C Programmer's Manual*. ♦

Preparing and Executing User-Defined Routines

The way to prepare a user-defined routine (UDR) depends on whether the UDR is a user-defined procedure or a user-defined function:

- To prepare a user-defined procedure, prepare the EXECUTE PROCEDURE statement that executes the procedure.

  To execute the prepared procedure, use the EXECUTE statement.
- To prepare a user-defined function, prepare the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that executes the function.

  You cannot include the INTO clause of EXECUTE FUNCTION (or EXECUTE PROCEDURE) in the PREPARE statement.

How to execute a prepared user-defined function depends on whether it returns only one group or multiple groups of values. Use the EXECUTE statement for user-defined functions that return only one group of values.

To execute user-defined functions that return more than one group of return values, you must associate the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor.

Restricted Statements in Single-Statement Prepares

In general, you can prepare any database manipulation (DML) statement.

IDS

You can prepare any single SQL statement except the following statements:

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
CLOSE
CONNECT
CREATE FUNCTION FROM
CREATE PROCEDURE FROM
CREATE ROUTINE FROM
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
INFO
LOAD
OPEN
OUTPUT
PREPARE
PUT
SET AUTOFREE
SET CONNECTION
SET DEFERRED_PREPARE
SET DESCRIPTOR
UNLOAD
WHENEVER

♦

XPS

You can prepare any single SQL statement except the following statements:

ALLOCATE DESCRIPTOR
CLOSE
CONNECT
CREATE PROCEDURE FROM
DEALLOCATE DESCRIPTOR
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
INFO
LOAD
OPEN
OUTPUT
PREPARE
PUT
SET CONNECTION
SET DEFERRED_PREPARE
SET DESCRIPTOR
UNLOAD
WHENEVER ♦

You can prepare a SELECT statement. If SELECT includes the INTO TEMP clause, you can execute the prepared statement with an EXECUTE statement. If it does not include the INTO TEMP clause, the statement returns rows of data. Use DECLARE, OPEN, and FETCH cursor statements to retrieve the rows.

A prepared SELECT statement can include a FOR UPDATE clause. This clause is used with the DECLARE statement to create an update cursor. The next example shows a SELECT statement with a FOR UPDATE clause in ESQL/C:

```
sprintf(up_query, "%s %s %s",
   "select * from customer ",
   "where customer_num between ? and ? ",
   "for update");
EXEC SQL prepare up_sel from :up_query;
EXEC SQL declare up_curs cursor for up_sel;
EXEC SQL open up_curs using :low_cust,:high_cust;
```

Preparing Statements When Parameters Are Known

In some prepared statements, all necessary information is known at the time the statement is prepared. The following example in ESQL/C shows two statements that were prepared from constant data:

```
sprintf(redo_st, "%s %s",
   "drop table workt1; ",
   "create table workt1 (wtk serial, wtv float)" );
EXEC SQL prepare redotab from :redo_st;
```

Preparing Statements That Receive Parameters

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question-mark (?) placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following ESQL/C examples show some uses of question-mark (?) placeholders:

```
EXEC SQL prepare s3 from
   'select * from customer where state matches ?';
EXEC SQL prepare in1 from 'insert into manufact values (?,?,?)';
sprintf(up_query, "%s %s",
   "update customer set zipcode = ?"
   "where current of zip_cursor");
EXEC SQL prepare update2 from :up_query;
EXEC SQL prepare exfunc from
   'execute function func1 (?, ?)';
```

You can use a placeholder to defer evaluation of a value until runtime only for an expression, but not for an SQL identifier, except as noted in "Preparing Statements with SQL Identifiers" on page 2-533.

The following example of an ESQL/C code fragment prepares a statement from a variable that is named **demoquery**. The text in the variable includes one question-mark (?) placeholder. The prepared statement is associated with a cursor and, when the cursor is opened, the USING clause of the OPEN statement supplies a value for the placeholder.

```
EXEC SQL BEGIN DECLARE SECTION;
   char queryvalue [6];
   char demoquery  [80];
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to 'stores_demo';
sprintf(demoquery, "%s %s",
      "select fname, lname from customer ",
      "where lname > ? ");
EXEC SQL prepare quid from :demoquery;
EXEC SQL declare democursor cursor for quid;
stcopy("C", queryvalue);
EXEC SQL open democursor using :queryvalue;
```

The USING clause is available in both OPEN statements that are associated with a cursor and EXECUTE statements (all other prepared statements).

You can use a question-mark (?) placeholder to represent the name of an ESQL/C or SPL collection variable.

Preparing Statements with SQL Identifiers

In general, you must specify SQL identifiers explicitly in the statement text when you prepare the statement. In a few special cases, however, you can use the question-mark (?) placeholder for an SQL identifier:

- For the database name in the DATABASE statement.
- For the dbspace name in the IN *dbspace* clause of the CREATE DATABASE statement.
- For the cursor name in statements that use cursor names.

Obtaining SQL Identifiers from User Input

If a prepared statement requires identifiers, but the identifiers are unknown when you write the prepared statement, you can construct a statement that receives SQL identifiers from user input.

The following ESQL/C example prompts the user for the name of a table and uses that name in a SELECT statement. Because this name is unknown until runtime, the number and data types of the table columns are also unknown. Therefore, the program cannot allocate host variables to receive data from each row in advance. Instead, this program fragment describes the statement into an **sqlda** descriptor and fetches each row with the descriptor. The fetch puts each row into memory locations that the program provides dynamically.

If a program retrieves all the rows in the active set, the FETCH statement would be placed in a loop that fetched each row. If the FETCH statement retrieves more than one data value (column), another loop exists after the FETCH, which performs some action on each data value.

```
#include <stdio.h>
EXEC SQL include sqlda;
EXEC SQL include sqltypes;

char *malloc( );

main()
{
   struct sqlda *demodesc;
   char tablename[19];
   int i;
EXEC SQL BEGIN DECLARE SECTION;
   char demoselect[200];
EXEC SQL END DECLARE SECTION;

/*  This program selects all the columns of a given tablename.
       The tablename is supplied interactively. */

EXEC SQL connect to 'stores_demo';

printf( "This program does a select * on a table\n" );
printf( "Enter table name: " );
scanf( "%s", tablename );

sprintf(demoselect, "select * from %s", tablename );

EXEC SQL prepare iid from :demoselect;
EXEC SQL describe iid into demodesc;

/* Print what describe returns */

for ( i = 0;  i < demodesc->sqld; i++ )
   prsqlda (demodesc->sqlvar + i);
```

```
/* Assign the data pointers. */

for ( i = 0;  i < demodesc->sqld; i++ )
   {
   switch (demodesc->sqlvar[i].sqltype & SQLTYPE)
      {
      case SQLCHAR:
         demodesc->sqlvar[i].sqltype = CCHARTYPE;
         /* make room for null terminator */
         demodesc->sqlvar[i].sqllen++;
         demodesc->sqlvar[i].sqldata =
            malloc( demodesc->sqlvar[i].sqllen );
         break;

      case SQLSMINT:    /* fall through */
      case SQLINT:      /* fall through */
      case SQLSERIAL:
         demodesc->sqlvar[i].sqltype = CINTTYPE;
         demodesc->sqlvar[i].sqldata =
            malloc( sizeof( int ) );
         break;
      /*  And so on for each type.  */
      }
   }

/* Declare and open cursor for select . */
EXEC SQL declare d_curs cursor for iid;
EXEC SQL open d_curs;

/* Fetch selected rows one at a time into demodesc. */
for( ; ; )
   {
   printf( "\n" );
   EXEC SQL fetch d_curs using descriptor demodesc;
   if ( sqlca.sqlcode != 0 )
        break;
   for ( i = 0;  i < demodesc->sqld; i++ )
      {
      switch (demodesc->sqlvar[i].sqltype)
         {
         case CCHARTYPE:
            printf( "%s: \"%s\n", demodesc->sqlvar[i].sqlname,
               demodesc->sqlvar[i].sqldata );
            break;
         case CINTTYPE:
            printf( "%s: %d\n", demodesc->sqlvar[i].sqlname,
               *((int *) demodesc->sqlvar[i].sqldata) );
            break;
         /* And so forth for each type... */
            }
      }
   }
EXEC SQL close d_curs;
EXEC SQL free d_curs;


/*  Free the data memory.  */

for ( i = 0;  i < demodesc->sqld; i++ )
   free( demodesc->sqlvar[i].sqldata );
free( demodesc );
```

```
    printf ("Program Over.\n");
    }


    prsqlda(sp)
       struct sqlvar_struct *sp;

       {
       printf ("type = %d\n", sp->sqltype);
       printf ("len = %d\n", sp->sqllen);
       printf ("data = %lx\n", sp->sqldata);
       printf ("ind = %lx\n", sp->sqlind);
       printf ("name = %s\n", sp->sqlname);
       }
```

Preparing Multiple SQL Statements

You can execute several SQL statements as one action if you include them in the same PREPARE statement. Multistatement text is processed as a unit; actions are not treated sequentially. Therefore, multistatement text cannot include statements that depend on actions that occur in a previous statement in the text. For example, you cannot create a table and insert values into that table in the same prepared statement block.

If a statement in a multistatement prepare returns an error, the whole prepared statement stops executing. The database server does not execute any remaining statements. In most situations, compiled products return error-status information on the error, but do not indicate which statement in the text causes an error. You can use the **sqlca.sqlerrd[4]** field in the **sqlca** to find the offset of the errors.

In a multistatement prepare, if no rows are returned from a WHERE clause in the following statements, the database server returns `SQLNOTFOUND (100)`:

- UPDATE…WHERE…
- SELECT INTO TEMP…WHERE…
- INSERT INTO…WHERE…
- DELETE FROM…WHERE…

In the next example, four SQL statements are prepared into a single ESQL/C string called **query**. Individual statements are delimited with semicolons.

A single PREPARE statement can prepare the four statements for execution, and a single EXECUTE statement can execute the statements that are associated with the **qid** statement identifier.

```
sprintf (query,  "%s %s %s %s %s %s %s",
   "update account set balance = balance + ? ",
      "where acct_number = ?;",
   "update teller set balance = balance + ? ",
      "where teller_number = ?;",
   "update branch set balance = balance + ? ",
      "where branch_number = ?;",
   "insert into history values (?, ?);";
EXEC SQL prepare qid from :query;

EXEC SQL begin work;
EXEC SQL execute qid using
      :delta, :acct_number, :delta, :teller_number,
      :delta, :branch_number, :timestamp, :values;
EXEC SQL commit work;
```

Here the semicolons (**;**) are required as SQL statement-terminator symbols between each SQL statement in the text that **query** holds.

Restricted Statements in Multistatement Prepared Objects

In addition to the statements listed as exceptions in "Restricted Statements in Single-Statement Prepares" on page 2-531, you cannot use the following statements in the text of a multiple-statement prepared object: ti

CLOSE DATABASE
CREATE DATABASE
DATABASE
DROP DATABASE
SELECT (*with one exception*)

Moreover, the following types of statements are not allowed in a multistatement prepare:

- Statements that can cause the current database to close during the execution of the multistatement sequence
- Statements that include references to TEXT or BYTE host variables

In general, you cannot use the SELECT statement in a multistatement prepare. The only form of the SELECT statement allowed in a multistatement prepare is a SELECT statement with an INTO temporary table clause.

Using Prepared Statements for Efficiency

To increase performance efficiency, you can use the PREPARE statement and an EXECUTE statement in a loop to eliminate overhead that redundant parsing and optimizing cause. For example, an UPDATE statement that is located within a WHILE loop is parsed each time the loop runs. If you prepare the UPDATE statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution. The following example shows how to prepare an ESQL/C statement to improve performance:

```
EXEC SQL BEGIN DECLARE SECTION;
   char disc_up[80];
   int cust_num;
EXEC SQL END DECLARE SECTION;

main()
{
   sprintf(disc_up, "%s %s","update customer ",
      "set discount = 0.1 where customer_num = ?");
   EXEC SQL prepare up1 from :disc_up;

   while (1)
      {
      printf("Enter customer number (or 0 to quit): ");
      scanf("%d", cust_num);
      if (cust_num == 0)
         break;
      EXEC SQL execute up1 using :cust_num;
      }
}
```

Related Information

Related statements: CLOSE, DECLARE, DESCRIBE, EXECUTE, FREE, OPEN, SET AUTOFREE, and SET DEFERRED_PREPARE

For information about basic concepts that relate to the PREPARE statement, see the *IBM Informix Guide to SQL: Tutorial*.

For information about more advanced concepts that relate to the PREPARE statement, see the *IBM Informix ESQL/C Programmer's Manual*.

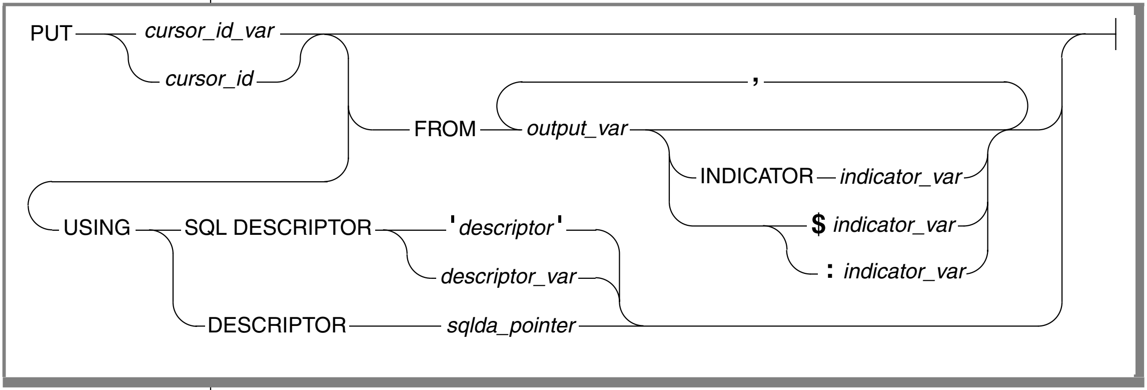
+

E/C

PUT

Use the PUT statement to store a row in an insert buffer for later insertion into the database. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor	Must be open	Identifier, p. 4-189
cursor_id_var	Host variable = *cursor_id*	Must be a character type; cursor must be open	Language specific
descriptor	Name of a system-descriptor area	Must already be allocated	Quoted String, p. 4-243
descriptor_var	Host-variable that contains *descriptor*	Must already be allocated	Quoted String, p. 4-243
indicator_var	Host variable to receive a return code if corresponding *output_var* receives a NULL value	Cannot be a DATETIME or INTERVAL data type	Language specific
output_var	Host variable whose contents replace a question-mark (?) placeholder in a prepared INSERT statement	Must be a character data type	Language specific
sqlda_pointer	Pointer to an sqlda structure	First character cannot be the ($) or (:) symbol	DESCRIBE, p. 2-351

Usage

PUT stores a row in an *insert buffer* that is created when the cursor is opened.

If the buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block, and the buffer is emptied. As a result, some PUT statement executions cause rows to be written to the database, and some do not.

You can use the FLUSH statement to write buffered rows to the database without adding a new row. The CLOSE statement writes any remaining rows before it closes an insert cursor.

If the current database uses explicit transactions, you must execute a PUT statement within a transaction.

The following example uses a PUT statement in ESQL/C:

```
EXEC SQL prepare ins_mcode from
   'insert into manufact values(?,?)';
EXEC SQL declare mcode cursor for ins_mcode;
EXEC SQL open mcode;
EXEC SQL put mcode from :the_code, :the_name;
```

X/O

The PUT statement is not an X/Open SQL statement. Therefore, you get a warning message if you compile a PUT statement in X/Open mode. ♦

Supplying Inserted Values

The values in the inserted row can come from one of the following sources:

- Constant values that are written into the INSERT statement
- Program variables that are named in the INSERT statement
- Program variables in the FROM clause of the PUT statement
- Values that are prepared in memory addressed by an **sqlda** structure or a system-descriptor area and then specified in the USING clause of the PUT statement

The system descriptor area or **sqlda** structure that *descriptor* or *sqlda_pointer* references must define a data type and memory location of each value that corresponds to a question-mark (?) placeholder in a prepared INSERT statement.

Using Constant Values in INSERT

The VALUES clause lists the values for the inserted columns. One or more of these values can be constants (that is, numbers or character strings).

When *all* the inserted values are constants, the PUT statement has a special effect. Instead of creating a row and putting it in the buffer, the PUT statement merely increments a counter. When you use a FLUSH or CLOSE statement to empty the buffer, one row and a repetition count are sent to the database server, which inserts that number of rows. In the following ESQL/C example, 99 empty customer records are inserted into the **customer** table. Because all values are constants, no disk output occurs until the cursor closes. (The constant zero for **customer_num** causes generation of a SERIAL value.)

```
int count;
EXEC SQL declare fill_c cursor for
   insert into customer(customer_num) values(0);
EXEC SQL open fill_c;
for (count = 1; count <= 99; ++count)
   EXEC SQL put fill_c;
EXEC SQL close fill_c;
```

Naming Program Variables in INSERT

When you associate the INSERT statement with a cursor (in the DECLARE statement), you create an insert cursor. In the INSERT statement, you can name program variables in the VALUES clause. When each PUT statement is executed, the contents of the program variables at that time are used to populate the row that is inserted into the buffer.

If you are creating an insert cursor (using DECLARE with INSERT), you must use only program variables in the VALUES clause. Variable names are not recognized in the context of a prepared statement; you associate a prepared statement with a cursor through its statement identifier.

The following ESQL/C example illustrates the use of an insert cursor. The code includes the following statements:

- The DECLARE statement associates a cursor called ins_curs with an INSERT statement that inserts data into the **customer** table.

  The VALUES clause specifies a data structure that is called **cust_rec**; the ESQL/C preprocessor converts **cust_rec** to a list of values, one for each component of the structure.
- The OPEN statement creates a buffer.
- A user-defined function (not defined within this example) obtains customer information from user input and stores it in **cust_rec**.

- The PUT statement composes a row from the current contents of the **cust_rec** structure and sends it to the row buffer.
- The CLOSE statement inserts into the **customer** table any rows that remain in the row buffer and closes the insert cursor.

```
int keep_going = 1;
EXEC SQL BEGIN DECLARE SECTION
   struct cust_row { /* fields of a row of customer table */ } cust_rec;
EXEC SQL END DECLARE SECTION

EXEC SQL declare ins_curs cursor for
      insert into customer values (:cust_row);
EXEC SQL open ins_curs;
while ( (sqlca.sqlcode == 0) && (keep_going) )
   {
   keep_going = get_user_input(cust_rec); /* ask user for new customer */
   if (keep_going )                       /* user did supply customer info
*/
      {
      cust_rec.customer_num = 0;          /* request new serial value */
      EXEC SQL put ins_curs;
      }
   if (sqlca.sqlcode == 0)                /* no error from PUT */
      keep_going = (prompt_for_y_or_n("another new customer") =='Y')
   }
EXEC SQL close ins_curs;
```

Use an indicator variable if the data to be inserted by the INSERT statement might be NULL.

Naming Program Variables in PUT

When the INSERT statement is prepared (see "PREPARE" on page 2-527), you cannot use program variables in its VALUES clause, but you can represent values by a question-mark (?) placeholder. List the program variables in the FROM clause of the PUT statement to supply the missing values.

The following ESQL/C example lists host variables in a PUT statement:

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
   char ins_comp[80];
   char u_company[20];
EXEC SQL END DECLARE SECTION;

main()
{
   EXEC SQL connect to 'stores_demo';
   EXEC SQL prepare ins_comp from
      'insert into customer (customer_num, company) values (0, ?)';
   EXEC SQL declare ins_curs cursor for ins_comp;
   EXEC SQL open ins_curs;

   while (1)
```

```
      {
      printf("\nEnter a customer: ");
      gets(u_company);
      EXEC SQL put ins_curs from :u_company;
      printf("Enter another customer (y/n) ? ");
      if (answer = getch() != 'y')
         break;
      }
   EXEC SQL close ins_curs;
   EXEC SQL disconnect all;
}
```

Indicator variables are optional, but you should use an indicator variable if the possibility exists that *output_var* might contain a NULL value.

If you specify the indicator variable without the INDICATOR keyword, you cannot put a blank space between *output_var* and *indicator_var*.

Using the USING Clause

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate input values from a system-descriptor area or an **sqlda** structure. Both of these descriptor structures describe the data type and memory location of one or more values to replace question-mark (?) placeholders.

Each time the PUT statement executes, the values that the descriptor structure describes are used to replace question-mark (?) placeholders in the INSERT statement. This process is similar to using a FROM clause with a list of variables, except that your program has full control over the memory location of the data values.

Specifying a System-Descriptor Area

The SQL DESCRIPTOR option specifies the name of a system-descriptor area.

The **COUNT** field in the system-descriptor area corresponds to the number of dynamic parameters in the prepared statement. The value of **COUNT** must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement. You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

X/O

A system-descriptor area conforms to the X/Open standards. ♦

The following ESQL/C example shows how to associate values from a system-descriptor area:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL put selcurs using sql descriptor 'desc1';
```

Specifying an sqlda Structure

Use the **DESCRIPTOR** option to introduce the name of a pointer to an **sqlda** structure. The following ESQL/C example shows how to associate values from an **sqlda** structure:

```
EXEC SQL put selcurs using descriptor pointer2;
```

IDS

Inserting into a Collection Cursor

A collection cursor allows you to access the individual elements of a collection variable. To declare a collection cursor, use the DECLARE statement and include the Collection-Derived-Table segment in the INSERT statement that you associate with the cursor. Once you open the collection cursor with the OPEN statement, the cursor can put elements in the collection variable.

To put elements, one at a time, into the insert cursor, use the PUT statement and the FROM clause. The PUT statement identifies the collection cursor that is associated with the collection variable. The FROM clause identifies the element value to be inserted into the cursor. The data type of any host variable in the FROM clause must match the element type of the collection.

Important: *The collection variable stores the elements of the collection. However, it has no intrinsic connection with a database column. Once the collection variable contains the correct elements, you must then save the variable into the actual collection column with the INSERT or UPDATE statement.*

Suppose you have a table called **children** with the following schema:

```
CREATE TABLE children
(
   age      SMALLINT,
   name     VARCHAR(30),
   fav_colors SET(VARCHAR(20)),
)
```

The following ESQL/C program fragment shows how to use an insert cursor to put elements into a collection variable called **child_colors**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection child_colors;
   char *favorites[]
   (
      "blue",
      "purple",
      "green",
      "white",
      "gold",
      0
   );
   int a = 0;
   char child_name[21];
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :child_colors;

/* Get structure of fav_colors column for untyped
 * child_colors collection variable */
EXEC SQL select fav_colors into :child_colors
   from children
   where name = :child_name;
/* Declare insert cursor for child_colors collection
 * variable and open this cursor */
EXEC SQL declare colors_curs cursor for
   insert into table(:child_colors)
   values (?);
EXEC SQL open colors_curs;
/* Use PUT to gather the favorite-color values
 * into a cursor */
while (fav_colors[a])
{
   EXEC SQL put colors_curs from :favorites[:a];
   a++
   ...
}
/* Flush cursor contents to collection variable */
EXEC SQL flush colors_curs;
EXEC SQL update children set fav_colors = :child_colors;

EXEC SQL close colors_curs;
EXEC SQL deallocate collection :child_colors;
```

After the FLUSH statement executes, the collection variable, **child_colors**, contains the elements {`"blue", "purple", "green", "white", "gold"`}. The UPDATE statement at the end of this program fragment saves the new collection into the **fav_colors** column of the database. Without this UPDATE statement, the new collection would not be added to the collection column.

Writing Buffered Rows

To open an insert cursor, the OPEN statement creates an insert buffer. The PUT statement puts a row into this insert buffer. The buffered rows are inserted into the database table as a block only when necessary; this process is called *flushing the buffer*. The buffer is flushed after any of the following events:

- Buffer is too full to hold the new row at the start of a PUT statement.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- An OPEN statement specifies an already open cursor, closing it before reopening it. (This implicit CLOSE statement flushes the buffer.)
- A COMMIT WORK statement executes.
- The buffer contains BYTE or TEXT data (flushed after a single PUT statement).

If the program terminates without closing an insert cursor, the buffer remains unflushed. Rows that were inserted into the buffer since the last flush are lost. Do not rely on the end of the program to close the cursor and flush the buffer.

Error Checking

The **sqlca** structure contains information on the success of each PUT statement as well as information that lets you count the rows that were inserted. The result of each PUT statement is contained in the following fields of the **sqlca**: **sqlca.sqlcode**, **SQLCODE**, and **sqlca.sqlerrd[2]**.

Data buffering with an insert cursor means that errors are not discovered until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, buffered rows that were not inserted before the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set to 0 if no error occurs; otherwise, it is set to an error code. The third element of the **sqlerrd** array is set to the number of rows that were successfully inserted into the database:

- If any row is put into the insert buffer, but *not* written to the database, **SQLCODE** and **sqlerrd** are set to 0 (**SQLCODE** because no error occurred, and **sqlerrd** because no rows were inserted).
- If a block of buffered rows is written to the database during the execution of a PUT statement, **SQLCODE** is set to 0 and **sqlerrd** is set to the number of rows that was successfully inserted into the database.
- If an error occurs while the buffered rows are written to the database, **SQLCODE** indicates the error, and **sqlerrd** contains the number of successfully inserted rows. (The uninserted rows are discarded from the buffer.)

Tip: *When you encounter an **SQLCODE** error, a **SQLSTATE** error value also exists. See the GET DIAGNOSTICS statement for details of how to obtain the message text.*

To count the number of pending and inserted rows in the database

1. Prepare two integer variables (for example, **total** and **pending)**.
2. When the cursor is opened, set both variables to 0.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a PUT or FLUSH statement executes or the cursor closes, subtract the third field of the **SQLERRD** array from **pending**.

At any time, (**total** - **pending)** represents the number of rows actually inserted. If no statements fail, **pending** contains zero after the cursor is closed. If an error occurs during a PUT, FLUSH, or CLOSE statement, the value that remains in **pending** is the number of uninserted (discarded) rows.

Related Information

Related statements: ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, FLUSH, DECLARE, GET DESCRIPTOR, OPEN, PREPARE, and SET DESCRIPTOR

For a task-oriented discussion of the PUT statement, see the *IBM Informix Guide to SQL: Tutorial*.

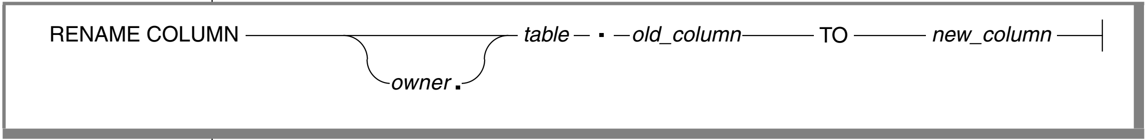
For more information about error checking, the system-descriptor area, and the **sqlda** structure, see the *IBM Informix ESQL/C Programmer's Manual*.

+

RENAME COLUMN

Use the RENAME COLUMN statement to change the name of a column.

Syntax



Element	Purpose	Restrictions	Syntax
new_column	New name to replace *old_column*	Must not match any other column name in *table*. See also "How Triggers Are Affected."	Identifier, p. 4-189
old_column	Column to rename	Must exist within *table*.	Identifier, p. 4-189
owner	Owner of the table	Must be the owner of the table.	Owner, p. 4-234
table	Table that contains *old_column*	Must exist in the current database.	Database Object Name, p. 4-46

Usage

You can rename a column of a table if any of the following conditions are true:

- You own the table or have Alter privilege on the table..
- You have the DBA privilege on the database.

XPS

You cannot rename the columns of a fragmented table if the table is fragmented by range. For more information, see "RANGE Method Clause" on page 2-244. ♦

How Views and Check Constraints Are Affected

If you rename a column that appears in a view, the text of the view definition in the **sysviews** system catalog table is updated to reflect the new column name. If you rename a column that appears in a check constraint, the text of the check constraint in the **syschecks** system catalog table is updated to reflect the new column name.

How Triggers Are Affected

If you rename a column that appears within a trigger, it is replaced with the new name only in the following instances:

- When it appears as part of a correlation name inside the FOR EACH ROW action clause of a trigger
- When it appears as part of a correlation name in the INTO clause of an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement
- When it appears as a triggering column in the UPDATE clause

When the trigger executes, if the database server encounters a column name that no longer exists in the table, an error is returned.

Example of RENAME COLUMN

The following example assigns the new name of **c_num** to the **customer_num** column in the **customer** table:

```
RENAME COLUMN customer.customer_num TO c_num
```

Related Information

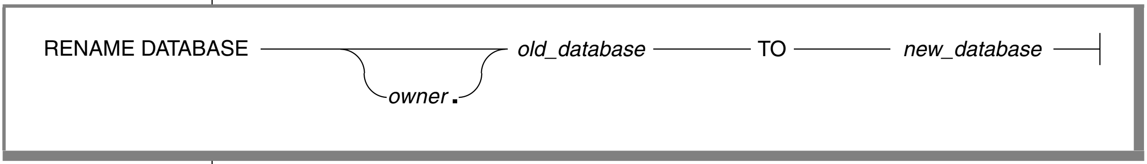
Related statements: ALTER TABLE, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, and RENAME TABLE

+

RENAME DATABASE

Use the RENAME DATABASE statement to change the name of a database.

Syntax



Element	Purpose	Restrictions	Syntax
new_database	New name for the database	Must be unique among database names of current database server and must not be opened by any users when RENAME DATABASE is issued.	Database Name, p. 4-44
old_database	Name that *new_database* replaces	Must exist on current database server, but it cannot be the name of the current database.	Database Name, p. 4-44
owner	Owner of database	Must be the owner of the database.	Owner, p. 4-234

Usage

You can rename a database if either of the following is true:

- You created the database.
- You have the DBA privilege on the database.

You can only rename local databases. You can rename a local database from inside an SPL routine.

Related Information

Related statement: CREATE DATABASE

IDS

Java

For information on how to update the three-part names of JAR files after you rename the database, see the *J/Foundation Developer's Guide*. ♦

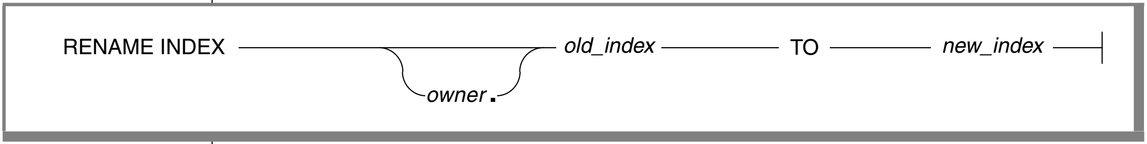
+

IDS

RENAME INDEX

Use the RENAME INDEX statement to change the name of an existing index.

Syntax



Element	Purpose	Restrictions	Syntax
new_index	New name for the index	Name must be unique to the database (or to the session, if the index is on a temporary table).	Identifier, p. 4-189
old_index	Index name that *new_index* replaces	Must exist, but it cannot be any of the following: An index on a system catalog table A system-generated constraint index A Virtual-Index Interface (VII)	Identifier, p. 4-189
owner	Owner of imdex	Must be the owner of the index.	Owner, p. 4-234

Usage

You can rename an index if you are the owner of the index or have the DBA privilege on the database.

When you rename an index, the database server changes the index name in the following system catalog tables: **sysindexes**, **sysconstraints**, **sysobjstate**, and **sysfragments**. When you rename an index on a temporary table, however, no system catalog tables are updated.

SPL routines that use the renamed index are reoptimized on their next use after the index is renamed.

Related Information

Related statements: ALTER INDEX, CREATE INDEX, and DROP INDEX

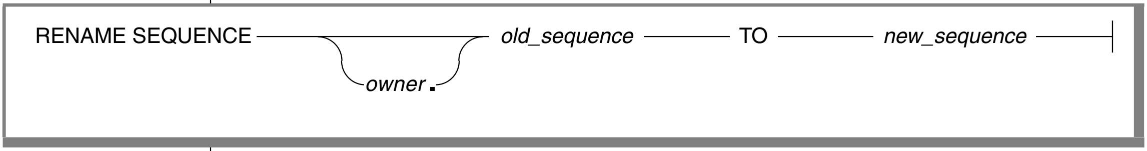
For a discussion of SPL-routine reoptimization, see your *Performance Guide*.

IDS

RENAME SEQUENCE

Use the RENAME SEQUENCE statement to change the name of a sequence.

Syntax



Element	**Purpose**	**Restrictions**	**Syntax**
new_sequence	New name that you declare here for an existing sequence	Must be unique among sequences, tables, views, and synonyms in the database	Identifier, p. 4-189
old_sequence	Current name of a sequence	Must exist in the current database	Identifier, p. 4-189
owner	Owner of the sequence	Must be the owner of the sequence	Owner Name, p. 4-234

Usage

To rename a sequence, you must be the owner of the sequence, or have the ALTER privilege on the sequence, or have the DBA privilege on the database.

You cannot use a synonym to specify the name of the sequence.

In a database that is not ANSI compliant, the name of *new_sequence* (or in an ANSI-compliant database, the combination of *owner*.*new_sequence*) must be unique among sequences, tables, views, and synonyms in the database.

Related Information

Related statements: ALTER SEQUENCE, CREATE SEQUENCE, DROP SEQUENCE, CREATE SYNONYM, DROP SYNONYM, GRANT, REVOKE, INSERT, UPDATE, and SELECT

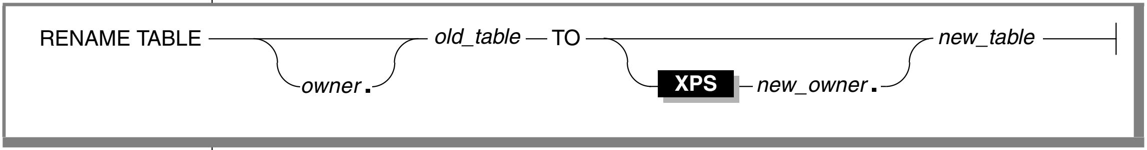
For information about generating values from a sequence, see "NEXTVAL and CURRVAL Operators" on page 4-102.

+

RENAME TABLE

Use the RENAME TABLE statement to change the name of a table.

Syntax



Element	Purpose	Restrictions	Syntax
new_table	New name for *old_table*	Cannot include an *owner***.** qualifier here.	Identifier, p. 4-189
old_table	Name that *new_table* replaces	Must be the name (not the synonym) of a table that exists in the current database.	Identifier, p. 4-189
owner	Current owner of the table	Must be the owner of the table.	Owner, p. 4-234
new_owner	The new owner of the table	Must have DBA privilege on the database (XPS)	Owner, p. 4-234

Usage

To rename a table, you must be the owner of the table, or have the ALTER privilege on the table, or have the DBA privilege on the database.

An error occurs if *old_table* is a synonym, rather than the name of a table.

IDS

You cannot change the table *owner* by renaming the table. An error occurs if you try to specify an *owner***.** qualifier for the new name of the table. ♦

XPS

A user with DBA privilege on the database can change the owner of a table, if the table is local. Both the table name and owner can be changed using one command.

The following example uses the RENAME TABLE statement to change the owner of a table:

```
RENAME TABLE tro.customer TO mike.customer
```

When the table owner is changed, you must specify both the old owner and new owner.

Important: *When the owner of a table is changed, the existing privileges granted by the original owner are retained.* ♦

ANSI

In an ANSI-compliant database, if you are not the owner of *old_table*, you must specify *owner***.***old_table* as the old name of the table. ♦

XPS

You cannot rename a table that contains a dependent GK index. ♦

The renamed table remains in the current database. You cannot use the RENAME TABLE statement to move a table from the current database to another database, nor to rename a table that resides in another database.

If *old_table* is referenced by a view in the current database, the view definition is updated in the **sysviews** system catalog table to reflect the new table name. For further information on the **sysviews** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

If *old_table* is a triggerring table, the database server takes these actions:

- Replaces the name of the table in the trigger definition but does *not* replace the table name where it appears inside any triggered actions
- Returns an error if the new table name is the same as a correlation name in the REFERENCING clause of the trigger definition

When the trigger executes, the database server returns an error if it encounters a table name for which no table exists.

The following example reorganizes the **items** table to move the **quantity** column from the fifth position to the third position by the following steps:

1. Create a new table, **new_table**, that contains the column **quantity** in the third position.
2. Fill the table with data from the current **items** table.
3. Drop the old **items** table.
4. Rename **new_table** with the name **items**.

The following example uses the RENAME TABLE statement as the last step:

```
CREATE TABLE new_table
   (
   item_num SMALLINT,
   order_numINTEGER,
   quantity SMALLINT,
   stock_numSMALLINT,
   manu_codeCHAR(3),
   total_priceMONEY(8)
   );
INSERT INTO new_table
   SELECT item_num, order_num, quantity, stock_num,
         manu_code, total_price FROM items;
DROP TABLE items;
RENAME TABLE new_table TO items;
```

Related Information

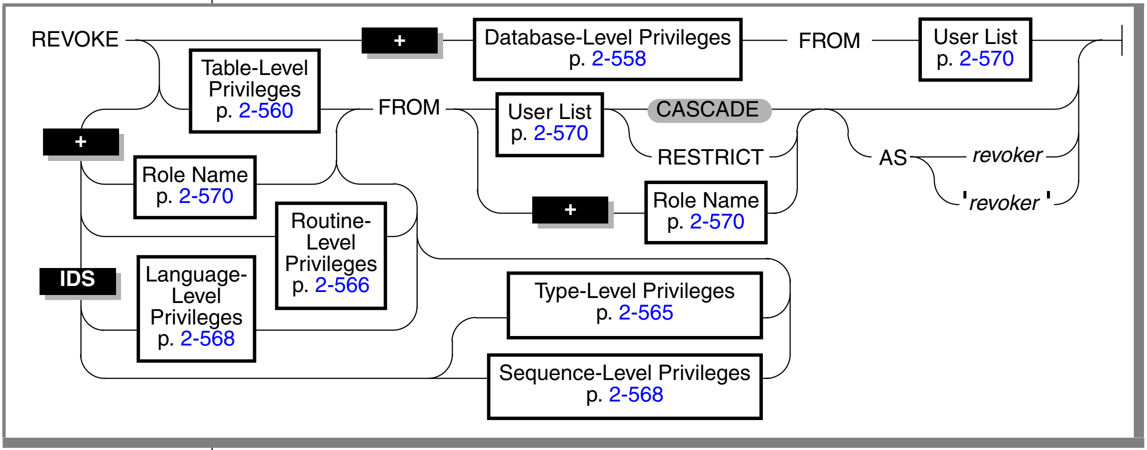
Related statements: ALTER TABLE, CREATE TABLE, DROP TABLE, and RENAME COLUMN

+

REVOKE

Use the REVOKE statement to cancel access privileges for specified users or for a specified role, or to cancel a specified role. (To cancel privileges on one or more fragments of a table that has been fragmented by expression, see "REVOKE FRAGMENT" on page 2-575.)

Syntax


REVOKE
+
Database-Level Privileges p. 2-558
FROM
User List p. 2-570
Table-Level Privileges p. 2-560
FROM
User List p. 2-570
CASCADE
RESTRICT
AS
revoker
'revoker'
+
Role Name p. 2-570
+
Role Name p. 2-570
Routine-Level Privileges p. 2-566
IDS
Language-Level Privileges p. 2-568
Type-Level Privileges p. 2-565
Sequence-Level Privileges p. 2-568

Element	Purpose	Restrictions	Syntax
revoker	Authorization identifier of the grantor of the privilege(s) to be revoked	Must be the grantor of the specified privileges.	Owner Name, p. 4-234.

Usage

You can revoke privileges if any of the following conditions is true for the privileges that you are attempting to revoke on some database object:

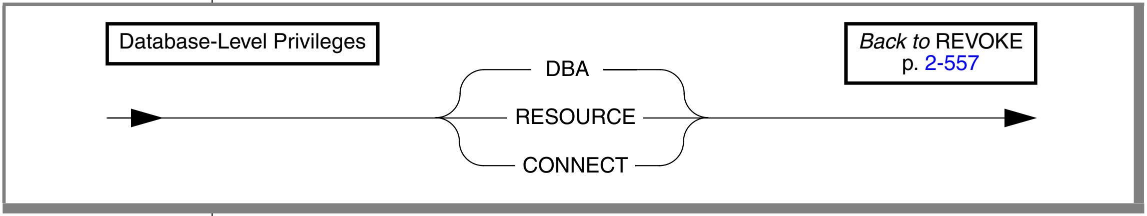
- You granted them and did not designate another user as grantor.
- The GRANT statement named you as grantor.
- You own an object on which PUBLIC has privileges by default.
- You have database-level DBA privileges and you specify in the AS clause the name of a user who is grantor of the privilege.

The REVOKE statement can cancel any of the following items for specific users or for a specified role:

- Privileges on a database
- Privileges on a table, synonym, view, or sequence object
- Privileges on a user-defined data type (UDT), a user-defined routine (UDR), or on the SPL language
- A role name.

You cannot revoke privileges from yourself. You cannot revoke *grantor* status from another user. To revoke a privilege that was granted to another user by the AS *grantor* clause of the GRANT statement, you must have the DBA privilege and you must use the AS clause to specify that user as *revoker*.

Database-Level Privileges



Three concentric layers of privileges, Connect, Resource, and DBA, authorize increasing power over database access and control. Only a user with the DBA privilege can grant or revoke database-level privileges.

Because of the hierarchical organization of the privileges (as outlined in the privilege definitions that are described later in this section), if you revoke either the Resource or the Connect privilege from a user with the DBA privilege, the statement has no effect. If you revoke the DBA privilege from a user who has the DBA privilege, the user retains the Connect privilege on the database. To deny database access to a user with the DBA or Resource privilege, you must first revoke the DBA or the Resource privilege and then revoke the Connect privilege in a separate REVOKE statement.

Similarly, if you revoke the Connect privilege from a user with the Resource privilege, the statement has no effect. If you revoke the Resource privilege from a user, the user retains the Connect privilege on the database.

Warning: *Although user **informix** and DBAs can modify most system catalog tables (only user **informix** can modify **systables**), It is strongly recommended that you not update, delete, or insert any rows in these tables. Modifying system catalog tables can destroy the integrity of the database. The use of the ALTER TABLE statement to modify the size of the next extent of system catalog tables is not supported.*

The following table lists the keyword for each database-level privilege.

Privilege	Purpose
DBA	Has all the capabilities of the Resource privilege and can perform the following additional operations: ■ Grant any database-level privilege, including the DBA privilege, to another user. ■ Grant any table-level privilege to another user or to a role. ■ Grant a role to a user or to another role. ■ Revoke a privilege whose grantor you specify as the *revoker* in the AS clause of the REVOKE statement. ■ Restrict the Execute privilege to DBAs when registering a UDR. ■ Execute the SET SESSION AUTHORIZATION statement. ■ Use the NEXT SIZE keywords to alter extent sizes in the system catalog tables. ■ Create any database object. ■ Create tables, views, and indexes, designating another user as owner of these objects. ■ Alter, drop, or rename database objects, regardless of who owns it. ■ Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement. ■ Execute DROP DATABASE and RENAME DATABASE statements. ■ Insert, delete, or update rows of any system catalog table except **systables.**

(1 of 2)

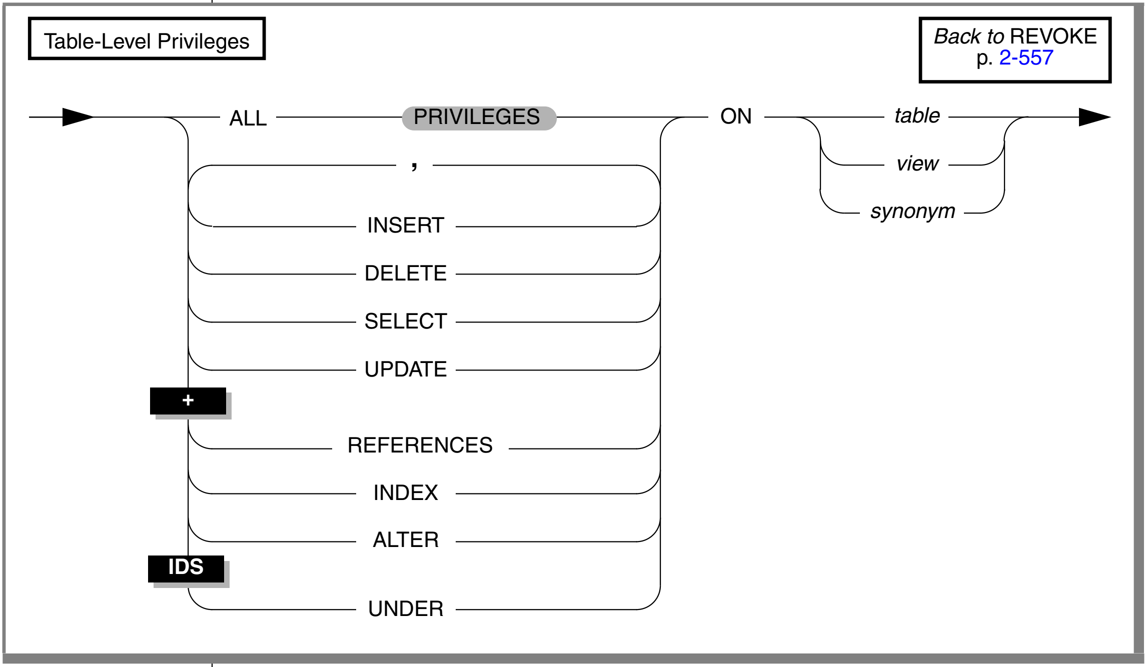
Privilege	Purpose
RESOURCE	Lets you extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following operations: ■ Create new tables. ■ Create new indexes. ■ Create new user-defined routines. ■ Create new data types.
CONNECT	If you have this privilege, you can query and modify data, and modify the database schema if you own the database object that you want to modify. A user holding the Connect privilege can perform the following operations: ■ Connect to the database with the CONNECT statement or another connection statement. ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided that the user has the necessary table-level privileges. ■ Create views, provided that the user has the Select privilege on the underlying tables. ■ Create synonyms. ■ Create temporary tables and create indexes on temporary tables. ■ Alter or drop a table or an index, if the user owns the table or index (or has the Alter, Index, or References privilege on the table). ■ Grant privileges on a table, if the user owns the table (or was given privileges on the table with the WITH GRANT OPTION keyword).

(2 of 2)

Table-Level Privileges

Select, Update, and References privileges can be granted on some columns of a table, view, or synonym, but are revoked for all columns. If Select privileges are revoked from a user for a table that is referenced in the SELECT statement defining a view that the same user owns, then that view is dropped, unless it also includes columns from tables in another database.

This is the syntax for specifying the table-level privileges to revoke:



Element	Purpose	Restrictions	Syntax
synonym	Synonym for the table or view on which privileges are revoked	Synonym and its table must exist.	Database Object Name, p. 4-46
table	Table on which privileges are revoked	Must exist.	Database Object Name, p. 4-46
view	View on which privileges are revoked	View must exist.	Database Object Name, p. 4-46

In one REVOKE statement, you can list one or more of the following keywords to specify the privileges you want to revoke from the same users.

Privilege	Purpose
INSERT	The holder can insert rows.
DELETE	The holder can delete rows.
SELECT	The holder can display data retrieved by a SELECT statement.

(1 of 2)

Privilege	Purpose
UPDATE	The holder can change column values.
INDEX	The holder can create permanent indexes. You must have the Resource privilege to take advantage of the Index privilege. (Any user with the Connect privilege can create indexes on temporary tables.)
ALTER	The holder can add or delete columns, modify column data types, add or delete constraints, change the locking mode of a table from PAGE to ROW, or add or drop a corresponding ROW type name for your table. The holder can also set the database object mode of indexes, constraints, and triggers; for details of how to change these modes, see "SET Database Object Mode" on page 2-652.
REFERENCES	The holder can reference columns in referential constraints. You must have the Resource privilege to take advantage of the References privilege. (You can add, however, a referential constraint during an ALTER TABLE statement. This action does not require that you have the Resource privilege on the database.) Revoke the References privilege to disallow cascading deletes.
UNDER (IDS only)	The holder can create subtables under a typed table.
ALL	This privilege provides all of the table privileges that are listed above. The PRIVILEGES keyword is optional.

(2 of 2)

See also "Table-Level Privileges" on page 2-463.

If a user receives the same privilege from two different grantors and one grantor revokes the privilege, the grantee still has the privilege until the second grantor also revokes the privilege. For example, if both you and a DBA grant the Update privilege on your table to **ted**, both you and the DBA must revoke the Update privilege to prevent **ted** from updating your table.

When to Use REVOKE Before GRANT

You can use combinations of REVOKE and GRANT to replace PUBLIC with specific users as grantees, and to remove table-level privileges on some columns.

Replacing PUBLIC with Specified Users

If a table owner grants a privilege to PUBLIC, the owner cannot revoke the same privilege from any specific user. For example, assume PUBLIC has default Select privileges on your **customer** table. Suppose that you issue the following statement in an attempt to exclude **ted** from accessing your table:

```
REVOKE ALL ON customer FROM ted
```

This statement results in ISAM error message 111, `No record found`, because the system catalog tables (**syscolauth** or **systabauth**) contain no table-level privilege entry for a user named **ted**. The REVOKE does not prevent **ted** from having all the table-level privileges given to PUBLIC on the **customer** table.

To restrict table-level privileges, first revoke the privileges with the PUBLIC keyword, then re-grant them to some appropriate user list. The following statements revoke the Index and Alter privileges from all users for the **customer** table, and then grant these privileges specifically to user **mary**:

```
REVOKE INDEX, ALTER ON customer FROM PUBLIC
GRANT INDEX, ALTER ON customer TO mary
```

Restricting Access to Specific Columns

The REVOKE statement has no syntax for revoking privileges on particular column names. When you revoke the Select, Update, or References privilege from a user, you revoke the privilege for all columns in the table. If you want a user to have some access to some, but not all the columns previously granted, issue a new GRANT statement to restore the appropriate privileges.

The next example cancels Select privileges for PUBLIC on certain columns:

```
REVOKE SELECT ON customer FROM PUBLIC
GRANT SELECT (fname, lname, company, city)
   ON customer TO PUBLIC
```

In the next example, **mary** first receives the ability to reference four columns in **customer**, then the table owner restricts references to two columns:

```
GRANT REFERENCES (fname, lname, company, city) ON
   customer TO mary
REVOKE REFERENCES ON customer FROM mary
GRANT REFERENCES (company, city)
   ON customer TO mary
```

Effect of the ALL Keyword

The ALL keyword revokes all table-level privileges. If any or all of the table-level privileges do not exist for the revokee, REVOKE with the ALL keyword executes successfully but returns the following SQLSTATE code:

```
01006--Privilege not revoked
```

For example, assume that user **hal** has the Select and Insert privileges on the **customer** table. User **jocelyn** wants to revoke all seven table-level privileges from user **hal**. So user **jocelyn** issues the following REVOKE statement:

```
REVOKE ALL ON customer FROM hal
```

This statement executes successfully but returns SQLSTATE code 01006. The SQLSTATE warning is returned with a successful statement, as follows:

- The statement succeeds in revoking the Select and Insert privileges from user **hal** because user **hal** had those privileges.
- SQLSTATE code 01006 is returned because the other privileges implied by the ALL keyword did not exist for user **hal**; therefore, these privileges were not revoked.

Tip: *The ALL keyword instructs the database server to revoke everything possible, including nothing. If the user from whom privileges are revoked has no privileges on the table, the REVOKE ALL statement still succeeds, because it revokes everything possible from the user (in this case, no privileges at all).*

Effect of ALL Keyword on UNDER Privilege

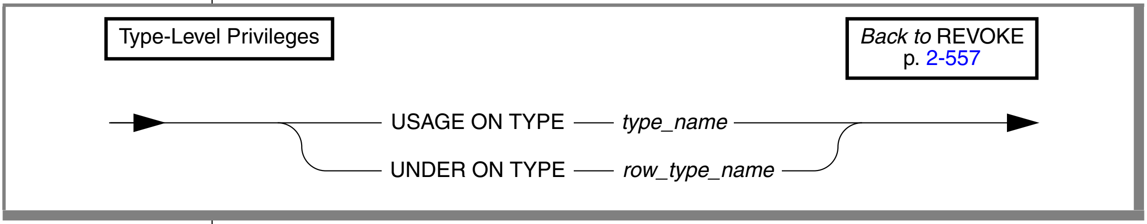
If you revoke ALL privileges on a typed table, the Under privilege is included in the privileges that are revoked. If you revoke ALL privileges on a table that is not based on a row type, the Under privilege is not included in the privileges that are revoked. (The Under privilege cannot be granted on a table that is not a typed table.)

IDS

Type-Level Privileges

You can revoke two privileges on data types:

- The Usage privilege on a user-defined data type
- The Under privilege on a named-row type



Element	Purpose	Restrictions	Syntax
row_type_name	Named-row type for which to revoke Under privilege	Must exist	Data Type, p. 4-49
type_name	User-defined type for which to revoke Usage privilege	Must exist	Data Type, p. 4-49

Usage Privilege

Any user can reference a built-in data type in an SQL statement, but not a DISTINCT data type based on a built-in data type. The creator of a user-defined data type or a DBA must explicitly grant the Usage privilege on that new data type, including a DISTINCT data type based on a built-in data type.

REVOKE with the USAGE ON TYPE keywords removes the Usage privilege that you granted earlier to another user or to a role.

Under Privilege

You own a named-row type that you create. If you want other users to be able to create subtypes under this named-row type, you must grant these users the Under privilege on your named-row type. If you later want to remove the ability of these users to create subtypes under the named-row type, you must revoke the Under privilege from these users. A REVOKE statement with the UNDER ON TYPE keywords removes the Under privilege that you granted earlier to these users.

For example, suppose that you created a row type named **rtype1**:

```
CREATE ROW TYPE rtype1 (cola INT, colb INT)
```

If you want another user named **kathy** to be able to create a subtype under this named-row type, you must grant the Under privilege on this named-row type to user **kathy**:

```
GRANT UNDER on rtype1 to kathy
```

Now user **kathy** can create another row type under the **rtype1** row type even though **kathy** is not the owner of the **rtype1** row type:

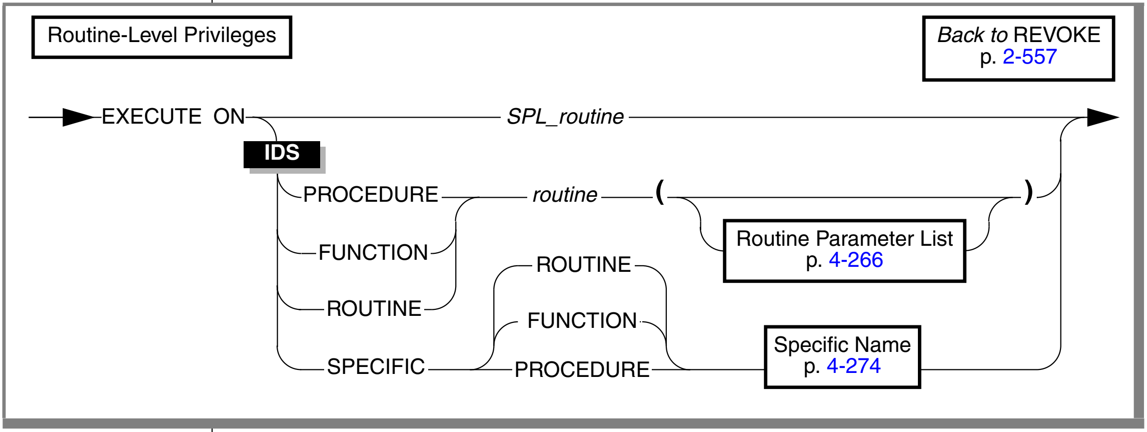
```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1
```

If you later want to remove the ability of user **kathy** to create subtypes under the **rtype1** row type, enter the following statement:

```
REVOKE UNDER on rtype1 FROM kathy
```

Routine-Level Privileges

If you revoke the Execute privilege on a UDR from a user, that user can no longer execute that UDR in any way. For details of how a user can execute a UDR, see "Routine-Level Privileges" on page 2-470.



Element	Purpose	Restrictions	Syntax
routine	A user-defined routine	Must exist.	Database Object Name, p. 4-46
SPL_routine	An SPL routine	Must be unique in the database.	Database Object Name, p. 4-46

ANSI

In an ANSI-compliant database, the *owner* name must qualify the *routine* name. ♦

When you create a UDR under any of the following circumstances, you must explicitly grant the Execute privilege before you can revoke it:

ANSI

- You create a UDR in an ANSI-compliant database. ♦
- You have DBA-level privileges and use the DBA keyword with CREATE to restrict the Execute privilege to users with the DBA database-level privilege.
- The **NODEFDAC** environment variable is set to `yes` to prevent PUBLIC from receiving any privileges that are not explicitly granted.

IDS

Any negator function for which you grant the Execute privilege requires a separate, explicit REVOKE statement. ♦

When you create a UDR without any of the preceding conditions in effect, PUBLIC can execute your UDR without a GRANT statement. To limit who executes your UDR, revoke the privilege using the keywords FROM PUBLIC and then grant it to a user list (see "User List" on page 2-570) or role (see "Role Name" on page 2-570).

IDS

If two or more UDRs have the same name, use the appropriate keyword from the following list to specify which of those UDRs a user can no longer execute.

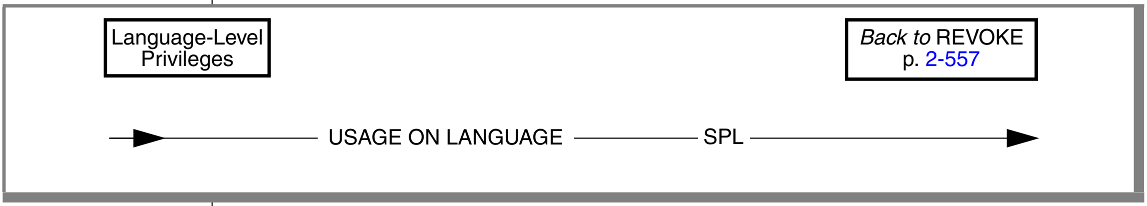
Privilege	**Purpose**
SPECIFIC	Prevents a user from executing the UDR identified by *specific name*
FUNCTION	Prevents execution of any function with the specified *routine name* (and parameter types that match *routine parameter list*, if supplied)
PROCEDURE	Prevents execution of any procedure with the specified *routine name* (and parameter types that match *routine parameter list*, if supplied)
ROUTINE	Prevents execution of both functions and procedures with the specified *routine name* (and parameter types that match *routine parameter list*, if supplied)

♦

IDS

Language-Level Privileges

A user must have the Usage privilege on a language to register a UDR that is written in that language.



When a user executes a CREATE FUNCTION or CREATE PROCEDURE statement to register a UDR, the database server verifies that the user has the Usage privilege on the language in which the UDR is written. If the user does not have the Usage privilege, the statement fails. (In this release of Dynamic Server, the C language and the Java language do not require Usage privilege.)

If you want to revoke the Usage privilege on the SPL language from a user or role, issue a REVOKE statement that includes USAGE ON LANGUAGE SPL keywords. The effect of issuing this statement is that the user or role can no longer register UDRs that are written in the specified language. For example, if you revoke the default Usage privilege in SPL from PUBLIC, the ability to create SPL routines is taken away from all users:

```
REVOKE USAGE ON LANGUAGE SPL FROM PUBLIC
```

You can issue a GRANT USAGE ON LANGUAGE statement to restore Usage privileges in SPL to a restricted group, such as the role named **developers**:

```
GRANT USAGE ON LANGUAGE SPL TO developers
```

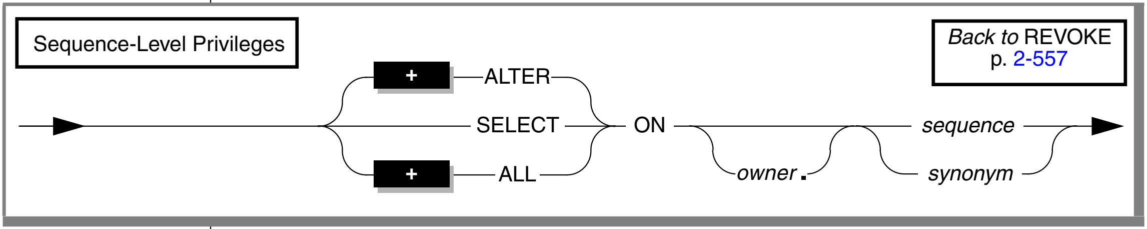
IDS

Sequence-Level Privileges

Although a sequence is logically a table, only a subset of the table privileges (as described in "Table-Level Privileges" on page 2-463) can be granted or revoked on a sequence. You can revoke either or both of the following privileges on a sequence object:

- Select privilege
- Alter privilege

Use this syntax to specify the privileges to revoke on a sequence object:



Element	Purpose	Restrictions	Syntax
owner	Owner of the sequence or of its synonym	Must be the owner	Owner Name,p. 4-234
sequence	Sequence on which to revoke privileges	Must exist	Identifier, p. 4-189
synonym	Synonym for the sequence object on which to revoke privileges	Must exist	Identifier, p. 4-189

The sequence must reside in the current database. (You can qualify the *sequence* or *synonym* identifier with a valid *owner* name, but the name of a remote *database* (or *database@server*) is not valid as a qualifier.)

Alter Privilege

You can revoke the Alter privilege on a sequence from another user or from a role. The Alter privilege enables a specified user or role to modify the definition of a sequence with the ALTER SEQUENCE statement or to rename the sequence with the RENAME SEQUENCE statement.

Select Privilege

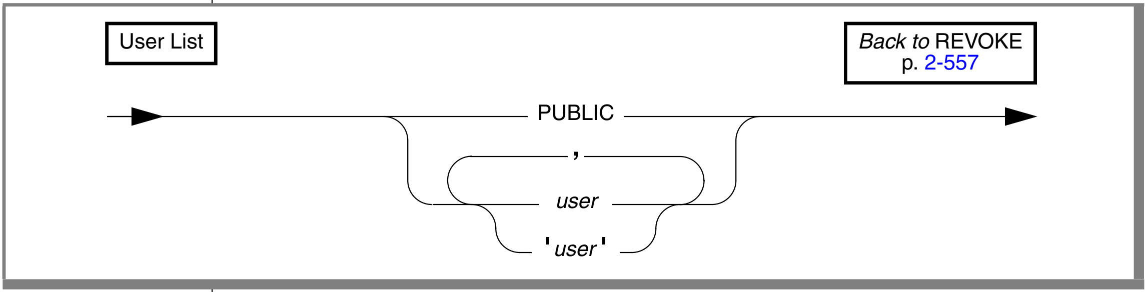
You can revoke the Select privilege on a sequence from another user or from a role. The Select privilege enables a specified user or role to use the *sequence*.CURRVAL and *sequence*.NEXTVAL in SQL and SPL statements to access and to increment the value of a sequence.

ALL Keyword

You can use the ALL keyword to revoke both Alter and Select privileges from another user or from a role.

User List

The user list specifies who loses the privileges that you are revoking. The user list can consist of the logins for a single user or multiple users, separated by commas. If you use the PUBLIC keyword as the user list, the REVOKE statement revokes privileges from all users.



Element	Purpose	Restrictions	Syntax
user	Login name of the user who is to lose the role or privilege	Put quotes around *user* to ensure that the name of the user is stored exactly as you type it. Use the single keyword PUBLIC for *user* to revoke a role or privilege from all authorized users.	Owner Name, p. 4-234

When the user list contains specific logins, you can combine the REVOKE statement with the GRANT statement to selectively secure tables, columns, UDRs, types, and so forth. For examples, see "When to Use REVOKE Before GRANT" on page 2-563.

Spell the user names in the list exactly as they were spelled in the GRANT statement. In a database that is not ANSI compliant, you can optionally use quotes around each user in the list.

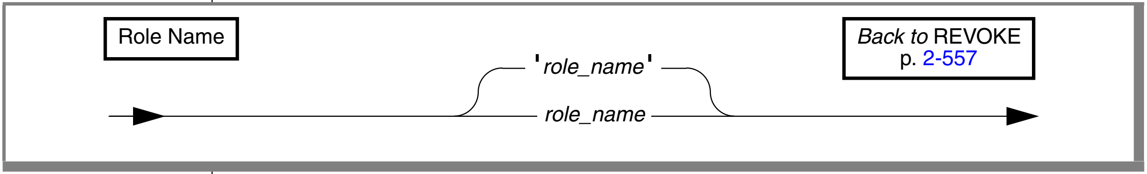
ANSI

In an ANSI-compliant database, if you do not use quotes to delimit *user*, the name of the user is stored in uppercase letters. ♦

Role Name

Only the DBA or a user who has been granted a role with the WITH GRANT OPTION can revoke a role or its privileges. Users cannot revoke roles from themselves.

When you revoke a role that was granted with the WITH GRANT OPTION, both the role and grant option are revoked. "Revoking Privileges Granted WITH GRANT OPTION" on page 2-571 explains revoking such a role.



Element	Purpose	Restrictions	Syntax
role_name	A role with one of these attributes: ■ Loses an existing privilege ■ Loses the use of another role ■ Is lost by a user or by another role	Must exist. If enclosed between quotation marks, *role_name* is case sensitive.	Identifier, p. 4-189

The following examples show the effects of REVOKE with *role_name*:

- Remove users or another role name from inclusion in the role:

```
REVOKE accounting FROM mary
REVOKE payroll FROM accounting
```

- Remove one or more privileges from a role:

```
REVOKE UPDATE ON employee FROM accounting
```

When you revoke table-level privileges from a role, you cannot use the RESTRICT or CASCADE clauses.

Revoking Privileges Granted WITH GRANT OPTION

If you revoke from *user* privileges that you granted using the WITH GRANT OPTION keywords, you sever the chain of privileges granted by that *user*. Thus, when you revoke privileges from users or a role, you also revoke the same privilege resulting from GRANT statements in the following contexts:

- Issued by your grantee
- Allowed because your grantee used the WITH GRANT OPTION clause
- Allowed because subsequent grantees granted the same privilege using the WITH GRANT OPTION clause

The following examples illustrate this situation. You, as the owner of the table **items**, issue the following statements to grant access to user **mary**:

```
REVOKE ALL ON items FROM PUBLIC
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION
```

User **mary** uses her new privilege to grant users **cathy** and **paul** access to the table:

```
GRANT SELECT, UPDATE ON items TO cathy
GRANT SELECT ON items TO paul
```

Later you revoke privileges on the **items** table to user **mary**:

```
REVOKE SELECT, UPDATE ON items FROM mary
```

This single statement effectively revokes all privileges on the **items** table from users **mary**, **cathy**, and **paul**.

The CASCADE keyword has the same effect as this default condition.

The AS Clause

Without the AS clause, the user who executes the REVOKE statement must be a grantor of the privilege that is being revoked. The DBA or the owner of the object can use the AS clause to specify another user (who must be the grantor of the privilege) as the revoker of the privileges. The AS clause provides the only mechanism by which privileges can be revoked on a database object whose *owner* is an authorization identifier, such as **informix**, that is not a valid user account known to the operating system.

Effect of CASCADE Keyword on UNDER Privileges

If you revoke the Under privilege on a typed table with the CASCADE option, the Under privilege is removed from the specified user, and any subtables created under the typed table by that user are dropped from the database.

If you revoke the Under privilege on a named ROW type with the CASCADE option when that data type is in use, the REVOKE fails. This exception to the normal behavior of the CASCADE option occurs because the database server supports the DROP ROW TYPE statement with the RESTRICT keyword only.

For example, assume that user **jeff** creates a ROW type named **rtype1** and grants the Under privilege on that ROW type to user **mary**. User **mary** now creates a ROW type named **rtype2** under ROW type **rtype1** and grants the Under privilege on ROW type **rtype2** to user **andy.** Then user **andy** creates a ROW type named **rtype3** under row type **rtype2**.

If user **jeff** now tries to revoke the Under privilege on ROW type **rtype1** from user **mary** with the CASCADE option, the REVOKE statement fails, because ROW type **rtype2** is still in use by ROW type **rtype3**.

Controlling the Scope of REVOKE with the RESTRICT Option

The RESTRICT keyword causes the REVOKE statement to fail when any of the following dependencies exist:

- A view depends on a Select privilege that you attempt to revoke.
- A foreign-key constraint depends on a References privilege that you attempt to revoke.
- You attempt to revoke a privilege from a user who subsequently granted this privilege to another user or users.

A REVOKE statement does not fail if it pertains to a user who has the right to grant the privilege to any other user but does not exercise that right.

For example, assume that user **clara** uses the WITH GRANT OPTION clause to grant the Select privilege on the **customer** table to user **ted**. Further assume that user **ted**, in turn, grants the Select privilege on the **customer** table to user **tania**. The following REVOKE statement that **clara** issued fails because **ted** used his authority to grant the Select privilege:

```
REVOKE SELECT ON customer FROM ted RESTRICT
```

By contrast, if user **ted** does not grant the Select privilege to **tania** or any other user, the same REVOKE statement succeeds. Even if **ted** does grant the Select privilege to another user, either of the following statements succeeds:

```
REVOKE SELECT ON customer FROM ted CASCADE
REVOKE SELECT ON customer FROM ted
```

Effect of Uncommitted Transactions

When REVOKE is executed, an exclusive row lock is placed on the entry in the **systables** system catalog table for the table from which privileges were revoked. The lock is not released until the transaction that contains the REVOKE statement is complete. When another transaction attempts to prepare a SELECT statement against this table, the transaction fails because the entry for this table in **systables** is exclusively locked. The attempt to prepare the SELECT statement will not succeed until the first transaction was committed.

Related Information

Related Statements: GRANT, GRANT FRAGMENT, and REVOKE FRAGMENT

For information about roles, see the following statements: CREATE ROLE, DROP ROLE, and SET ROLE.

For a discussion of privileges, see the *IBM Informix Database Design and Implementation Guide*.

For a discussion of how to embed GRANT and REVOKE statements in programs, see the *IBM Informix Guide to SQL: Tutorial*.

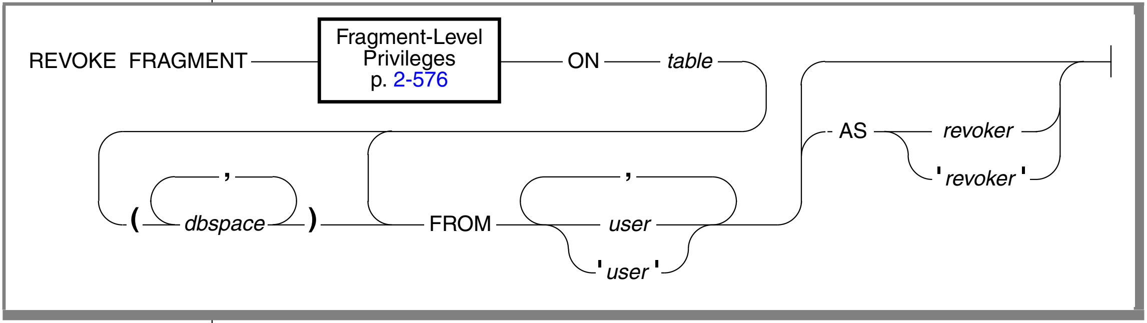
+

IDS

REVOKE FRAGMENT

Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete fragment-level privileges that were granted to users on individual fragments of a fragmented table.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that stores the fragments The default is all fragments of *table* on which *user* holds fragment-level privileges.	Must exist and must store fragment(s) of *table*.	Identifier, p. 4-189
revoker	The user (who is not executing the statement) who is grantor of the privileges to be revoked	Must be grantor of the privileges on the fragment(s).	Owner Name, p. 4-234.
table	Table whose fragment privileges are to be revoked	Must exist and must be fragmented by expression.	Database Object Name, p. 4-46
user	Users from whom the specified privileges are to be revoked	Must be a valid authorization identifier.	Owner Name, p. 4-234.

Usage

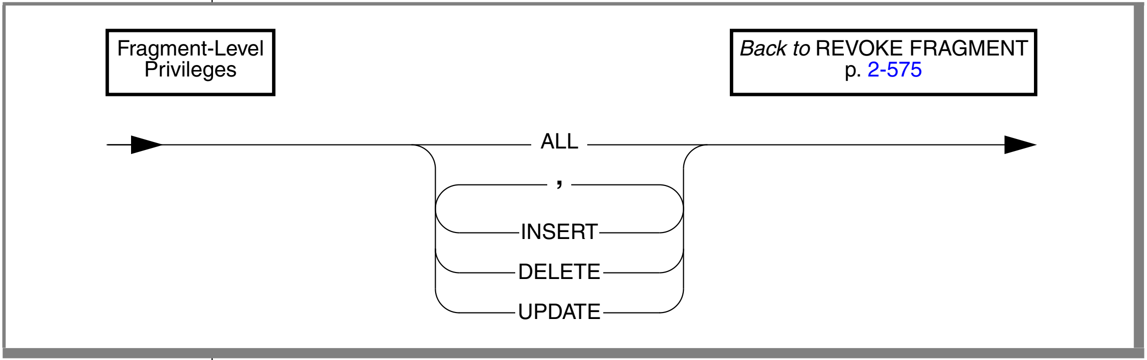
Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete privilege on one or more fragments of a fragmented table from one or more users. This can also be used by the DBA to revoke privileges on a fragment whose owner is another user.

The REVOKE FRAGMENT statement is only valid for tables that are fragmented according to an expression-based distribution scheme. For an explanation of an expression-based distribution scheme, see "Expression Distribution Scheme" on page 2-25.

You can specify one fragment or a comma-separated list of fragments in the REVOKE FRAGMENT statement. To specify a fragment, specify the name of the *dbspace* in which the fragment resides. If present, the *dbspace* list must be enclosed between a pair of parentheses that follow the ON *table* clause.

If you do not specify any *dbspace*, the specified users lose the specified privileges on all fragments for which the users currently have those privileges.

Fragment-Level Privileges



You can revoke fragment-level privileges individually or in combination. List the keywords that correspond to the privileges that you are revoking from *user*. The following table defines each of the fragment-level privileges.

Privilege	Purpose
ALL	Provides insert, delete, and update privileges on a fragment
INSERT	Lets you insert rows in the fragment
DELETE	Lets you delete rows in the fragment
UPDATE	Lets you update rows in the fragment and name any column of the table in an UPDATE statement

If you specify the ALL keyword in a REVOKE FRAGMENT statement, the specified users lose all fragment-level privileges that they currently possess on the specified fragments.

For example, assume that a user currently has the Update privilege on one fragment of a table. If you use the ALL keyword to revoke all current privileges on this fragment from this user, the user loses the Update privilege that he or she had on this fragment.

For the distinction between fragment-level and table-level privileges, see the sections "Definition of Fragment-Level Authorization" and "Role of Fragment-Level Authority in Command Validation" on page 2-482.

The AS Clause

Without the AS clause, the user who executes the REVOKE statement must be a grantor of the privilege that is being revoked. The DBA or the owner of the fragment can use the AS clause to specify another user (who must be the grantor of the privilege) as the revoker of privileges on a fragment.

The AS clause provides the only mechanism by which privileges can be revoked on a fragment whose *owner* is an authorization identifier that is not a valid user account known to the operating system.

Examples of the REVOKE FRAGMENT Statement

Examples that follow are based on the **customer** table. They all assume that the **customer** table is fragmented by expression into three fragments that reside in the dbspaces that are named **dbsp1**, **dbsp2**, and **dbsp3**.

Revoking One Privilege

The following statement revokes the Update privilege on the fragment of the **customer** table in **dbsp1** from user **ed**:

```
REVOKE FRAGMENT UPDATE ON customer (dbsp1) FROM ed
```

Revoking More Than One Privilege

The following statement revokes the Update and Insert privileges on the fragment of the **customer** table in **dbsp1** from user **susan**:

```
REVOKE FRAGMENT UPDATE, INSERT ON customer (dbsp1) FROM susan
```

Revoking All Privileges

The following statement revokes all privileges currently granted to user **harry** on the fragment of the **customer** table in **dbsp1**:

```
REVOKE FRAGMENT ALL ON customer (dbsp1) FROM harry
```

Revoking Privileges on More Than One Fragment

The following statement revokes all privileges currently granted to user **millie** on the fragments of the **customer** table in **dbsp1** and **dbsp2**:

```
REVOKE FRAGMENT ALL ON customer (dbsp1, dbsp2) FROM millie
```

Revoking Privileges from More Than One User

The following statement revokes all privileges currently granted to users **jerome** and **hilda** on the fragment of the **customer** table in **dbsp3**:

```
REVOKE FRAGMENT ALL ON customer (dbsp3) FROM jerome, hilda
```

Revoking Privileges Without Specifying Fragments

The following statement revokes all current privileges from user **mel** on all fragments for which this user currently has privileges:

```
REVOKE FRAGMENT ALL ON customer FROM mel
```

Related Information

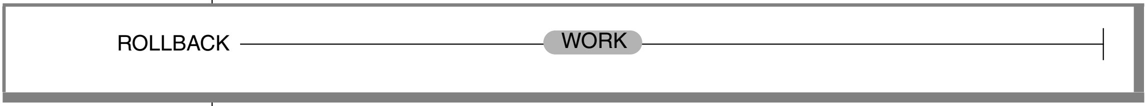
Related statements: GRANT FRAGMENT and REVOKE

For a discussion of fragment-level and table-level privileges, see the section "Fragment-Level Privileges" on page 2-481. See also the *IBM Informix Database Design and Implementation Guide*.

ROLLBACK WORK

Use the ROLLBACK WORK statement to cancel a transaction deliberately and undo any changes that occurred since the beginning of the transaction. The ROLLBACK WORK statement restores the database to the state that it was in before the transaction began.

Syntax



Usage

The ROLLBACK WORK statement is valid only in databases with transaction logging.

In a database that is not ANSI compliant, start a transaction with a BEGIN WORK statement. You can end a transaction with a COMMIT WORK statement or cancel the transaction with a ROLLBACK WORK statement. The ROLLBACK WORK statement restores the database to the state that existed before the transaction began.

Use ROLLBACK WORK only at the end of a multistatement operation.

The ROLLBACK WORK statement releases all row and table locks that the cancelled transaction holds. If you issue a ROLLBACK WORK statement when no transaction is pending, an error occurs.

ANSI

In an ANSI-compliant database, transactions are implicit. You do not need to mark the beginning of a transaction with a BEGIN WORK statement. You only need to mark the end of each transaction with a COMMIT WORK statement or cancel the transaction with a ROLLBACK WORK statement. If you issue a ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect. ♦

E/C

In ESQL/C, the ROLLBACK WORK statement closes all open cursors except those that are declared with hold. Hold cursors remain open after a transaction is committed or rolled back.

If you use the ROLLBACK WORK statement within an SPL routine that a WHENEVER statement calls, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This step prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

WORK Keyword

The WORK keyword is optional in a ROLLBACK WORK statement. The following two statements are equivalent:

```
ROLLBACK;

ROLLBACK WORK;
```

Related Information

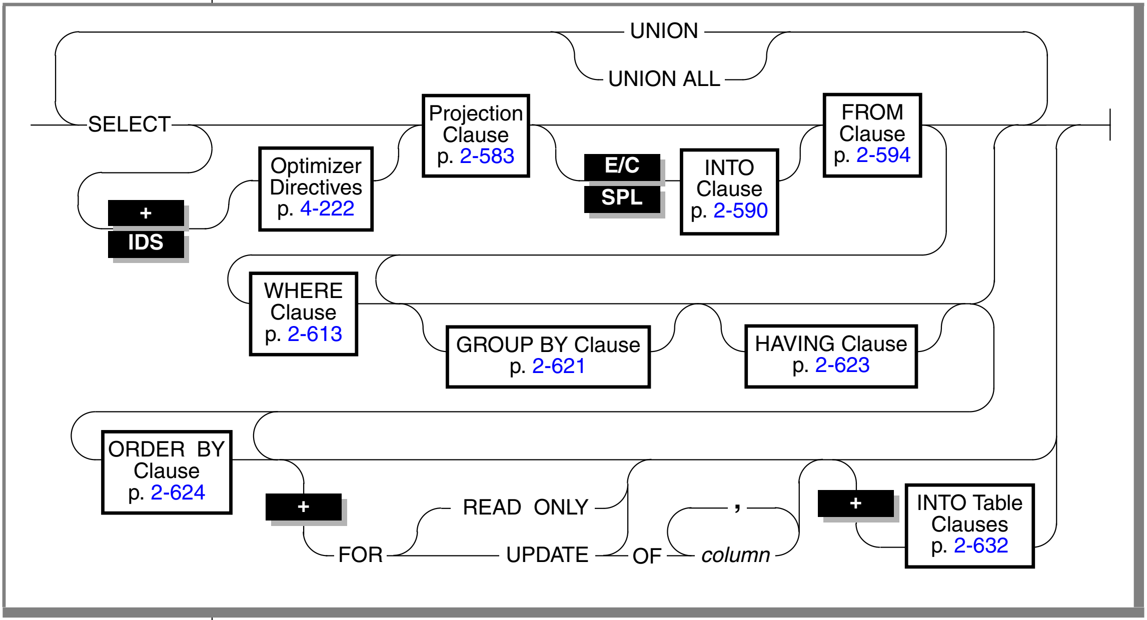
Related statements: BEGIN WORK and COMMIT WORK

For a discussion of transactions and ROLLBACK WORK, see the *IBM Informix Guide to SQL: Tutorial*.

SELECT

Use the SELECT statement to query a database or the contents of an SPL or ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
column	Name of a column that can be updated after a FETCH	Must be in a FROM clause *table*, but does not need to be in the select list of the Projection clause	Identifier, p. 4-189

Usage

The SELECT statement can return data from tables in the current database, or in another database of the current database server, or in a database of another database server. Only the SELECT keyword, the Projection clause, and the FROM clause are required specifications.

The SELECT statement can include various basic clauses, which are identified in the following list.

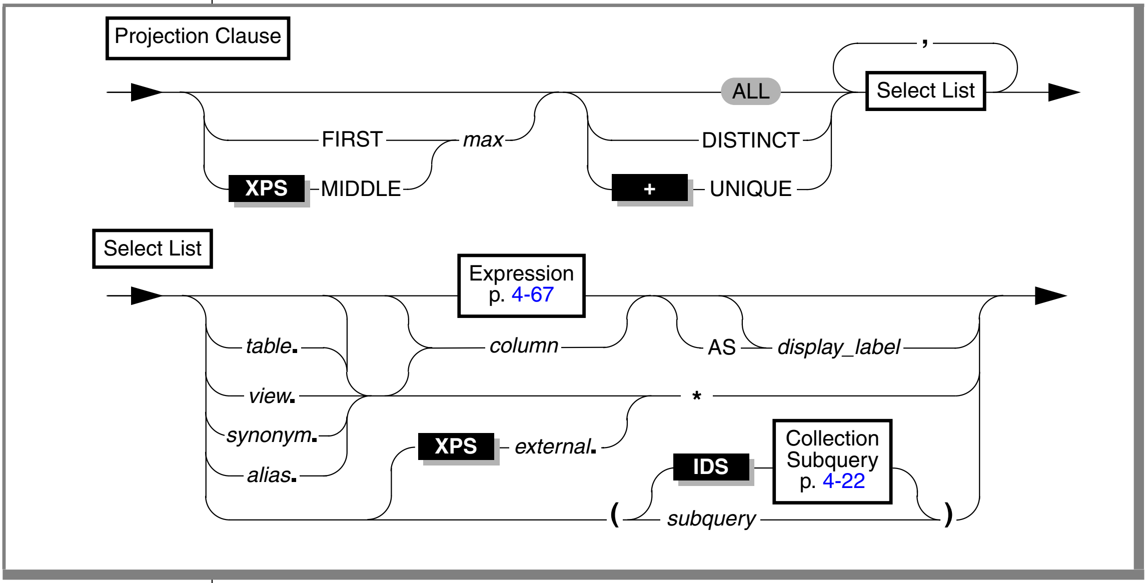
Clause	Page	Effect
Optimizer Directive	4-222	Specifies how the query should be implemented
Projection	2-583	Specifies a list of items to be read from the database
INTO	2-590	Specifies variables to receive the result set
FROM	2-594	Specifies table(s) that contain the selected column(s)
ON	2-609	Specifies join conditions as pre-join filters
WHERE	2-613	Sets conditions on selected rows and post-join filters
GROUP BY	2-621	Combines groups of rows into summary results
HAVING	2-623	Sets conditions on the summary results
ORDER BY	2-624	Sorts the selected rows according to column values
FOR UPDATE	2-629	Enables updating of the selected rows after a fetch
FOR READ ONLY	2-631	Disables updating of the selected rows after a fetch
INTO TEMP	2-633	Puts the results of the query into a temporary table
INTO EXTERNAL	2-635	Stores the results of the query in an external table
INTO SCRATCH	2-637	Stores the results in an unlogged temporary table ♦
UNION ALL	2-637	Combines the result sets of two SELECT statements
UNION	2-637	Same as UNION ALL, but discards duplicate rows

XPS

Sections that follow describe these clauses of the SELECT statement.

Projection Clause

The Projection clause (sometimes called the SELECT clause) specifies a list of database objects or expressions to be retrieved, and whether to omit duplicate values. (The the *select list* is sometimes also called the *projection list*.)



Element	Purpose	Restrictions	Syntax
alias	Temporary *table* or *view* name. See "FROM Clause" on page 2-594.	To use an *alias* here, the FROM clause must declare an *alias* for table or view	Identifier, p. 4-189
column	Column from which to retrieve data	Must exist in a in a data source that the FROM clause references	Identifier, p. 4-189
display _label	Temporary name that you declare here for a column	For restrictions on display labels, see "Declaring a Display Label" on page 2-589	Identifier, p. 4-189
external	External table from which to retrieve data	Must exist	Database Object Name, p. 4-46
max	Integer (> 0) specifying maximum number of rows to return	If *max* > rows that match query criteria, then all matching rows are returned	Literal Number, p. 4-216
subquery	Embedded query	Cannot include the FIRST *max* clause nor the ORDER BY clause	SELECT, p. 2-581
table, view, synonym	Name of a table, view, or synonym from which to retrieve data	The synonym and the table or view to which it points must exist	Database Object Name, p. 4-46

The asterisk (*) specifies all columns in the table or view. Use this symbol to retrieve all the columns in their defined order. To retrieve all the columns in some other order, or to retrieve a subset of the columns, you must specify the individual column names explicitly in the select list.

IDS

A distributed query that accesses tables of another database server cannot reference a column or expression of an opaque or user-defined data type. ♦

Using the FIRST Option

The FIRST *max* option specifies a maximum number of rows to retrieve that match conditions specified in the SELECT statement. Beyond that specified number, any additional rows matching the selection criteria are not returned. The following example retrieves at most 10 rows from a table:

```
SELECT FIRST 10 a, b FROM tab1;
```

When you use this option with an ORDER BY clause, you can retrieve the first number of rows according to the order criteria. For example, the following query finds the ten highest-paid employees:

```
SELECT FIRST 10 name, salary FROM emp ORDER BY salary DESC
```

XPS

If you are using Extended Parallel Server, you can also use the FIRST option to select the first rows that result from a union query. In the following example, the FIRST option is applied to the result of the UNION expression:

```
SELECT FIRST 10 a, b FROM tab1 UNION SELECT a, b FROM tab2     ♦
```

The FIRST option is not valid in the following contexts:

- In the definition of a view
- In nested SELECT statements
- In subqueries
- SPL In a singleton SELECT (where *max* = 1) within an SPL routine ♦
- In the SELECT clause of an INSERT statement
- In a SELECT statement that inserts the retrieved data into another table, such as a temporary, scratch, or external table
- Where embedded SELECT statements are used as expressions
- IDS As part of a UNION query
- In a distributed query on a remote database server ♦

IDS

Using FIRST as a Column Name with Dynamic Server

FIRST is a keyword, but the database server can also interpret it as a column name. If no integer follows the keyword, the database server interprets FIRST as a column identifier. For example, if table **T** has columns **first**, **second**, and **third**, the following query would return data from the column named **first**:

```
SELECT first FROM T
```

XPS

Using the MIDDLE Option

The MIDDLE option, like the FIRST option, can specify a maximum number of rows to retrieve that match conditions specified in the SELECT statement. The FIRST option returns the first *max* rows (for *max* a number that you specify) that satisfy the selection criteria, but the MIDDLE option returns *max* rows from the middle of the set of qualifying rows.

The syntax and restrictions for this option are the same as those for the FIRST option. For more information, see "Using the FIRST Option" on page 2-584.

Allowing Duplicates

You can apply the ALL, UNIQUE, or DISTINCT keywords to indicate whether duplicate values are returned, if any exist. If you do not specify any of these keywords before the projection list, all the rows are returned by default.

Keyword	Effect
ALL	Specifies that all selected values are returned, regardless of whether duplicates exist. (If you specify no keyword, ALL is the default state.)
DISTINCT	Eliminates duplicate rows from the query results
UNIQUE	Eliminates duplicate rows from the query results. (Here UNIQUE as a synonym for DISTINCT is an extension to the ANSI/ISO standard.)

For example, the next query lists unique values in the **stock_num** and **manu_code** columns of all **items** table rows, excluding any duplicate values:

```
SELECT DISTINCT stock_num, manu_code FROM items
```

You can specify DISTINCT or UNIQUE no more than once in each level of a query or subquery. The following example uses DISTINCT in both the query and in the subquery:

```
SELECT DISTINCT stock_num, manu_code FROM items
   WHERE order_num =    (SELECT DISTINCT order_num FROM orders
      WHERE customer_num = 120)
```

Expressions in the Select List

You can use any basic type of expression (column, constant, built-in function, aggregate function, and user-defined routine), or combination thereof, in the select list. The expression types are described in "Expression" on page 4-67. Sections that follow present examples of simple expression in the select list.

You can combine simple numeric expressions by connecting them with arithmetic operators for addition, subtraction, multiplication, and division. If you combine a column expression and an aggregate function, however, you must include the column expression in the GROUP BY clause. (See also "Relationship of GROUP BY and Projection Clauses" on page 2-622.)

In general, you cannot use variables (for example, host variables in an ESQL/C application) in the select list by themselves. You can include a variable in the select list, however, if an arithmetic or concatenation operator connects it to a constant.

In a FOREACH SELECT statement, you cannot use SPL variables in the select list, by themselves or with column names, when the tables in the FROM clause are remote tables. You can use SPL variables by themselves or with a constant in the select list only when the tables in the FROM clause are local tables.

The Boolean operator NOT is not valid in the Projection clause.

Selecting Columns

Column expressions are the most commonly used expressions in a SELECT statement. For a complete description of the syntax and use of column expressions, see "Column Expressions" on page 4-82. The following examples use column expressions in a select list:

```
SELECT orders.order_num, items.price FROM orders, items
SELECT customer.customer_num ccnum, company FROM customer
SELECT catalog_num, stock_num, cat_advert [1,15] FROM catalog
SELECT lead_time - 2 UNITS DAY FROM manufact
```

Selecting Constants

If you include a constant expression in the select list, the same value is returned for each row that the query returns (except when the constant expression is **NEXTVAL**). For a complete description of the syntax and use of constant expressions, see "Constant Expressions" on page 4-95. Examples that follow show constant expressions within a select list:

```
SELECT 'The first name is', fname FROM customer
SELECT TODAY FROM cust_calls
SELECT SITENAME FROM systables WHERE tabid = 1
SELECT lead_time - 2 UNITS DAY FROM manufact
SELECT customer_num + LENGTH('string') from customer
```

Selecting Built-In Function Expressions

A built-in function expression uses a function that is evaluated for each row in the query. All built-in function expressions require arguments. This set of expressions contains the time functions and the length function when they are used with a column name as an argument. The following examples show built-in function expressions within the select list of the Projection clause:

```
SELECT EXTEND(res_dtime, YEAR TO SECOND) FROM cust_calls
SELECT LENGTH(fname) + LENGTH(lname) FROM customer
SELECT HEX(order_num) FROM orders
SELECT MONTH(order_date) FROM orders
```

Selecting Aggregate Function Expressions

An aggregate function returns one value for a set of queried rows. This value depends on the set of rows that the WHERE clause of the SELECT statement qualifies. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows that the FROM clause forms. Examples that follow show aggregate functions in a select list:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013
SELECT COUNT(*) FROM orders WHERE order_num = 1001
SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer
```

Selecting User-Defined Function Expressions

User-defined functions extend the range of functions that are available to you and allow you to perform a subquery on each row that you select.

The following example calls the **get_orders( )** user-defined function for each **customer_num** and displays the returned value under the `n_orders` label:

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
      FROM customer
```

SPL

If an SPL routine in a SELECT statement contains certain SQL statements, the database server returns an error. For information on which SQL statements cannot be used in an SPL routine that is called within a query, see "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279. ♦

For the complete syntax of user-defined function expressions, see "User-Defined Functions" on page 4-165.

Selecting Expressions That Use Arithmetic Operators

You can combine numeric expressions with arithmetic operators to make complex expressions. You cannot combine expressions that contain aggregate functions with column expressions. These examples show expressions that use arithmetic operators within a select list in the Projection clause:

```
SELECT stock_num, quantity*total_price FROM customer
SELECT price*2 doubleprice FROM items
SELECT count(*)+2 FROM customer
SELECT count(*)+LENGTH('ab') FROM customer
```

IDS

Selecting ROW Fields

You can select a specific field of a named or unnamed ROW type column with *row*.*field* notation, using a period (.) as a separator between the *row* and *field* names. For example, suppose you have the following table structure:

```
CREATE ROW TYPE one (a INTEGER, b FLOAT)
CREATE ROW TYPE two (c one, d CHAR(10))
CREATE ROW TYPE three (e CHAR(10), f two)
CREATE TABLE new_tab OF TYPE two
CREATE TABLE three_tab OF TYPE three
```

The following examples show expressions that are valid in the select list:

```
SELECT t.c FROM new_tab t
SELECT f.c.a FROM three_tab
SELECT f.d FROM three_tab
```

You can also enter an asterisk (*) in place of a field name to signify that all fields of the ROW-type column are to be selected.

For example, if the **my_tab** table has a ROW-type column named **rowcol** that contains four fields, the following SELECT statement retrieves all four fields of the **rowcol** column:

```
SELECT rowcol.* FROM my_tab
```

You can also retrieve all fields from a row-type column by specifying only the column name. This example has the same effect as the previous query:

```
SELECT rowcol FROM my_tab
```

You can use *row.field* notation not only with ROW-type columns but with expressions that evaluate to ROW-type values. For more information, see "Column Expressions" on page 4-82 in the Expression segment.

Declaring a Display Label

You can declare a display label for any column or column expression in the select list of the Projection clause. This temporary name is in scope only while the SELECT statement is executing.

DB

In DB-Access, a display label appears as the heading for that column in the output of the SELECT statement. ♦

E/C

In ESQL/C, the value of *display_label* is stored in the **sqlname** field of the **sqlda** structure. For more information on the **sqlda** structure, see the *IBM Informix ESQL/C Programmer's Manual*. ♦

If your display label is an SQL keyword, use the AS keyword to clarify the syntax. For example, to use UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION as a display label, use the AS keyword with the display label. The next statement uses AS with **minute** as a display label:

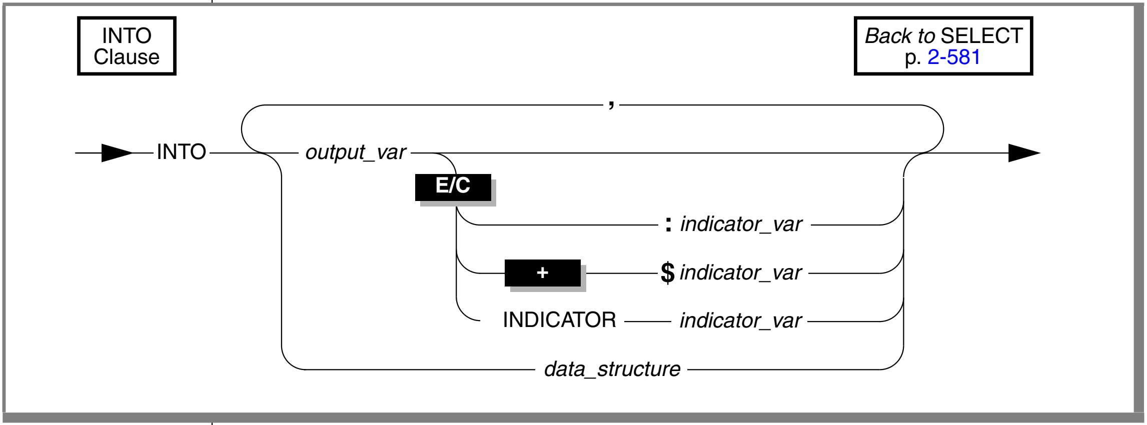
```
SELECT call_dtime AS minute FROM cust_calls
```

For the keywords of SQL, see Appendix A, "Reserved Words for IBM Informix Dynamic Server," or Appendix B, "Reserved Words for IBM Informix Extended Parallel Server."

If you are creating a temporary table, you must supply a display label for any columns that are not simple column expressions. The display label is used as the name of the column in the temporary table. If you are using the SELECT statement to define a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead.

INTO Clause

Use the INTO clause in an SPL routine or an ESQL/C program to specify the program variables or host variables to receive data that SELECT retrieves.



Element	Purpose	Restrictions	Syntax
data_structure	Structure that was declared as a host variable	Data types of elements must be able to store the values that are being selected	Language specific
indicator_var	Program variable to receive a return code if corresponding *output_var* receives a NULL value	Optional; use an indicator variable if the possibility exists that the value of the corresponding *output_var* is NULL.	Language specific
output_var	Program or host variable to receive value of the corresponding select list item. Can be a collection variable	Order of receiving variables must match the order of corresponding items in the select list of Projection clause	Language specific

The INTO clause specifies one or more variables that receive the values that the query returns. If it returns multiple values, they are assigned to the list of variables in the order in which you specify the variables.

If the SELECT statement stands alone (that is, it is not part of a DECLARE statement and does not use the INTO clause), it must be a singleton SELECT statement. A *singleton* SELECT statement returns only one row.

The number of receiving variables must be equal to the number of items in the select list of the Projection clause. The data type of each receiving variable should be compatible with the data type of the corresponding column or expression in the select list.

For the actions that the database server takes when the data type of the receiving variable does not match that of the selected item, see "Warnings in ESQL/C" on page 2-593.

The following example shows a singleton SELECT statement in ESQL/C:

```
EXEC SQL select fname, lname, company_name
   into :p_fname, :p_lname, :p_coname
   where customer_num = 101;
```

SPL

In an SPL routine, if a SELECT returns more than one row, you must use the FOREACH statement to access the rows individually. The INTO clause of the SELECT statement holds the fetched values. For more information, see "FOREACH" on page 3-27. ♦

E/C

INTO Clause with Indicator Variables

If the possibility exists that a data value returned from the query is NULL, use an ESQL/C indicator variable in the INTO clause. For more information, see the *IBM Informix ESQL/C Programmer's Manual*.

INTO Clause with Cursors

If the SELECT statement returns more than one row, you must use a cursor in a FETCH statement to fetch the rows individually. You can put the INTO clause in the FETCH statement rather than in the SELECT statement, but you should not put it in both.

The following ESQL/C code examples show different ways you can use the INTO clause. As both examples show, first you must use the DECLARE statement to declare a cursor.

Using the INTO clause in the SELECT statement

```
EXEC SQL declare q_curs cursor for
   select lname, company
      into :p_lname, :p_company
      from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
   EXEC SQL fetch q_curs;
EXEC SQL close q_curs;
```

Using the INTO clause in the FETCH statement

```
EXEC SQL declare q_curs cursor for
   select lname, company from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
   EXEC SQL fetch q_curs into :p_lname, :p_company;
EXEC SQL close q_curs;
```

E/C

Preparing a SELECT... INTO Query

In ESQL/C, you cannot prepare a query that has an INTO clause. You can prepare the query without the INTO clause, declare a cursor for the prepared query, open the cursor, and then use the FETCH statement with an INTO clause to fetch the cursor into the program variable.

Alternatively, you can declare a cursor for the query without first preparing the query and include the INTO clause in the query when you declare the cursor. Then open the cursor and fetch the cursor without using the INTO clause of the FETCH statement.

E/C

Using Array Variables with the INTO Clause

In ESQL/C, if you use a DECLARE statement with a SELECT statement that contains an INTO clause, and the variable is an array element, you can identify individual elements of the array with integer literals or variables. The value of the variable that is used as a subscript is determined when the cursor is declared; the subscript variable subsequently acts as a constant.

The following ESQL/C code example declares a cursor for a SELECT ... INTO statement using the variables **i** and **j** as subscripts for the array **a**. After you declare the cursor, the INTO clause of the SELECT statement is equivalent to `INTO a[5], a[2].`

```
i = 5
j = 2
EXEC SQL declare c cursor for
   select order_num, po_num into :a[i], :a[j] from orders
      where order_num =1005 and po_num =2865
```

You can also use program variables in the FETCH statement to specify an element of a program array in the INTO clause. With the FETCH statement, the program variables are evaluated at each fetch rather than when you declare the cursor.

Error Checking

If the data type of the receiving variable does not match that of the selected item, the data type of the selected item is converted, if possible, to the data type of the variable. If the conversion is impossible, an error occurs, and a negative value is returned in the **status** variable, **sqlca.sqlcode,** or **SQLCODE**. In this case, the value in the program variable is unpredictable.

ANSI

In an ANSI-compliant database, if the number of variables that are listed in the INTO clause differs from the number of items in the select list of the Projection clause, you receive an error. ♦

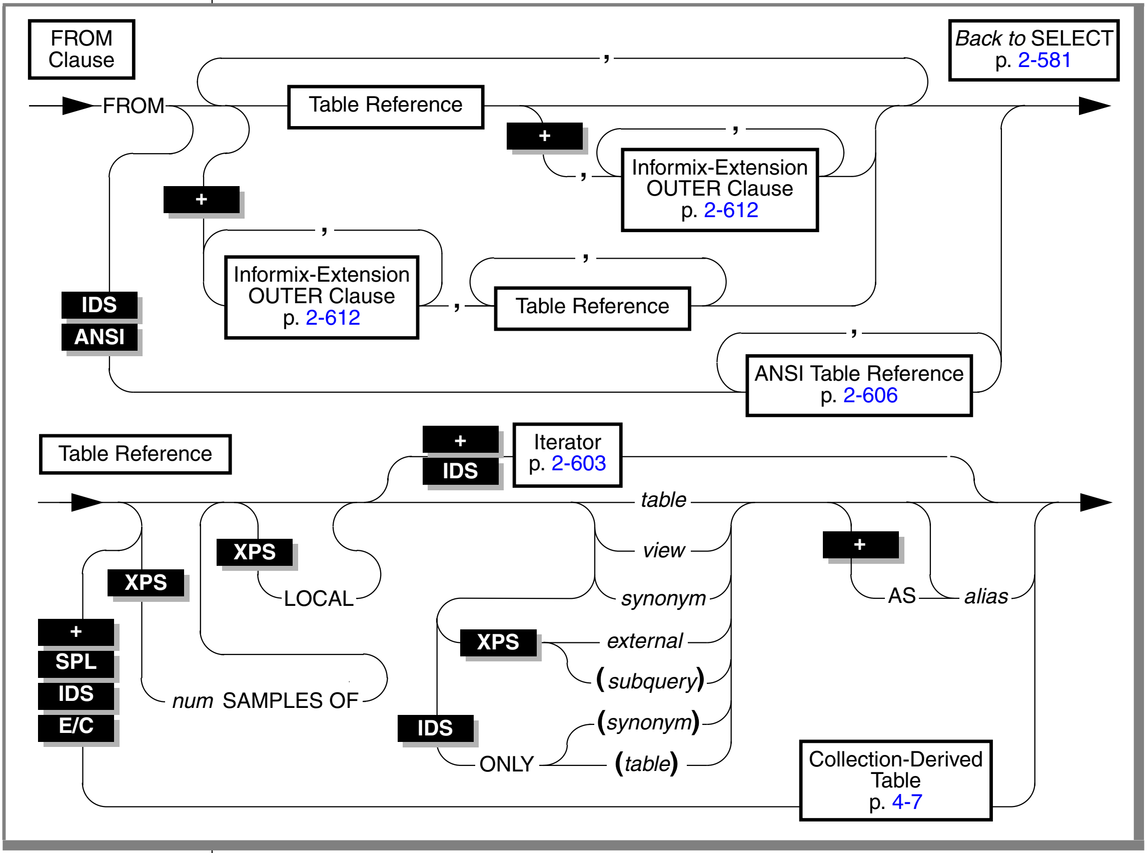
E/C

Warnings in ESQL/C

In ESQL/C, if the number of variables listed in the INTO clause differs from the number of items in the Projection clause, a warning is returned in the **sqlwarn** structure: **sqlca.sqlwarn.sqlwarn3**. The actual number of variables that are transferred is the lesser of the two numbers. For information about the **sqlwarn** structure, see the *IBM Informix ESQL/C Programmer's Manual*.

FROM Clause

The FROM clause lists the tables from which you are selecting the data.



Element	Purpose	Restrictions	Syntax
alias	Temporary name for a table or view in the SELECT statement	See "The AS Keyword" on page 2-595.	Identifier, p. 4-216
external	External table from which to retrieve data	Must exist but cannot be the outer table in an outer join.	Database Object Name, p. 4-46
num	Number of rows to be sampled	Unsigned integer > 0.	Literal Number, p. 4-216
subquery	Specifies rows to be retrieved	Cannot be a correlated subquery.	SELECT, p. 2-581
synonym, table, view	Synonym or name of a source from which to retrieve data	Synonym and table or view to which it points must exist.	Database Object Name, p. 4-46

If the FROM clause specifies more than one data source, the query is called a *join*, because its result set can join rows from several table references. For more information about joins, see "Queries that Join Tables" on page 2-604.

Aliases for Tables or Views

You can declare an alias for a table or view in the FROM clause. If you do so, you must use the alias to refer to the table or view in other clauses of the SELECT statement. You can also use aliases to make the query shorter.

The following example shows typical uses of the FROM clause. The first query selects all the columns and rows from the **customer** table. The second query uses a join between the **customer** and **orders** table to select all the customers who have placed orders.

```
SELECT * FROM customer
SELECT fname, lname, order_num FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
```

The next example is equivalent to the second query in the preceding example, but it declares aliases in the FROM clause and uses them in the WHERE clause:

```
SELECT fname, lname, order_num FROM customer c, orders o
   WHERE c.customer_num = o.customer_num
```

Aliases (sometimes called *correlation names*) are especially useful with a self-join. For more information about self-joins, see "Self-Joins" on page 2-620. In a self-join, you must list the table name twice in the FROM clause and declare a different alias for each of the two instances of table name.

The AS Keyword

If you use a potentially ambiguous word as an alias (or a display label), you must begin its declaration with the keyword AS. This keyword is required if you use any of the keywords ORDER, FOR, AT, GROUP, HAVING, INTO, NOT, UNION, WHERE, WITH, CREATE, or GRANT as an alias for a table or view.

The database server would issue an error if the next example did not include the AS keyword to indicate that **not** is a display label, rather than an operator:

```
CREATE TABLE t1(a INT);
SELECT a AS not FROM t1
```

If you do not declare an alias for a collection-derived table, the database server assigns an implementation-dependent name to it.

XPS

Table Expressions

The term *table expression* refers to the use of a view name, a table name, or uncorrelated subquery in the FROM clause. These can be simple or complex:

- Simple table expressions

  A *simple* table expression is one whose underlying query can be folded into the main query while preserving the correctness of the query result.

- Complex table expressions

  A *complex* table expression is one whose underlying query cannot be folded into the main query while preserving the correctness of the query result. The database server materializes such table expressions into a temporary table that is used in the main query.

In either case, the table expression is evaluated as a general SQL query, and its results can be thought of as a logical table. This logical table and its columns can be used just like an ordinary base table, but it is not persistent. It exists only during the execution of the query that references it.

Table expressions have the same syntax as general SELECT statements, but with the same restrictions that apply to subqueries in other contexts. A table expression cannot include the following syntax elements:

- ORDER BY clause
- SELECT INTO clause

In addition, table expressions are not valid in the following contexts:

- CREATE TRIGGER statements
- CREATE GK INDEX statements

Queries and correlated subqueries are not supported in the FROM clause.

Apart from these restrictions, any valid SQL query can be a table expression. A table expression can be nested within another table expression, and can include tables and views in its definition. You can use table expressions in CREATE VIEW statements to define views.

Usability and Performance Considerations

Although equivalent functionality is available through views, subqueries as table expressions simplify the formulation of queries, make the syntax more flexible and intuitive, and support the ANSI/ISO standard for SQL.

Performance might be affected, however, if you use table expressions. It is advisable to use subqueries if you really do not need to use table expressions.

The following are examples of valid table expressions:

```
SELECT * FROM (SELECT * FROM t);

SELECT * FROM (SELECT * FROM t) AS s;

SELECT * FROM (SELECT * FROM t) AS s WHERE t.a = s.b;

SELECT *
FROM (SELECT * FROM t) AS s, (SELECT * FROM u) AS v
   WHERE s.a = v.b;

SELECT * FROM (SELECT * FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT * FROM u WHERE u.b = 2 GROUP BY 1) AS v
   WHERE s.a = v.b;

SELECT * FROM (SELECT a AS colA FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT b AS colB FROM u WHERE u.b = 2 GROUP BY 1) AS v
   WHERE s.colA = v.colB;

CREATE VIEW vu AS SELECT * FROM (SELECT * FROM t);

SELECT * FROM ((SELECT * FROM t) AS r) AS s;
```

XPS

Restrictions on External Tables in Joins and Subqueries

In Extended Parallel Server, when you use external tables in joins or subqueries, the following restrictions apply:

- No more than one external table is valid in a query.
- The external table cannot be the outer table in an outer join.
- For subqueries that cannot be converted to joins, you can use an external table in the main query, but not the subquery.
- You cannot do a self-join on an external table.

For more information on subqueries, refer to your *Performance Guide*.

XPS

Application Partitioning: The LOCAL Keyword

In Extended Parallel Server, the LOCAL table feature allows client applications to read data only from the *local* fragments of a table. In other words, it allows the application to read only the fragments that reside on the coserver to which the client is connected. This feature supports application partitioning. An application can connect to multiple coservers, execute a LOCAL read on each coserver, and assemble the final result on the client computer.

You qualify the name of a table with the LOCAL keyword to indicate that you want to retrieve rows from fragments only on the local coserver. The LOCAL keyword has no effect on data retrieved from nonfragmented tables.

When a query involves a join, you must plan carefully if you want to extract data that the client can aggregate. The simplest way to ensure that a join will retrieve data suitable for aggregation is to limit the number of LOCAL tables to one. The client can then aggregate data with respect to that table. The following example shows a query that returns data suitable for aggregation by the client:

```
SELECT x.col1, y.col2
   FROM LOCAL tab1 x, tab2 y     {can be aggregated by client}
   INTO TEMP t1
   WHERE x.col1 = y.col1;        {tab1 is local}
```

The following example shows data that the client cannot aggregate:

```
SELECT x.col1, y.col2
   FROM LOCAL tab1 x, LOCAL tab2     {client cannot aggregate}
   INTO SCRATCH s4
   WHERE x.col1 = y.col1;            {tab1 and tab2 are local}
```

The client must submit exactly the same query to each coserver to retrieve data that can be aggregated.

XPS

Sampled Queries: The SAMPLES OF Keywords

In Extended Parallel Server, *sampled queries* are supported. Sampled queries are queries that are based on *sampled tables*. A sampled table is the result of randomly selecting a specified number of rows from the table, rather than all rows that match the selection criteria.

You can use a sampled query to gather quickly an approximate profile of data within a large table. If you use a sufficiently large sample size, you can examine trends in the data by sampling the data instead of scanning all the data. In such cases, sampled queries can provide better performance than scanning the data. If the value specified for the sample size is greater than the number of rows in the table, the whole table is scanned.

To indicate that a table is to be sampled, specify the number of samples to return before the SAMPLES OF keywords of the FROM clause. You can run sampled queries against tables and synonyms, but not against views.

Sampled queries are not supported in the INSERT, DELETE, UPDATE, or other SQL statements.

A sampled query has at least one sampled table. You do not need to sample all tables in a sampled query. You can specify the SAMPLES OF option for some tables in the FROM clause but not specify it for other tables.

The sampling method is known as *sampling without replacement*. This means that a sampled row is not sampled again. The database server applies selection criteria *after* samples are selected. Thus, the selection criteria restrict the sample set, rather than the rows from which the sample is taken.

If a table is fragmented, the database server divides the specified number of samples among the fragments. The number of samples from a fragment is proportional to the ratio of the size of a fragment to the size of the table. In other words, the database server takes more samples from larger fragments.

Important: *You must run UPDATE STATISTICS LOW before you run the query with the SAMPLES OF option. If you do not run UPDATE STATISTICS, the SAMPLE clause is ignored, and all data values are returned. For better results, run UPDATE STATISTICS MEDIUM before you run the query with the SAMPLES OF option.*

The results of a sampled query contain a certain amount of deviation from a complete scan of all rows. You can reduce this expected error to an acceptable level by increasing the proportion of sampled rows to actual rows. When you use sampled queries in joins, the expected error increases dramatically; you must use larger samples in each table to retain an acceptable level of accuracy.

For example, you might want to generate a list of how many of each part is sold from the **parts_sold** table, which is known to contain approximately 100,000,000 rows.

The following query provides a sampling ratio of one percent and returns an approximate result:

```
SELECT part_number, COUNT(*) * 100 AS how_many
   FROM 1000000 SAMPLES OF parts_sold
   GROUP BY part_number;
```

IDS

The ONLY Keyword

If the SELECT statement queries a supertable, rows from both the supertable and its subtables are returned. To query rows from the supertable only, you must include the ONLY keyword in the FROM clause, as in this example:

```
SELECT * FROM ONLY(super_tab)
```

IDS

Selecting from a Collection Variable

The SELECT statement in conjunction with the Collection-Derived-Table segment allows you to select elements from a collection variable.

The Collection-Derived-Table segment identifies the collection variable from which to select the elements. (See "Collection-Derived Table" on page 4-7.)

Using Collection Variables with SELECT

To modify the contents of a column of a collection data type, you can use the SELECT statement with a collection variable in different ways:

- You can select the contents (if any) of a collection column into a collection variable.

  You can assign the data type of the column to a collection variable of type COLLECTION (that is, an untyped collection variable).
- You can select the contents from a collection variable to determine the data that you might want to update.
- You can select the contents from a collection variable INTO another variable in order to update certain collection elements.

  The INTO clause identifies the variable for the element value that is selected from the collection variable. The data type of the host variable in the INTO clause must be compatible with the data type of the corresponding element of the collection.

E/C

- You can use a collection cursor to select one or more elements from an ESQL/C collection variable.

  For more information, including restrictions on the SELECT statement, see "Associating a Cursor with a Collection Variable" on page 2-339 in the DECLARE statement. ♦

SPL

- You can use a collection cursor to select one or more elements from an SPL collection variable.

  For more information, including restrictions on the SELECT statement, see "Using a SELECT...INTO Statement" on page 3-29. ♦

When one of the tables to be joined is a collection, the FROM clause cannot specify a join. This restriction applies when the collection variable holds your collection-derived table. See also "Collection-Derived Table" on page 4-7 and the INSERT, UPDATE, and DELETE statements in this chapter.

IDS

E/C

Selecting from a Row Variable

The SELECT statement can include the Collection-Derived-Table segment to select one or more fields from a **row** variable. The Collection-Derived-Table segment identifies the **row** variable from which to select the fields. For more information, see "Collection-Derived Table" on page 4-7.

To select fields

1. Create a **row** variable in your ESQL/C program.
2. Optionally, fill the **row** variable with field values.

   You can select a ROW-type column into the **row** variable with the SELECT statement (without the Collection-Derived-Table segment). Alternatively, you can insert field values into the **row** variable with the UPDATE statement and the Collection-Derived-Table segment.
3. Select row fields from the **row** variable with the SELECT statement and the Collection-Derived-Table segment.
4. Once the **row** variable contains the correct field values, you can use the INSERT or UPDATE statement on a table or view name to save the contents of the **row** variable in a named or unnamed ROW column.

The INTO clause can specify a host variable to hold a field value selected from the **row** variable.

The type of the host variable must be compatible with that of the field. For example, this code fragment puts the **width** field value into the **rect_width** host variable.

```
EXEC SQL BEGIN DECLARE SECTION;
   ROW (x INT, y INT, length FLOAT, width FLOAT) myrect;
   double rect_width;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT rect INTO :myrect FROM rectangles
   WHERE area = 200;
EXEC SQL SELECT width INTO :rect_width FROM table(:myrect);
```

The SELECT statement on a **row** variable has the following restrictions:

- No expressions are allowed in the select list of the Projection clause.
- ROW columns cannot be in a WHERE clause comparison condition.
- The Projection clause must be an asterisk (*) if the row-type contains fields of opaque, distinct, or built-in data types.
- Columns listed in the Projection clause can have only unqualified names. They cannot use the syntax *database@server:table.column.*
- The following clauses are not allowed: GROUP BY, HAVING, INTO TEMP, ORDER BY, and WHERE.
- The FROM clause has no provisions to do a join.

You can modify the **row** variable with the Collection-Derived-Table segment of the UPDATE statements. (The INSERT and DELETE statements do not support a **row** variable in the Collection-Derived-Table segment.)

The **row** variable stores the fields of the row. It has no intrinsic connection, however, with a database column. Once the **row** variable contains the correct field values, you must then save the variable into the ROW column with one of the following SQL statements:

- To update the ROW column in the table with the **row** variable, use an UPDATE statement on a table or view name and specify the **row** variable in the SET clause. For more information, see "Updating ROW-Type Columns" on page 2-770.
- To insert a row into a ROW column, use the INSERT statement on a table or view and specify the **row** variable in the VALUES clause. See "Inserting Values into ROW-Type Columns" on page 2-497.

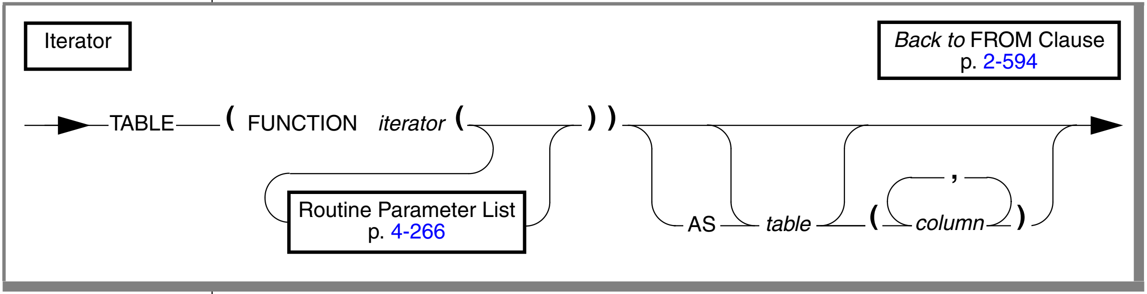
For examples of how to use SPL row variables, see the *IBM Informix Guide to SQL: Tutorial*. For information using ESQL/C **row** variables, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

IDS

Iterator Functions

The FROM clause can include a call to an iterator function to specify the source for a query. An *iterator function* is a user-defined function that returns to its calling SQL statement several times, each time returning a value.

You can query the returned result set of an iterator UDR using a virtual table interface. Use this syntax to invoke an iterator function in the FROM clause.



Element	Purpose	Restrictions	Syntax
column	Name for a virtual column in *table*	Cannot include qualifiers	Identifier, p. 4-189
iterator	Name of the iterator function	Must be registered in the database	Identifier, p. 4-189
table	Name declared here for the virtual table holding *iterator* result set	Cannot include qualifiers	Identifier, p. 4-189

The *table* can only be referenced within the context of this query. After the SELECT statement terminates, the virtual table no longer exists.

The number of columns must match the number of values returned by the iterator. An external function can return no more than one value (but that can be of a collection data type.) An SPL routine can return multiple values.

To reference the virtual *table* columns in other parts of the SELECT statement, for example, in the WHERE clause, or HAVING clause, you must declare its name and the virtual column names in the FROM clause.

You do not need to declare the *table* name or *column* names in the FROM clause if you use the asterisk notation in the Projection list of the SELECT clause:

```
SELECT * FROM ...
```

For more information and examples of using iterator functions in queries, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Queries that Join Tables

If the FROM clause specifies more than one table reference, the query can join rows from several tables. A *join condition* specifies a relationship between at least one column from each table to be joined. Because the columns in a join condition are being compared, they must have compatible data types.

The FROM clause of the SELECT statement can specify several types of joins:

FROM Clause Keyword	Corresponding Result Set
CROSS JOIN	Cartesian product (all possible pairs of rows)
INNER JOIN	Only rows from CROSS that satisfy the join condition
LEFT OUTER JOIN	Qualifying rows of one table, and all rows of another
RIGHT OUTER JOIN	Same as LEFT, but roles of the two tables are reversed
FULL OUTER JOIN	The union of all rows from an INNER join of the two tables, and of all rows of each table that have no match in the other table (using NULL values in the selected columns of the other table).

The last four categories are collectively called "Join Types" in the literature of the relational model; a CROSS JOIN ignores the data values in joined tables, returning every possible pair of rows where one row is from each table.

In an inner (or simple) join, the result contains only the combination of rows that satisfy the join conditions. Outer joins preserve rows that otherwise would be discarded by inner joins. In an outer join, the result contains the combination of rows that satisfy the join conditions *and* the rows from the dominant table that would otherwise be discarded. The rows from the dominant table that do not have matching rows in the subordinate table contain NULL values in the columns selected from the subordinate table.

IDS

Dynamic Server supports the two different syntaxes for left outer joins:

- Informix-extension syntax
- ANSI-compliant syntax

Earlier versions of the database server supported only Informix-extension syntax for outer joins. Dynamic Server continues to support this legacy syntax, but using the ANSI-compliant syntax for joins provides greater flexibility in creating queries. In view definitions, however, the legacy syntax does not require materialized views, so it might offer performance advantages.

If you use ANSI-compliant syntax to specify a join in the FROM clause, you must also use ANSI-compliant syntax for all outer joins in the same query block. Thus, you cannot begin another outer join with only the OUTER keyword. The following query, for example, is not valid:

```
SELECT * FROM customer, OUTER orders RIGHT JOIN cust_calls
   ON (customer.customer_num = orders.customer_num)
   WHERE customer.customer_num = 104);
```

This returns an error because it attempts to combine the Informix-extension OUTER syntax with the ANSI-compliant RIGHT JOIN syntax for outer joins.

See the section "Informix-Extension Outer Joins" on page 2-612 for the Informix-extension syntax for LEFT OUTER joins. ♦

IDS
ANSI

ANSI-Compliant Joins

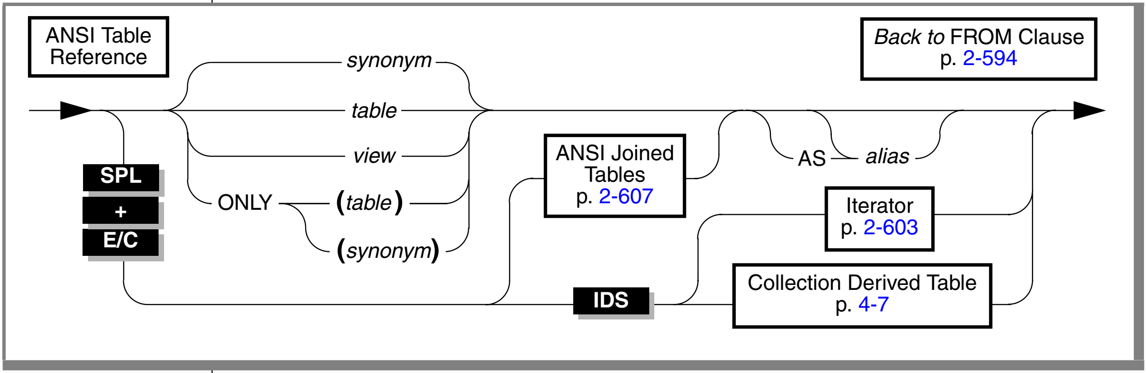
The ANSI-compliant syntax for joins supports these join specifications:

- To use a CROSS join, or a LEFT OUTER, RIGHT OUTER, or FULL OUTER join, or an INNER (or simple) join, see "Creating an ANSI Join" on page 2-606 and "ANSI INNER Joins" on page 2-608.
- To use pre-join filters, see "Using the ON Clause" on page 2-609.
- To use one or more post-join filters in the WHERE clause, see "Specifying a Post-Join Filter" on page 2-610.)
- To have the dominant or subordinate part of an outer join be the result set of another join, see "Using a Join as the Dominant or Subordinate Part of an Outer Join" on page 2-611.

Important: *Use the ANSI-compliant syntax for joins when you create new queries in Dynamic Server.* ♦

ANSI Table Reference

This diagram shows the ANSI-compliant syntax for a table reference.



Element	Purpose	Restrictions	Syntax
alias	Temporary name for a table or view within the scope of the SELECT	See "The AS Keyword" on page 2-595.	Identifier, p. 4-189
synonym, table, view	Source from which to retrieve data	The synonym and the table or view to which it points must exist.	Database Object Name, p. 4-46

Here the ONLY keyword has the same semantics as in the Informix-extension Table Reference segment, as described in "The ONLY Keyword" on page 2-600.

The AS keyword is optional when you declare an alias (also called a *correlation name*) for a table reference, as described in "The AS Keyword" on page 2-595, unless the alias conflicts with an SQL keyword.

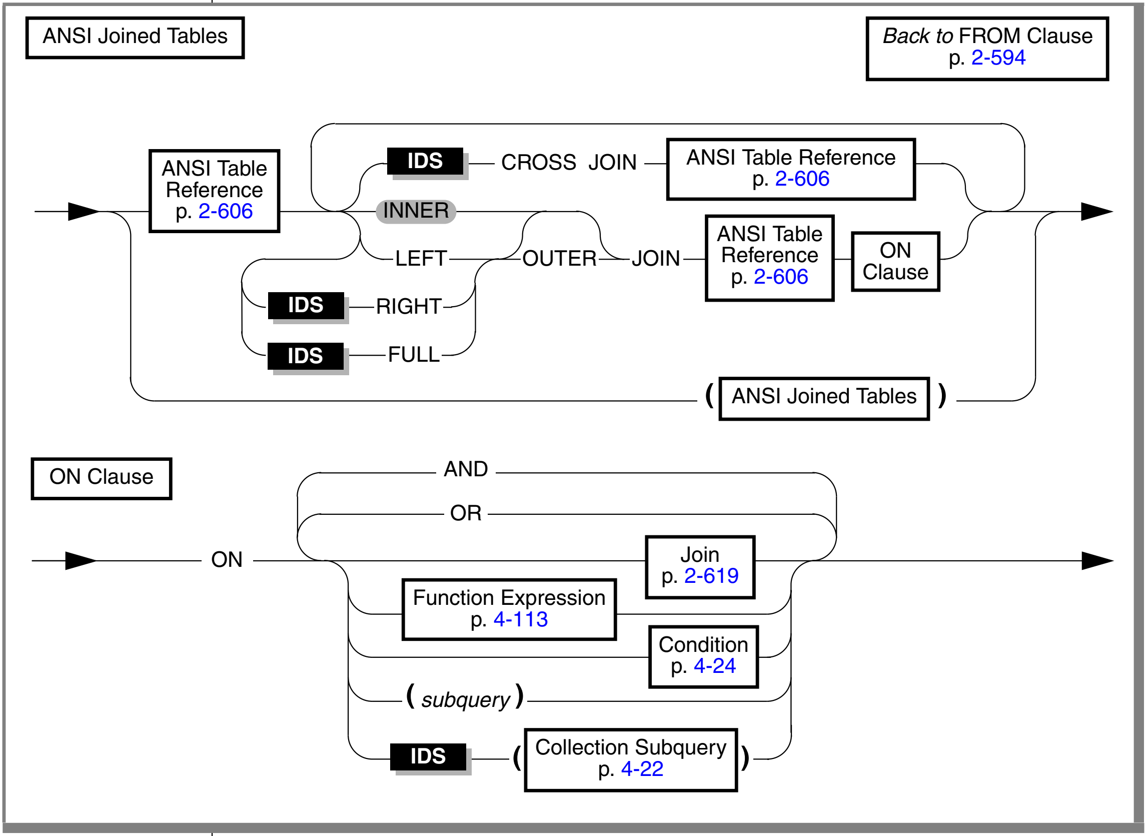
Creating an ANSI Join

With ANSI-compliant joined table syntax, as shown in the following diagram, you can specify the INNER JOIN, CROSS JOIN, NATURAL JOIN, LEFT JOIN (or LEFT OUTER JOIN), RIGHT JOIN (and FULL OUTER JOIN keywords. The OUTER keyword is optional in ANSI-compliant outer joins.

You must use the same form of join syntax (either Informix extension or ANSI-compliant) for all outer joins in the same query block. Except for CROSS joins, the ANSI-compliant syntax requires you to specify the join condition in the ON clause, as described in "Using the ON Clause" on page 2-609.

ANSI Joined Tables

This is the ANSI-compliant syntax for specifying inner and outer joins.



Element	**Purpose**	**Restrictions**	**Syntax**
subquery	Embedded query	Cannot contain the FIRST or the ORDER BY clause	SELECT, p. 2-581

The ANSI-Joined Table segment must be enclosed in parentheses if it is immediately followed by another join specification. For example, the first of the following two queries returns an error; the second query is correct:

```
SELECT * FROM (T1 LEFT JOIN T2) CROSS JOIN T3 ON (T1.c1 = T2.c5)
      WHERE (T1.c1 < 100);     -- Ambiguous order of operations
SELECT * FROM ((T1 LEFT JOIN T2)) CROSS JOIN T3 ON (T1.c1 = T2.c5)
      WHERE (T1.c1 < 100);     -- Unambiguous order of operations
```

ANSI CROSS Joins

The CROSS keyword specifies a Cartesian product (with no ON clause), to return all possible paired combinations that include a row from each table.

ANSI INNER Joins

To create an inner (or simple) join using the ANSI-compliant syntax, specify the join with the JOIN or INNER JOIN keywords. If you specify only the JOIN keyword, the database server creates an implicit inner join by default. An inner join returns all the rows in a table that have one or more matching rows in the other table (or tables). The unmatched rows are discarded.

ANSI LEFT OUTER Joins

The LEFT keyword specifies a join that treats the first table reference as the dominant table in the join. In a left outer join, the subordinate part of the outer join appears to the right of the keyword that begins the outer join specification. The result set includes all the rows that an INNER join returns, plus all rows that would otherwise have been discarded from the dominant table.

ANSI RIGHT OUTER Joins

The RIGHT keyword specifies a join that treats the second table reference as the dominant table in the join. In a right outer join, the subordinate part of the outer join appears to the left of the keyword that begins the outer join specification. The result set includes all the rows that an INNER join returns, plus all rows that would otherwise have been discarded from the dominant table.

ANSI FULL OUTER Joins

The FULL keyword specifies the union of (1) the result of an INNER join of two tables, and (2) all rows of the left table that have no match in the right table, and (3) all rows of the right table that have no match in the left table. (Here (2) and (3) return NULL values in the selected columns of the other table.)

In an ANSI-compliant join that specifies the LEFT, RIGHT, or FULL keywords in the FROM clause, the OUTER keyword is optional.

Optimizer directives that you specify for an ANSI-compliant joined query are ignored, but are listed under Directives Not Followed in **sqlexplain.out**.

Using the ON Clause

Use the ON clause to specify the join condition and any expressions as optional join filters for inner and outer joins. It is not valid with cross joins..

The following example from the **stores_demo** database illustrates how the join condition in the ON clause combines the **customer** and **orders** tables:

```
SELECT c.customer_num, c.company, c.phone, o.order_date
   FROM customer c LEFT JOIN orders o
      ON c.customer_num = o.customer_num
```

The following table shows part of the joined **customer** and **orders** tables.

customer_num	company	phone	order_date
101	All Sports Supplies	408-789-8075	05/21/1998
102	Sports Spot	415-822-1289	null
103	Phil's Sports	415-328-4543	null
104	Play Ball!	415-368-1100	05/20/1998
—	—	—	—

In an outer join, the join filters (expressions) that you specify in the ON clause determine which rows of the subordinate table join to the dominant (or outer) table. The dominant table, by definition, returns all its rows in the joined table. That is, a join filter in the ON clause has no effect on the dominant table.

If the ON clause specifies a join filter on the dominant table, the database server joins only those dominant table rows that meet the criterion of the join filter to rows in the subordinate table. The joined result contains all rows from the dominant table. Rows in the dominant table that do not meet the criterion of the join filter are extended with NULL values for the subordinate columns.

The following example from the **stores_demo** database illustrates the effect of a join filter in the ON clause:

```
SELECT c.customer_num, c.company, c.phone, o.order_date
   FROM customer c LEFT JOIN orders o
      ON c.customer_num = o.customer_num
         AND c.company <> "All Sports Supplies"
```

The row that contains All Sports Supplies remains in the joined result.

customer_num	company	phone	order_date
101	All Sports Supplies	408-789-8075	null
102	Sports Spot	415-822-1289	null
103	Phil's Sports	415-328-4543	null
104	Play Ball!	415-368-1100	05/20/1998
—	—	—	—

Even though the order date for customer number 101 is 05/21/1998 in the **orders** table, the effect of placing the join filter (`c.company <> "All Sports Supplies"`) prevents this row in the dominant **customer** table from being joined to the subordinate **orders** table. Instead, a NULL value for **order_date** is extended to the row of All Sports Supplies.

Applying a join filter to a base table in the subordinate part of an outer join can improve performance. For more information, see your *Performance Guide*.

Specifying a Post-Join Filter

When you use the ON clause to specify the join, you can use the WHERE clause as a post-join filter. The database server applies the post-join filter of the WHERE clause to the results of the outer join.

The following example illustrates the use of a post-join filter. This query returns data from the **stores_demo** database. Suppose you want to determine which items in the catalog are not being ordered. The next query creates an outer join of the data from the **catalog** and **items** tables and then determines which catalog items from a specific manufacturer (HRO) have not sold:

```
SELECT c.catalog_num, c.stock_num, c.manu_code, i.quantity
FROM catalog c LEFT JOIN items i
   ON c.stock_num = i.stock_num AND c.manu_code = i.manu_code
   WHERE i.quantity IS NULL AND c.manu_code = "HRO"
```

The WHERE clause contains the post-join filter that locates the rows of HRO items in the catalog for which nothing has been sold.

When you apply a post-join filter to a base table in the dominant or subordinate part of an outer join, you might improve performance. For more information, see your *Performance Guide*.

Using a Join as the Dominant or Subordinate Part of an Outer Join

With the ANSI join syntax, you can nest joins. You can use a join as the dominant or subordinate part of an outer or inner join.

Suppose you want to modify the previous query (the post-join filter example) to get more information that will help you determine whether to continue carrying each unsold item in the catalog. You can modify the query to include information from the **stock** table so that you can see a short description of each unsold item with its cost.

```
SELECT c.catalog_num, c.stock_num, s.description, s.unit_price,
   s.unit_descr, c.manu_code, i.quantity
FROM (catalog c INNER JOIN stock s
   ON c.stock_num = s.stock_num
      AND c.manu_code = s.manu_code)
    LEFT JOIN items i
      ON c.stock_num = i.stock_num
         AND c.manu_code = i.manu_code
   WHERE i.quantity IS NULL
      AND c.manu_code = "HRO"
```

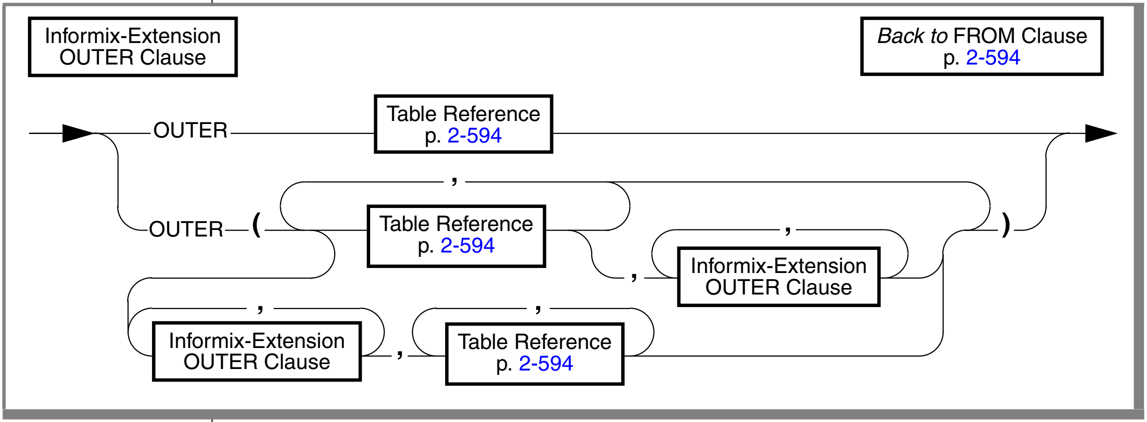
In this example, an inner join between the **catalog** and **stock** tables forms the dominant part of an outer join with the **items** table.

For additional examples of outer joins, see the *IBM Informix Guide to SQL: Tutorial*.

+

Informix-Extension Outer Joins

The Informix-extension syntax for outer joins begins with an implicit left outer join. That is, you begin an Informix-extension outer join with the OUTER keyword. This is the syntax of the Informix-extension OUTER clause.


Informix-Extension OUTER Clause
Back to FROM Clause p. 2-594
OUTER
Table Reference p. 2-594
OUTER
(
,
Table Reference p. 2-594
,
,
Informix-Extension OUTER Clause
)
,
Informix-Extension OUTER Clause
,
,
Table Reference p. 2-594

If you use this syntax for an outer join, you must use Informix-extension syntax for all outer joins in a single query block, and you must include the join condition in the WHERE clause. You cannot begin another outer join with the LEFT JOIN or the LEFT OUTER JOIN keywords.

This example uses the OUTER keyword to create an outer join that lists all customers and their orders, regardless of whether they have placed orders:

```
SELECT c.customer_num, c.lname, o.order_num FROM customer c,
   OUTER orders o WHERE c.customer_num = o.customer_num
```

This example returns all the rows from the **customer** table with the rows that match in the **orders** table. If no record for a customer appears in the **orders** table, the returned **order_num** column for that customer has a NULL value.

If you have a complex outer join, that is, the query has more than one outer join, you must either embed the additional outer join or joins in parentheses as the syntax diagram shows or establish join conditions, or relationships, between the dominant table and each subordinate table in the WHERE clause.

When an expression or a condition in the WHERE clause relates two subordinate tables, you must use parentheses around the joined tables in the FROM clause to enforce dominant-subordinate relationships, as in this example:

```
SELECT c.company, o.order_date, i.total_price, m.manu_name
   FROM customer c,
      OUTER (orders o, OUTER (items i, OUTER manufact m))
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
      AND i.manu_code = m.manu_code;
```

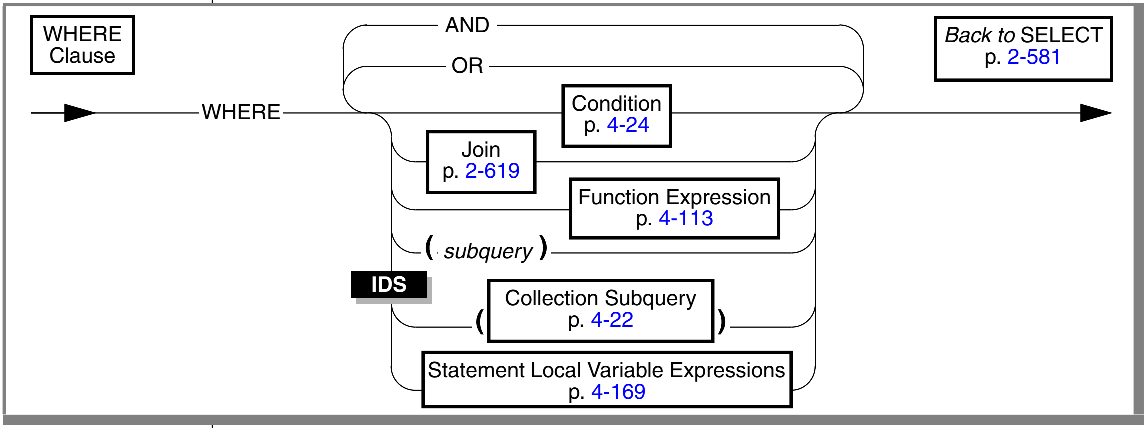
When you omit parentheses around the subordinate tables in the FROM clause, you must establish join conditions between the dominant table and each subordinate table in the WHERE clause. If a join condition is between two subordinate tables, the query will fail, but the following example successfully returns a result:

```
SELECT c.company, o.order_date, c2.call_descr
   FROM customer c, OUTER orders o, OUTER cust_calls c2
   WHERE c.customer_num = o.customer_num
      AND c.customer_num = c2.customer_num;
```

The *IBM Informix Guide to SQL: Tutorial*. has examples of complex outer joins.

WHERE Clause

The WHERE clause can specify join conditions for Informix-extension joins, post-join filters for ANSI-compliant joins, and search criteria on data values.


WHERE Clause
AND
OR
Back to SELECT p. 2-581
WHERE
Condition p. 4-24
Join p. 2-619
Function Expression p. 4-113
(subquery)
IDS
(Collection Subquery p. 4-22)
Statement Local Variable Expressions p. 4-169

Element	Purpose	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST or ORDER BY clause.	SELECT, p. 2-581

Using a Condition in the WHERE Clause

You can use these simple conditions or comparisons in the WHERE clause:

- Relational-operator condition
- IN or BETWEEN . . . AND
- IS NULL or IS NOT NULL
- LIKE or MATCHES

You also can use a SELECT statement within the WHERE clause; this is called a *subquery*. The following subquery WHERE clause operators are valid:

- IN or EXISTS
- ALL, ANY, or SOME

For more information, see "Condition" on page 4-24.

In the WHERE clause, an aggregate function is not valid unless it is part of a subquery, or if it is on a correlated column originating from a parent query, and the WHERE clause is in a subquery within a HAVING clause.

Relational-Operator Condition

A relational-operator condition is satisfied if the expressions on each side of the operator fulfill the relation that the operator specifies. The following SELECT statements use the greater than (>) and equal (=) relational operators:

```
SELECT order_num FROM orders
   WHERE order_date > '6/04/98'
SELECT fname, lname, company
   FROM customer
   WHERE city[1,3] = 'San'
```

Quotes are required around 'San' because the substring is from a character column. See the "Relational-Operator Condition" on page 4-28.

IN Condition

The IN condition is satisfied when the expression to the left of the IN keyword is included in the list of values to the right of the keyword.

The following examples show the IN condition:

```
SELECT lname, fname, company FROM customer
   WHERE state IN ('CA','WA', 'NJ')
SELECT * FROM cust_calls
   WHERE user_id NOT IN (USER )
```

For more information, see the "IN Subquery" on page 4-37.

BETWEEN Condition

The BETWEEN condition is satisfied when the value to the left of BETWEEN is in the inclusive range of the two values on the right of BETWEEN. The first two queries in the following example use literal values after the BETWEEN keyword. The third query uses the built-in CURRENT function and a literal interval to search for dates between the current day and seven days earlier.

```
SELECT stock_num, manu_code FROM stock
   WHERE unit_price BETWEEN 125.00 AND 200.00
SELECT DISTINCT customer_num, stock_num, manu_code
   FROM orders, items
   WHERE order_date BETWEEN '6/1/97' AND '9/1/97'
SELECT * FROM cust_calls WHERE call_dtime
   BETWEEN (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT
```

For more information, see the "BETWEEN Condition" on page 4-29.

IS NULL Condition

The IS NULL condition is satisfied if the column contains a NULL value. If you use the NOT option, the condition is satisfied when the column contains a value that is not NULL. The following example selects the order numbers and customer numbers for which the order has not been paid:

```
SELECT order_num, customer_num FROM orders
   WHERE paid_date IS NULL
```

For a complete description, see the "IS NULL Condition" on page 4-32.

LIKE or MATCHES Condition

The LIKE or MATCHES condition is satisfied if either of the following is true:

- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that the quoted string specifies. You can use wildcard characters in the string.
- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that is specified by the column that follows the LIKE or MATCHES keyword. The value of the column on the right serves as the matching pattern in the condition.

The following SELECT statement returns all rows in the **customer** table in which the **lname** column begins with the literal string `'Baxter'`. Because the string is a literal string, the condition is case sensitive.

```
SELECT * FROM customer WHERE lname LIKE 'Baxter%'
```

The next SELECT statement returns all rows in the **customer** table in which the value of the **lname** column matches the value of the **fname** column:

```
SELECT * FROM customer WHERE lname LIKE fname
```

The following examples use the LIKE condition with a wildcard. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain a percent (%) sign. Backslash (****) is used as the default escape character for the percent (%) sign wildcard. The third SELECT statement uses the ESCAPE option with the LIKE condition to retrieve rows from the **customer** table in which the **company** column includes a percent (%) sign. The `z` is used as an escape character for the percent (%) sign.

```
SELECT stock_num, manu_code FROM stock
   WHERE description LIKE '%ball'
SELECT * FROM customer WHERE company LIKE '%\%%'
SELECT * FROM customer WHERE company LIKE '%z%%' ESCAPE 'z'
```

The following examples use MATCHES with a wildcard in several SELECT statements. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain an asterisk (*). The backslash (\ ) is used as the default escape character for a literal asterisk (*) character. The third statement uses the ESCAPE option with the MATCHES condition to retrieve rows from the **customer** table where the **company** column includes an asterisk (*). The z character is specified as an escape character for the asterisk (*) character.

```
SELECT stock_num, manu_code FROM stock
   WHERE description MATCHES '*ball'

SELECT * FROM customer WHERE company MATCHES '*\**'

SELECT * FROM customer WHERE company MATCHES '*z**' ESCAPE 'z'
```

See also the "LIKE and MATCHES Condition" on page 4-32.

IN Subquery

With the IN subquery, more than one row can be returned, but only one column can be returned.

This example shows the use of an IN subquery in a SELECT statement:

```
SELECT DISTINCT customer_num FROM orders
   WHERE order_num NOT IN
      (SELECT order_num FROM items
         WHERE stock_num = 1)
```

For additional information, see the "IN Condition" on page 4-30.

EXISTS Subquery

With the EXISTS subquery, one or more columns can be returned.

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table).

It is appropriate to use an EXISTS subquery in this SELECT statement because you need the correlated subquery to test both **stock_num** and **manu_code** in the **items** table.

```
SELECT stock_num, manu_code FROM stock
   WHERE NOT EXISTS
      (SELECT stock_num, manu_code FROM items
         WHERE stock.stock_num = items.stock_num AND
            stock.manu_code = items.manu_code)
```

The preceding example would work equally well if you use a SELECT * in the subquery in place of the column names, because you are testing for the existence of a row or rows.

For additional information, see the "EXISTS Subquery" on page 4-38.

ALL, ANY, SOME Subqueries

The following examples return the order number of all orders that contain an item whose total price is greater than the total price of every item in order number 1023.

The first SELECT uses the ALL subquery, and the second SELECT produces the same result by using the MAX aggregate function.

```
SELECT DISTINCT order_num FROM items
   WHERE total_price > ALL (SELECT total_price FROM items
       WHERE order_num = 1023)

SELECT DISTINCT order_num FROM items
   WHERE total_price > SELECT MAX(total_price) FROM items
       WHERE order_num = 1023)
```

The following SELECT statements return the order number of all orders that contain an item whose total price is greater than the total price of at least one of the items in order number 1023. The first SELECT statement uses the ANY keyword, and the second SELECT statement uses the MIN aggregate function.

```
SELECT DISTINCT order_num FROM items
   WHERE total_price > ANY (SELECT total_price FROM items
       WHERE order_num = 1023)

SELECT DISTINCT order_num FROM items
   WHERE total_price > (SELECT MIN(total_price) FROM items
      WHERE order_num = 1023)
```

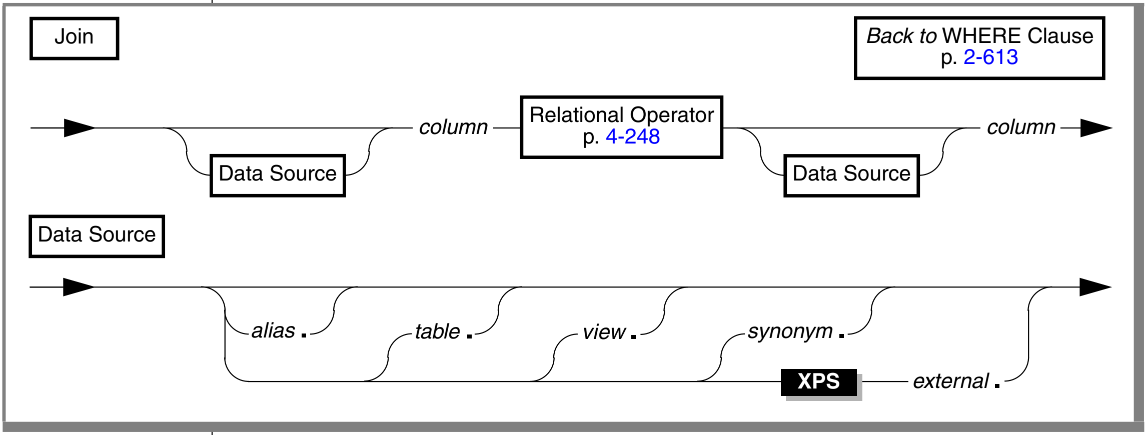
You can omit the keywords ANY, ALL, or SOME in a subquery if the subquery returns exactly one value. If you omit ANY, ALL, or SOME, and the subquery returns more than one value, you receive an error. The subquery in the next example returns only one row, because it uses an aggregate function:

```
SELECT order_num FROM items
   WHERE stock_num = 9 AND quantity =
      (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

See also "ALL, ANY, and SOME Subqueries" on page 4-39.

Specifying a Join in the WHERE Clause

You join two tables by creating a relationship in the WHERE clause between at least one column from one table and at least one column from another. The join creates a temporary composite table where each pair of rows (one from each table) that satisfies the join condition is linked to form a single row.



Element	Purpose	Restrictions	Syntax
alias	Temporary alternative name assigned to a table or view	See "Self-Joins" on page 2-620; "FROM Clause" on page 2-594	Identifier, p. 4-189
column	Column of a table or view to be joined	Must exist in the table or view	Identifier, p. 4-189
external	External table from which to retrieve data	External table must exist	Database Object Name, p. 4-46
synonym, table, view	Name of the synonym, table, or view to be joined	Synonym and the table or view to which it points must exist	Database Object Name, p. 4-46

Rows from the tables or views are *joined* when there is a match between the values of specified columns. When the columns to be joined have the same name, you must qualify each column name with its data source.

Two-Table Joins

You can create two-table joins, multiple-table joins, self-joins, and outer joins (Informix-extension syntax). The following example shows a two-table join:

```
SELECT order_num, lname, fname FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
```

You can omit the *column* if the tables are joined in the projection clause.

Multiple-Table Joins

A multiple-table join is a join of more than two tables. Its structure is similar to the structure of a two-table join, except that you have a join condition for more than one pair of tables in the WHERE clause. When columns from different tables have the same name, you must qualify the column name with its associated table or table alias, as in *table.column*. For the full syntax of a table name, see "Database Object Name" on page 4-46.

The following multiple-table join yields the company name of the customer who ordered an item as well as its stock number and manufacturer code:

```
SELECT DISTINCT company, stock_num, manu_code
   FROM customer c, orders o, items i
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
```

Self-Joins

You can join a table to itself. To do so, you must list the table name twice in the FROM clause and assign it two different table aliases. Use the aliases to refer to each of the *two* tables in the WHERE clause. The next example is a self-join on the **stock** table. It finds pairs of stock items whose unit prices differ by a factor greater than 2.5. The letters x and y are each aliases for the **stock** table.

```
SELECT x.stock_num, x.manu_code, y.stock_num, y.manu_code
   FROM stock x, stock y WHERE x.unit_price > 2.5 * y.unit_price
```

XPS

Extended Parallel Server does not support self-joins with an external table. ♦

Informix-Extension Outer Joins

The next outer join lists the company name of the customer and all associated order numbers, if the customer has placed an order. If not, the company name is still listed, and a NULL value is returned for the order number.

```
SELECT company, order_num FROM customer c, OUTER orders o
   WHERE c.customer_num = o.customer_num
```

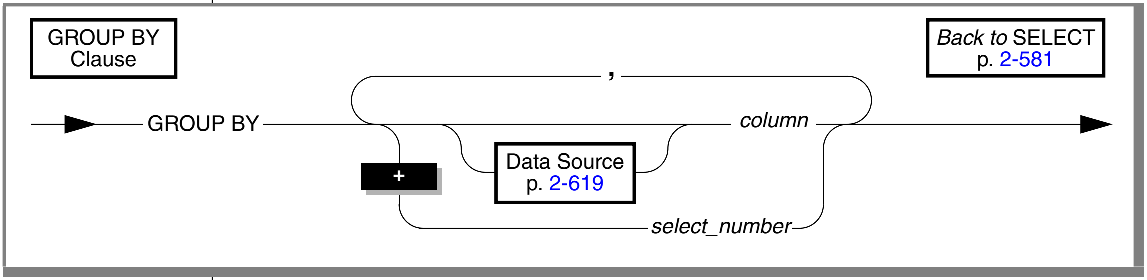
XPS

You cannot use an external table as the outer table in an outer join. ♦

For more information about outer joins, see the *IBM Informix Guide to SQL: Tutorial*.

GROUP BY Clause

Use the GROUP BY clause to produce a single row of results for each group. A *group* is a set of rows that have the same values for each column listed.



Element	Purpose	Restrictions	Syntax
column	A column (or an expression) from the select list of the Projection clause	See "Relationship of GROUP BY and Projection Clauses,"p. 2-622.	Identifier, p. 4-189
select_number	Integer specifying the ordinal position of a column or expression in the select list	See "Using Select Numbers" on page 2-623.	Literal Number, p. 4-216

The SELECT statement with a GROUP BY clause returns a single row of results for each group of rows that have the same value in *column*, or that have the same value in the column or expression that the *select_number* specifies.

Relationship of GROUP BY and Projection Clauses

A GROUP BY clause restricts what the Projection clause can specify. If you use a GROUP BY clause, each *column* specified in the select list of the Projection clause must also be included in the GROUP BY clause.

If you use an aggregate function and one or more column expressions in the select list, you must put all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause.

Constant expressions and BYTE or TEXT column expressions are not valid in the GROUP BY list.

If the select list includes a BYTE or TEXT column, you cannot use the GROUP BY clause. In addition, you cannot include ROWID in a GROUP BY clause.

IDS

If your select list includes a column of a user-defined data type, the column cannot be used in a GROUP BY clause unless the UDT can use the built-in bit-hashing function. Any UDT that cannot use the built-in bit-hashing function must be created with the CANNOTHASH modifier, which tells the database server that the UDT cannot be used in a GROUP BY clause. ♦

The following example names one column that is not in an aggregate expression. The **total_price** column should not be in the GROUP BY list because it appears as the argument of an aggregate function. The COUNT and SUM aggregates are applied to each group, not to the whole query set.

```
SELECT order_num, COUNT(*), SUM(total_price)
   FROM items GROUP BY order_num
```

If a column stands alone in a column expression in the select list, you must use it in the GROUP BY clause. If a column is combined with another column by an arithmetic operator, you can choose to group by the individual columns or by the combined expression using the number.

NULL Values in the GROUP BY Clause

In a column listed in a GROUP BY clause, each row that contains a NULL value belongs to a single group. That is, all NULL values are grouped together.

Using Select Numbers

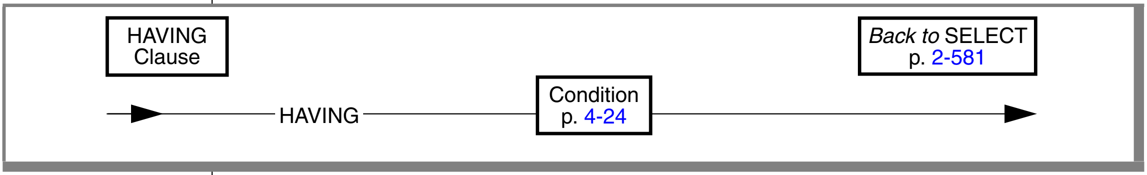
You can use one or more integers in the GROUP BY clause to stand for column expressions. In the next example, the first SELECT statement uses select numbers for **order_date** and **paid_date - order_date** in the GROUP BY clause. You can group only by a combined expression using the select numbers.

In the second SELECT statement, you cannot replace the `2` with the arithmetic expression **paid_date - order_date**.

```
SELECT order_date, COUNT(*), paid_date - order_date
   FROM orders GROUP BY 1, 3
SELECT order_date, paid_date - order_date
   FROM orders GROUP BY order_date, 2
```

HAVING Clause

Use the HAVING clause to apply one or more qualifying conditions to groups.



In the following examples, each condition compares one calculated property of the group with another calculated property of the group or with a constant. The first SELECT statement uses a HAVING clause that compares the calculated expression `COUNT(*)` with the constant `2`. The query returns the average total price per item on all orders that have more than two items.

The second SELECT statement lists customers and the call months for customers who have made two or more calls in the same month.

```
SELECT order_num, AVG(total_price) FROM items
   GROUP BY order_num HAVING COUNT(*) > 2
SELECT customer_num, EXTEND (call_dtime, MONTH TO MONTH)
   FROM cust_calls GROUP BY 1, 2 HAVING COUNT(*) > 1
```

You can use the HAVING clause to place conditions on the GROUP BY column values as well as on calculated values. This example returns **cust_code** and **customer_num**, **call_dtime**, and groups them by **call_code** for all calls that have been received from customers with **customer_num** less than 120:

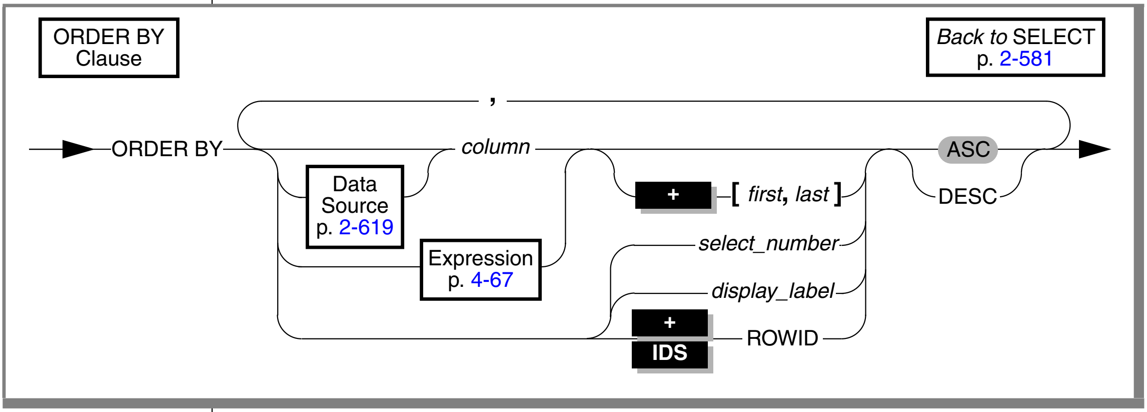
```
SELECT customer_num, EXTEND (call_dtime), call_code
   FROM cust_calls GROUP BY call_code, 2, 1
   HAVING customer_num < 120
```

The HAVING clause generally complements a GROUP BY clause. If you omit the GROUP BY clause, the HAVING clause applies to all rows that satisfy the query, and all rows in the table make up a single group. The following example returns the average price of all the values in the table, as long as more than ten rows are in the table:

```
SELECT AVG(total_price) FROM items HAVING COUNT(*) > 10
```

ORDER BY Clause

The ORDER BY clause sorts query results by specified columns or expressions.



Element	Purpose	Restrictions	Syntax
column	Sort rows by value in this column	Must also be in select list (XPS only)	Identifier, p. 4-189
display_label	Temporary name for a column	Must be declared in select list	Identifier, p. 4-189
first, last	Position of first and last character in substring to sort the query results	Integers; for BYTE, TEXT, and character data types only	Literal Number, p. 4-216
select_ number	Ordinal position of a column in select list of the Projection clause	See "Using Select Numbers" on page 623.	Literal Number, p. 4-216

To order query results by an expression, you must also declare a display label for the expression in the Projection clause, as in the following example:

```
SELECT paid_date - ship_date span, customer_num FROM orders
   ORDER BY span
```

XPS

For Extended Parallel Server, any *column* in the ORDER BY clause must also appear explicitly or by * notation in the select list of the Projection clause. To order query results by an expression calculated from column values, you must declare a display label for that expression in the select list of the Projection clause, and specify this label in the ORDER BY clause. ♦

The next query selects one column from the **orders** table and sorts the results by another. (With Extended Parallel Server, **order-date** must also appear in the Projection clause.) By default, the rows are listed in ascending order.

```
SELECT ship_date FROM orders ORDER BY order_date
```

You can order by an aggregate only if the query also has a GROUP BY clause. This query declares the display label **maxwgt** for use in the ORDER BY clause:

```
SELECT ship_charge, MAX(ship_weight) maxwgt
   FROM orders GROUP BY ship_charge ORDER BY maxwgt
```

GLS

If the current processing locale defines a localized collation, then NCHAR and NVARCHAR column values are sorted in that localized order. ♦

SPL

The ORDER BY clause is not valid within an SPL routine. ♦

IDS

No *column* in the ORDER BY clause can be of a collection data type.

Dynamic Server supports columns and expressions in the ORDER BY clause that do not appear in the select list of the Projection clause. You can omit a display label for the derived column in the select list and specify the derived column by means of a select number in the ORDER BY clause.

The select list of the Projection clause must include any column or expression that the ORDER BY clause specifies, however, if any of the following is true:

- The query includes the DISTINCT, UNIQUE, or UNION operator.
- The query includes the INTO TEMP *table* clause.

- The distributed query accesses a remote database whose server requires every column or expression in the ORDER BY clause to also appear in the select list of the Projection clause.
- An expression in the ORDER BY clause includes a display label for a column substring. (See "Ordering by a Substring" on page 2-626.) ♦

Ordering by a Substring

You can order by a substring instead of by the entire length of a character, BYTE, or TEXT column, or of an expression returning a character string. The substring is the portion used for the sort. You define the substring by specifying between brackets ([ ]) integer subscripts (the *first* and *last* parameters), representing the starting and ending character positions of the substring.

The following SELECT statement queries the **customer** table and specifies a column substring in the ORDER BY column. The column substring instructs the database server to sort the query results by the portion of the **lname** column contained in the sixth through ninth characters of the column.

```
SELECT * from customee ORDER BY lname[6,9]
```

Assume that the value of **lname** in one row of the **customer** table is `Greenburg`. Because of the column substring in the ORDER BY clause, the database server determines the sort position of this row by using the value `burg`, rather than the complete column value `Greenburg`.

When ordering by an expression, you can specify substrings only for expressions that return a character data type. If you specify a column substring in the ORDER BY clause, the column must have one of the following data types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.

IDS

Dynamic Server can also support LVARCHAR column substrings in the ORDER BY clause, if the column is in a database of the local database server. ♦

GLS

For information on the GLS aspects of using column substrings in the ORDER BY clause, see the *IBM Informix GLS User's Guide*. ♦

Ascending and Descending Orders

You can use the ASC and DESC keywords to specify ascending (smallest value first) or descending (largest value first) order. The default order is ascending. For DATE and DATETIME data types, *smallest* means earliest in time and *largest* means latest in time. For character data types in the default locale, the order is the ASCII collating sequence, as listed in "Collating Order for U.S. English Data" on page 4-250.

GLS

For NCHAR or NVARCHAR data types, the localized collating order of the current session is used, if that is different from the code set order. For more information about collation, see "SET COLLATION" on page 2-643. ♦

If you specify the ORDER BY clause, NULL values are ordered as less than values that are not NULL. Using the ASC order, a NULL value comes before any non-NULL value; using DESC order, the NULL comes last.

Nested Ordering

If you list more than one column in the ORDER BY clause, your query is ordered by a nested sort. The first level of sort is based on the first column; the second column determines the second level of sort. The following example of a nested sort selects all the rows in the **cust_calls** table and orders them by **call_code** and by **call_dtime** within **call_code**:

```
SELECT * FROM cust_calls ORDER BY call_code, call_dtime
```

Using Select Numbers

In place of column names, you can enter in the ORDER BY clause one or more integers that refer to the position of items listed in the select list of the Projection clause. You can also use a select number to sort by an expression.

The following example orders by the expression **paid_date - order_date** and **customer_num**, using select numbers in a nested sort:

```
SELECT order_num, customer_num, paid_date - order_date
   FROM orders
   ORDER BY 3, 2
```

Select numbers are required in the ORDER BY clause when SELECT statements are joined by the UNION or UNION ALL keywords, or when compatible columns in the same position have different names.

Ordering by Rowids

You can specify the ROWID keyword in the ORDER BY clause. This specifies the **rowid** column, a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The **rowid** column contains a unique internal record number that is associated with a row in a table. (It is recommended, however, that you utilize primary keys as your access method, rather than exploiting the **rowid** column.)

The ORDER BY clause cannot specify the **rowid** column if the table from which you are selecting is a fragmented table that has no **rowid** column.

XPS

In Extended Parallel Server, you cannot specify the ROWID keyword in the ORDER BY clause unless you also included ROWID in the Projection clause. ♦

IDS

In Dynamic Server, you do not need to include the ROWID keyword in the Projection clause when you specify ROWID in the ORDER BY clause. ♦

For further information on how to use the **rowid** column in column expressions, see "Expression" on page 4-67.

E/C

ORDER BY Clause with DECLARE

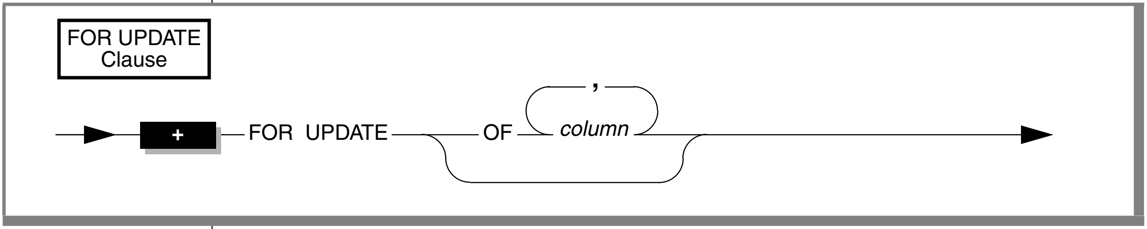
In ESQL/C, you cannot use a DECLARE statement with a FOR UPDATE clause to associate a cursor with a SELECT statement that has an ORDER BY clause.

Placing Indexes on ORDER BY Columns

When you include an ORDER BY clause in a SELECT statement, you can improve the performance of the query by creating an index on the column or columns that the ORDER BY clause specifies. The database server uses the index that you placed on the ORDER BY columns to sort the query results in the most efficient manner. For more information on how to create indexes that correspond to the columns of an ORDER BY clause, see "Using the ASC and DESC Sort-Order Options" on page 2-149.

FOR UPDATE Clause

Use the FOR UPDATE clause when you intend to update the values returned by a prepared SELECT statement when the values are fetched. Preparing a SELECT statement that contains a FOR UPDATE clause is equivalent to preparing the SELECT statement without the FOR UPDATE clause and then declaring a FOR UPDATE cursor for the prepared statement.



Element	Purpose	Restrictions	Syntax
column	Name of a column that can be updated after a FETCH	Must be in the FROM clause *table*, but it does not need to be in the select list of the Projection clause.	Identifier, p. 4-189

The FOR UPDATE keyword notifies the database server that updating is possible, causing it to use more stringent locking than it would with a select cursor. You cannot modify data through a cursor without this clause. You can specify which columns can be updated.

After you declare a cursor for a SELECT...FOR UPDATE statement, you can update or delete the currently selected row using an UPDATE OR DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they replace the usual test expressions in the WHERE clause.

To update rows with a specific value, your program might contain statements such as the sequence of statements that the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
    char fname[ 16];
    char lname[ 16];
    EXEC SQL END DECLARE SECTION;
. . .

 EXEC SQL connect to 'stores_demo';
 /* select statement being prepared contains a for update clause */
 EXEC SQL prepare x from 'select fname, lname from customer for update';
 EXEC SQL declare xc cursor for x;
```

```
for (;;)
  {
  EXEC SQL fetch xc into $fname, $lname;
  if (strncmp(SQLSTATE, '00', 2) != 0) break;
  printf("%d %s %s\n",cnum, fname, lname );
  if (cnum == 999)              --update rows with 999 customer_num
     EXEC SQL update customer set fname = 'rosey' where current of xc;
  }

EXEC SQL close xc;
EXEC SQL disconnect current;
```

A SELECT…FOR UPDATE statement, like an update cursor, allows you to perform updates that are not possible with the UPDATE statement alone, because both the decision to update and the values of the new data items can be based on the original contents of the row. The UPDATE statement cannot query the table that is being updated.

Syntax That is Incompatible with the FOR UPDATE Clause

A SELECT statement that uses a FOR UPDATE clause must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot include any of the following clauses or keywords: DISTINCT, FOR READ ONLY, GROUP BY, INTO SCRATCH, INTO TEMP, INTO EXTERNAL, ORDER BY, UNION, or UNIQUE.

For information on how to declare an update cursor for a SELECT statement that does not include a FOR UPDATE clause, see "Using the FOR UPDATE Option" on page 2-328.

FOR READ ONLY Clause

Use the FOR READ ONLY keywords to specify that the select cursor declared for the SELECT statement is a read-only cursor. A read-only cursor is a cursor that cannot modify data. This section provides the following information about the FOR READ ONLY clause:

- When you must use the FOR READ ONLY clause
- Syntax restrictions on a SELECT statement that uses a FOR READ ONLY clause

Using the FOR READ ONLY Clause in Read-Only Mode

Normally, you do not need to include the FOR READ ONLY clause in a SELECT statement. SELECT is a read-only operation by definition, so the FOR READ ONLY clause is usually unnecessary. In certain circumstances, however, you must include the FOR READ ONLY keywords in a SELECT statement.

ANSI

If you have used the High-Performance Loader (HPL) in express mode to load data into the tables of an ANSI-compliant database, and you have not yet performed a level-0 backup of this data, the database is in read-only mode. When the database is in read-only mode, the database server rejects any attempts by a select cursor to access the data unless the SELECT or the DECLARE includes a FOR READ ONLY clause. This restriction remains in effect until the user has performed a level-0 backup of the data.

In an ANSI-compliant database, select cursors are update cursors by default. An update cursor is a cursor that can be used to modify data. These update cursors are incompatible with the read-only mode of the database. For example, this SELECT statement against the **customer_ansi** table fails:

```
EXEC SQL declare ansi_curs cursor for
   select * from customer_ansi;
```

The solution is to include the FOR READ ONLY clause in your select cursors. The read-only cursor that this clause specifies is compatible with the read-only mode of the database. For example, the following SELECT FOR READ ONLY statement against the **customer_ansi** table succeeds:

```
EXEC SQL declare ansi_read cursor for
   select * from customer_ansi for read only;
```

♦

DB

DB-Access executes all SELECT statements with select cursors, so you must specify FOR READ ONLY in all queries that access data in a read-only ANSI-compliant database. The FOR READ ONLY clause causes DB-Access to declare the cursor for the SELECT statement as a read-only cursor. ♦

For more information on level-0 backups, see your *IBM Informix Backup and Restore Guide*. For more information on select cursors, read-only cursors, and update cursors, see "DECLARE" on page 2-323.

IDS

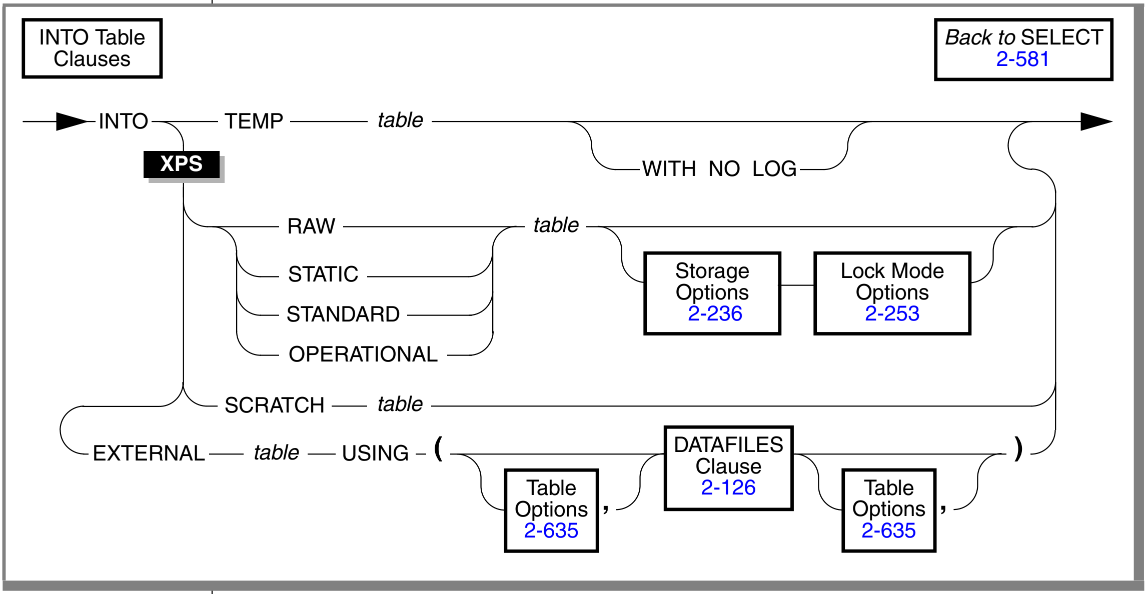
For more information on the express mode of the HPL, see the *IBM Informix High-Performance Loader User's Guide*. ♦

Syntax That Is Incompatible with the FOR READ ONLY Clause

If you attempt to include both the FOR READ ONLY clause and the FOR UPDATE clause in the same SELECT statement, the SELECT statement fails. For information on declaring a read-only cursor for a SELECT statement that does not include a FOR READ ONLY clause, see "DECLARE" on page 2-323.

INTO Table Clauses

Use the INTO Table clauses to specify a temporary or external table to receive the data that the SELECT statement retrieves.



Element	Purpose	Restrictions	Syntax
table	Table to receive the results of the query	Must be unique among names of tables, views, and synonyms that you own in the current database	Database Object Name, p. 4-46

You must have the Connect privilege on a database to create a temporary or external table in that database. The name of a temporary table need not be unique among the names of temporary tables of other users.

Column names in the temporary or external table must be specified in the Projection clause, where you must supply a display label for all expressions that are not simple column expressions. The display label becomes the column name in the temporary or external table. If you do not declare a display label for a column expression, the table uses the column name from the select list of the Projection clause.

The following INTO TEMP example creates the **pushdate** table with two columns, **customer_num** and **slowdate**:

```
SELECT customer_num, call_dtime + 5 UNITS DAY slowdate
   FROM cust_calls INTO TEMP pushdate
```

Results When No Rows are Returned

When you use an INTO Table clause combined with the WHERE clause, and no rows are returned, the **SQLNOTFOUND** value is `100` in ANSI-compliant databases and `0` in databases that are not ANSI compliant. If the SELECT INTO TEMP…WHERE… statement is a part of a multistatement PREPARE and no rows are returned, the **SQLNOTFOUND** value is `100` for both ANSI-compliant databases and databases that are not ANSI-compliant.

E/C

Restrictions with INTO Table Clauses in ESQL/C

In ESQL/C, do not use the INTO clause with an INTO *table* clause. If you do, no results are returned to the program variables and the **sqlca.sqlcode, SQLCODE** variable is set to a negative value.

INTO TEMP Clause

The INTO TEMP clause creates a temporary table to hold the query results. The default initial and next extents for a temporary table are four pages. The temporary table must be accessible by the built-in RSAM access method of the database server; you cannot specify another access method.

If you use the same query results more than once, using a temporary table saves time. In addition, using an INTO TEMP clause often gives you clearer and more understandable SELECT statements.

Data values in a temporary table are static; they are not updated as changes are made to the tables that were used to build the temporary table. You can use the CREATE INDEX statement to create indexes on a temporary table.

A logged, temporary table exists until one of the following situations occurs:

- The application disconnects.
- A DROP TABLE statement is issued on the temporary table.
- The database is closed.

IDS

If your database does not have transaction logging, the temporary table behaves in the same way as a table created with the WITH NO LOG option. ♦

If you define more than one temporary dbspace, the INTO TEMP clause loads rows from query results into each of the dbspaces in round-robin fashion.

Using the WITH NO LOG Option

Use the WITH NO LOG option to reduce the overhead of transaction logging because operations on nonlogging temporary tables are not logged.

A nonlogging temporary table exists until one of the following events occurs:

- The application disconnects.
- A DROP TABLE statement is issued on the temporary table.

Because nonlogging temporary tables do not disappear when the database is closed, you can use a nonlogging temporary table to transfer data from one database to another while the application remains connected. The behavior of a temporary table that you create with the WITH NO LOG option of the INTO TEMP clause is the same as that of a scratch table.

For more information about temporary tables, see "CREATE Temporary TABLE" on page 2-261.

XPS

Using SELECT INTO to Create a New Permanent Table

Using Extended Parallel Server, you can use the SELECT INTO statement to create a new permanent table based on the result set of a SELECT statement.

When using SELECT INTO to create a new table, you must specify its type. You can optionally specify storage and lock mode options for the new table.

The column names of the new permanent table are the names specified in the select list. If a "*" appears in the select list, it will be expanded to the columns of the corresponding tables or views in the SELECT statement.

All expressions other than simple column expressions must have a display label. This is used as the name of the column in the new table. If a column expression has no display label, the table uses the column name. If there are duplicate labels or column names in the select list, an error will be returned.

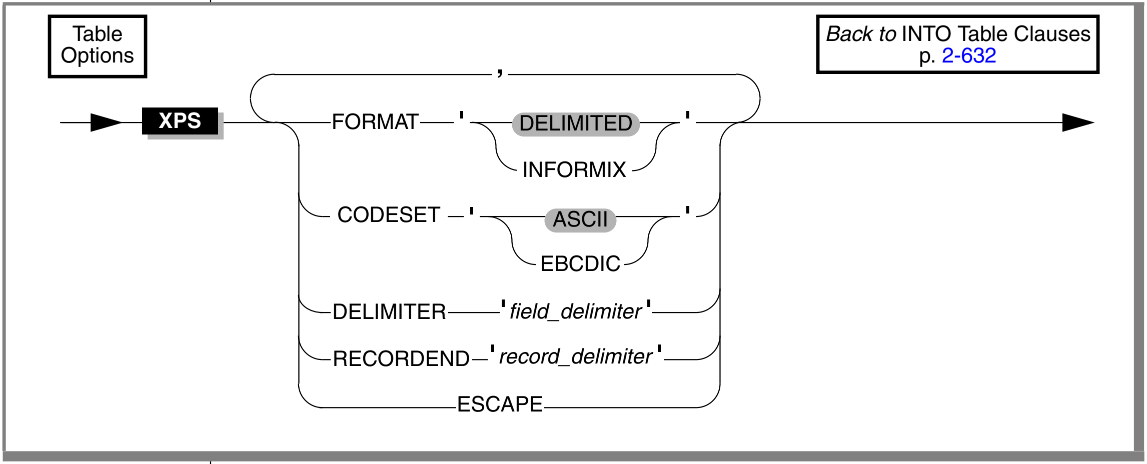
XPS

INTO EXTERNAL Clause

The INTO EXTERNAL clause unloads query results into an external table, creating a default external table description that you can use when you subsequently reload the files. This combines the functionality of the CREATE EXTERNAL TABLE . . . SAMEAS and INSERT INTO . . . SELECT statements.

Table Options

The SELECT statement supports a subset of the CREATE EXTERNAL TABLE syntax for "Table Options" on page 2-128. Use the Table Options clause of the SELECT INTO EXTERNAL statement to specify the format of the unloaded data in the external table.



Element	Purpose	Restrictions	Syntax
field_delimiter	Character to separate fields. Default is the pipe (|) character	See "Specifying Delimiters" on page 636.	Quoted String, p. 4-243
record_delimiter	Character to separate records	See "Specifying Delimiters" on page 636.	Quoted String, p. 4-243

The following table describes the keywords that apply to unloading data. If you want to specify additional table options in the external-table description for the purpose of reloading the table later, see "Table Options" on page 2-128.

In the SELECT...INTO EXTERNAL statement, you can specify all table options that are discussed in the CREATE EXTERNAL TABLE statement except the fixed-format option.

You can use the INTO EXTERNAL clause when the format type of the created data file is either delimited text (if you use the DELIMITED keyword) or text in Informix internal data format (if you use the INFORMIX keyword). You cannot use it for a fixed-format unload.

Keyword	Purpose
CODESET	Specifies the type of code set. Options are EBCDIC or ASCII.
DELIMITER	Specifies the character that separates fields in a delimited text file
ESCAPE	Directs the database server to recognize ASCII special characters embedded in ASCII-text-based data files If you do not specify ESCAPE when you load data, the database server does not check the character fields in text data files for embedded special characters. If you do not specify ESCAPE when you unload data, the database server does not create embedded hexadecimal characters in text fields.
FORMAT	Specifies the format of the data in the data files
RECORDEND	Specifies the character that separates records in a delimited text file

Specifying Delimiters

If you do not set the **RECORDEND** environment variable, the default value for *record_delimiter* is the newline character (CTRL-n).

If you use a non-printing character as a delimiter, encode it as the octal representation of the ASCII character. For example, '\006' can represent CTRL-F.

For more information on external tables, see "CREATE EXTERNAL TABLE" on page 2-121.

XPS

INTO SCRATCH Clause

Extended Parallel Server supports the INTO SCRATCH clause. This can improve performance, because scratch tables are not logged. A scratch table does not support indexes or constraints. It persists until the application disconnects, or a DROP TABLE statement is issued on the temporary table.

Because scratch tables do not disappear when the database is closed, you can use a scratch table to transfer data from one database to another while the application remains connected. A scratch table is identical to a temporary table that is created with the WITH NO LOG option. For more information about scratch tables, see "CREATE Temporary TABLE" on page 2-261.

UNION Operator

Place the UNION operator between two SELECT statements to combine the queries into a single query. You can string several SELECT statements together using the UNION operator. Corresponding items do not need to have the same name. Omitting the ALL keyword excludes duplicate rows.

Restrictions on a Combined SELECT

Several restrictions apply on the queries that you can connect with a UNION operator, as the following list describes:

E/C

- In ESQL/C, you cannot use an INTO clause in a compound query unless exactly one row is returned, and you are not using a cursor. In this case, the INTO clause must be in the first SELECT statement. ♦
- The number of items in the Projection clause of each query must be the same, and the corresponding items in each Projection clause must have compatible data types.
- The Projection clause of each query cannot specify BYTE or TEXT columns. (This restriction does not apply to UNION ALL operations.)
- If you use an ORDER BY clause, it must follow the last Projection clause, and you must refer to the item ordered by integer, not by identifier. Sorting takes place after the set operation is complete.

You can store the combined results of a UNION operator in a temporary table, but the INTO TEMP clause can appear only in the final SELECT statement.

Duplicate Rows in a Combined SELECT

If you use the UNION operator alone, the duplicate rows are removed from the complete set of rows. That is, if multiple rows contain identical values in each column, only one row is retained. If you use the UNION ALL operator, all the selected rows are returned (the duplicates are not removed).

The next example uses UNION ALL to join two SELECT statements without removing duplicates. The query returns a list of all the calls that were received during the first quarter of 1997 and the first quarter of 1998.

```
SELECT customer_num, call_code FROM cust_calls
   WHERE call_dtime BETWEEN
         DATETIME (1997-1-1) YEAR TO DAY
      AND DATETIME (1997-3-31) YEAR TO DAY
UNION ALL
SELECT customer_num, call_code FROM cust_calls
   WHERE call_dtime BETWEEN
         DATETIME (1998-1-1)YEAR TO DAY
      AND DATETIME (1998-3-31) YEAR TO DAY
```

If you want to remove duplicates, use the UNION operator without the keyword ALL in the query. In the preceding example, if the combination `101 B` were returned in both SELECT statements, a UNION operator would cause the combination to be listed once. (If you want to remove duplicates within each SELECT statement, use the DISTINCT keyword in the Projection clause, as described in "Projection Clause" on page 2-583.)

IDS

UNION in Subqueries

You can use the UNION and UNION ALL operators in subqueries of SELECT statements within the WHERE clause, the FROM clause, and in collection subqueries. In this release of Dynamic Server, however, subqueries that include UNION or UNION ALL are not supported in the following contexts:

- In the definition of a view
- In the event or in the Action clause of a trigger
- With the FOR UPDATE clause or with an Update cursor
- In a distributed query (accessing tables outside the local database)

For more information about collection subqueries, see "Collection Subquery" on page 4-22. For more information about the FOR UPDATE clause, see "FOR UPDATE Clause" on page 2-629.

In a combined subquery, the database server can resolve a column name only within the scope of its qualifying table reference. The following query, for example, returns an error:

```
SELECT * FROM t1 WHERE EXISTS
   (SELECT a FROM t2
   UNION
   SELECT b FROM t3 WHERE t3.c IN
      (SELECT t4.x FROM t4 WHERE t4.4 = t2.z))
```

Here **t2.z** in the innermost subquery cannot be resolved, because **z** occurs outside the scope of reference of the table reference **t2**. Only column references that belong to **t4**, **t3**, or **t1** can be resolved in the innermost subquery. The scope of a table reference extends downwards through subqueries, but not across the UNION operator to sibling SELECT statements.

Related Information

For task-oriented discussions of the SELECT statement, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion of the GLS aspects of the SELECT statement, see the *IBM Informix GLS User's Guide*.

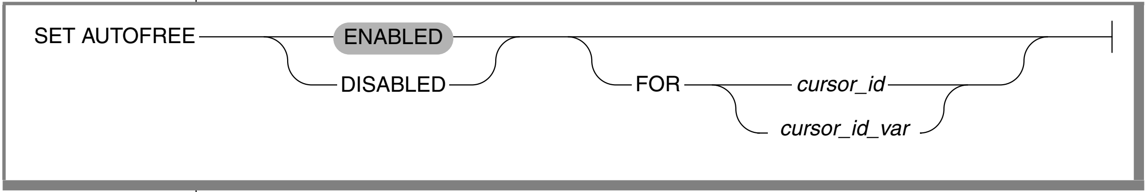
For information on how to access row and collections with ESQL/C host variables, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

+
IDS
E/C

SET AUTOFREE

Use the SET AUTOFREE statement to instruct the database server to enable or disable a memory-management feature that can free the memory allocated for a cursor automatically, as soon as the cursor is closed. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor for which Autofree is to be reset	Must already be declared within the program.	Identifier, p. 4-189
cursor_id_var	Host variable that holds the value of *cursor_id*	Must store a *cursor_id* already declared in the program.	Must conform to language-specific rules for names.

Usage

When the Autofree feature is enabled for a cursor, and the cursor is subsequently closed, you do not need to explicitly use the FREE statement to release the memory that the database server allocated for the cursor. If you issue SET AUTOFREE but specify no option, the default is ENABLED.

The SET AUTOFREE statement that enables the Autofree feature must appear before the OPEN statement that opens a cursor. The SET AUTOFREE statement does not affect the memory allocated to a cursor that is already open. After a cursor is Autofree enabled, you cannot open that cursor a second time.

Globally Affecting Cursors with SET AUTOFREE

If you include no FOR *cursor _id* or FOR *cursor_id_var* clause, then the scope of SET AUTOFREE is all subsequently-declared cursors in the program (or more precisely, all cursors declared before a subsequent SET AUTOFREE statement with no FOR clause globally resets the Autofree feature). This example enables the Autofree feature for all subsequent cursors in the program:

```
EXEC SQL set autofree;
```

The next example disables the Autofree feature for all subsequent cursors:

```
EXEC SQL set autofree disabled;
```

Using the FOR Clause to Specify a Specific Cursor

If you specify FOR *cursor _id* or FOR *cursor_id_var*, then SET AUTOFREE affects only the cursor that you specify after the FOR keyword.

This option allows you to override a global setting for all cursors. For example, if you issue a SET AUTOFREE ENABLED statement for all cursors in a program, you can issue a subsequent SET AUTOFREE DISABLED FOR statement to disable the Autofree feature for a specific cursor.

In the following example, the first statement enables the Autofree feature for all cursors, while the second statement disables the Autofree feature for the cursor named **x1**:

```
EXEC SQL set autofree enabled;
EXEC SQL set autofree disabled for x1;
```

Here the **x1** cursor must have been declared but not yet opened.

Associated and Detached Statements

When a cursor is automatically freed, its associated prepared statement (or associated statement) is also freed.

The term *associated statement* has a special meaning in the context of the Autofree feature. A cursor is associated with a prepared statement if it is the first cursor that you declare with the prepared statement, or if it is the first cursor that you declare with the statement after the statement is detached.

The term *detached statement* has a special meaning in the context of the Autofree feature. A prepared statement is detached if you do not declare a cursor with the statement, or if the cursor with which the statement is associated was freed.

If the Autofree feature is enabled for a cursor that has an associated prepared statement, and that cursor is closed, the database server frees the memory allocated to the prepared statement as well as the memory allocated for the cursor. Suppose that you enable the Autofree feature for the following cursor:

```
/*Cursor associated with a prepared statement */
EXEC SQL prepare sel_stmt 'select * from customer';
EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

When the database server closes the **sel_curs2** cursor, it automatically performs the equivalent of the following FREE statements:

```
FREE sel_curs2;
FREE sel_stmt;
```

Because memory for the **sel_stmt** statement is freed automatically, you cannot declare a new cursor on it unless you prepare the statement again.

Closing Cursors Implicitly

A potential problem exists with cursors that have the Autofree feature enabled. In a database that is not ANSI-compliant, if you do not close a cursor explicitly and then open it again, the cursor is closed implicitly. This implicit closing of the cursor triggers the Autofree feature. The second time the cursor is opened, the database server generates an error message (`cursor not found`) because the cursor is already freed.

Related Information

Related statements: CLOSE, DECLARE, FETCH, FREE, OPEN, and PREPARE

For more information on the Autofree feature, see the *IBM Informix ESQL/C Programmer's Manual*.

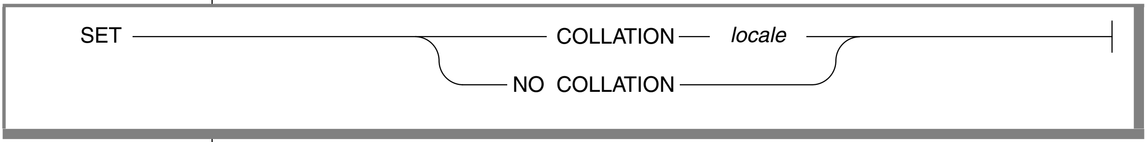
+

IDS

SET COLLATION

Use the SET COLLATION statement to specify a new collating order for the session, superseding the **DB_LOCALE** environment variable. SET NO COLLATION restores the default collation. Use these statements with ESQL/C.

Syntax



Element	**Purpose**	**Restrictions**	**Syntax**
locale	Name of a locale whose collating order is to be used in this session	Must be the name of a locale that the database server can access	Quoted String, p. 4-243

Usage

As the *IBM Informix GLS User's Guide* explains, the database server uses locale files to specify the character set, the collating order, and other conventions of some natural language to display and manipulate character strings and other data values. The *collating order* of the database locale is the sequential order in which the database server sorts character strings.

If you set no value for **DB_LOCALE**, the default locale, based on United States English, is **en_us.8859-1** for UNIX, or Code Page 1252 for Windows systems. Otherwise, the database server uses the **DB_LOCALE** setting as its locale. SET COLLATION overrides the collating order of **DB_LOCALE** at runtime for all database servers previously accessed in the same session.

The new collating order remains in effect for the rest of the session, or until you issue another SET COLLATION statement. Other sessions are not affected, but database objects that you created with a non-default collation will use whatever collating order was in effect at their time of their creation.

By default, the collating order is the code set order, but some locales also support a *locale-specific* order. In most contexts. only NCHAR and NVARCHAR data values can be sorted according to a locale-specific collating order.

Specifying a Collating Order with SET COLLATION

SET COLLATION replaces the current collating order with that of the specified *locale* for all database servers previously accessed in the current session. For example, this specifies the collating order of the German language:

```
EXEC SQL set collation "de_de.8859-1";
```

If the next action of a database server in this session sorted NCHAR or NVARCHAR values, this would follow the German collating order.

Suppose that, in the same session, the following SET NO COLLATION statement restores the **DB_LOCALE** setting for the collating order:

```
EXEC SQL set no collation;
```

After SET NO COLLATION executes, subsequent collation in the same session is based on the **DB_LOCALE** setting. Any database objects that you created using the German collating order, however, such as check constraints, indexes, prepared objects, triggers, or UDRs, will continue to apply German collation to NCHAR and NVARCHAR data types.

Restrictions on SET COLLATION

Although SET COLLATION enables you to change the collating order of the database server dynamically within a session, you should be aware of these limitations on the effects of the SET COLLATION statement.

- Only collation performed by the database server are affected. Client processes that sort data are not affected by SET COLLATION.
- Only the current session is affected. Other sessions are not affected directly by your SET COLLATION statements (but any database objects that you create will sort in their creation-time collating order).
- Changing the collating order does not change the code set. The database server always uses the code set specified by **DB_LOCALE**.
- Only NCHAR and NVARCHAR values sort in locale-specific order

Because SET COLLATION changes only the collating order, rather than the current locale or code set, you cannot use this statement to insert character data from different locales, such as French and Japanese, into the same database. You must use Unicode if the database needs to store characters from two or more languages that require inherently different code sets.

Collation Performed by Database Objects

Although the database reverts to the **DB_LOCALE** collating order after the session ends (or after you execute SET NO COLLATION), indexes that you create using a non-default collating sequence persist in the database. You can create multiple indexes on the same set of columns, called *multilingual indexes*, using different collating orders that SET COLLATION specifies.

Only one clustered index, however, can exist on a given set of columns.

Only one unique constraint or primary key can exist on a given set of columns, but you can create multiple unique indexes on the same set of columns, if each index has a different collation order.

The query optimizer ignores indexes that apply any collation other than the current session collation to NCHAR or NVARCHAR columns when calculating the cost of a query.

The collating order of an attached index must be the same as that of its table, and this must be the default collating order specified by **DB_LOCALE**.

The ALTER INDEX statement cannot change the collation of an index. Any previous SET COLLATION statement is ignored when ALTER INDEX executes.

You can use SET COLLATION with CREATE TABLE or ALTER TABLE to create columns with different collations. (Comparing such columns might not provide meaningful results, because they might be in different languages.)

When synonyms are created for remote tables or views, the participating databases must have the same collating order. Existing synonym, however, can be used in other databases that support SET COLLATION and the collating order of the synonym, regardless of the **DB_LOCALE** setting.

Check constraints, cursors, prepared objects, triggers, and SPL routines that sort NCHAR or NVARCHAR values use the collation that was in effect at the time of their creation, if this is different from the **DB_LOCALE** setting.

The effect on performance is sensitive to to how many different collations are used when creating database objects that sort in a localized order.

Related Information

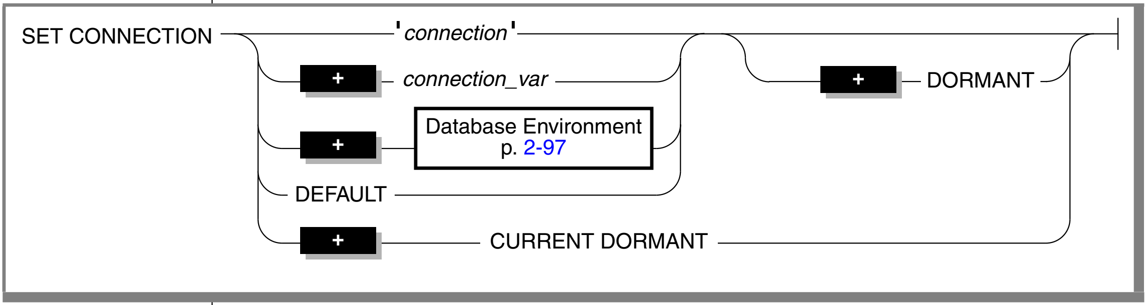
For information on locales, see the *IBM Informix GLS User's Guide*.

E/C

SET CONNECTION

Use the SET CONNECTION statement to reestablish a connection between an application and a database environment and make the connection current. You can also use this statement with the DORMANT option to put the current connection in a dormant state. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
connection	Name that you declared for the initial connection that the CONNECT statement made	The database must already exist.	Quoted String, p. 4-243
connection_var	Host variable that contains the value of *connection*	Must be of a character data type.	Language specific

Usage

You can use SET CONNECTION to make a dormant connection current, or to make the current connection dormant.

SET CONNECTION is not valid as a prepared statement.

Making a Dormant Connection the Current Connection

If you use the SET CONNECTION statement without the DORMANT option, *connection* must represent a dormant connection. A *dormant connection* is a connection that is established but that is not current.

The SET CONNECTION statement, with no DORMANT option, makes the specified dormant connection the current one. The connection that the application specifies must be dormant. The connection that is current when the statement executes becomes dormant.

The SET CONNECTION statement in the following example makes connection `con1` the current connection and makes `con2` a dormant connection:

```
CONNECT TO 'stores_demo' AS 'con1'
...
CONNECT TO 'demo' AS 'con2'
...
SET CONNECTION 'con1'
```

A dormant connection has a *connection context* associated with it. When an application makes a dormant connection current, it reestablishes that connection to a database environment and restores its connection context. (For more information on connection context, see the CONNECT statement on page 2-92.) Reestablishing a connection is comparable to establishing the initial connection, except that it typically avoids authenticating the permissions for the user again, and it saves reallocating resources associated with the initial connection. For example, the application does not need to reprepare any statements that have previously been prepared in the connection, nor does it need to redeclare any cursors.

Making a Current Connection the Dormant Connection

In the SET CONNECTION *connection* DORMANT statement, *connection* must represent the current connection. The SET CONNECTION statement with the DORMANT option makes the specified current connection a dormant connection. For example, the following SET CONNECTION statement makes connection `con1` dormant:

```
SET CONNECTION 'con1' DORMANT
```

The SET CONNECTION statement with the DORMANT option generates an error if you specify a connection that is already dormant. For example, if connection `con1` is current and connection `con2` is dormant, the following SET CONNECTION statement returns an error message:

```
SET CONNECTION 'con2' DORMANT
```

The following SET CONNECTION statement executes successfully:

```
SET CONNECTION 'con1' DORMANT
```

Dormant Connections in a Single-Threaded Environment

In a single-threaded ESQL/C application (one that does not use threads), the DORMANT option makes the current connection dormant. Using this option makes single-threaded ESQL/C applications upwardly compatible with thread-safe ESQL/C applications. A single-threaded environment, however, can have only one active connection while the program executes.

Dormant Connections in a Thread-Safe Environment

In a thread-safe ESQL/C application, the DORMANT option makes an active connection dormant. Another thread can now use the connection by issuing the SET CONNECTION statement without the DORMANT option. A thread-safe environment can have many threads (concurrent pieces of work performing particular tasks) in one ESQL/C application, and each thread can have one active connection.

An active connection is associated with a particular thread. Two threads cannot share the same active connection. Once a thread makes an active connection dormant, that connection is available to other threads. A dormant connection is still established but is not currently associated with any thread. For example, if the connection named `con1` is active in the thread named `thread_1`, the thread named `thread_2` cannot make connection `con1` its active connection until `thread_1` has made connection `con1` dormant.

The following code fragment from a thread-safe ESQL/C program shows how a particular thread within a thread-safe application makes a connection active, performs work on a table through this connection, and then makes the connection dormant so that other threads can use the connection:

```
thread_2()
{  /* Make con2 an active connection */
   EXEC SQL connect to 'db2' as 'con2';
   /*Do insert on table t2 in db2*/
   EXEC SQL insert into table t2 values(10);
   /* make con2 available to other threads */
   EXEC SQL set connection 'con2' dormant;
}
```

If a connection to a database environment was initiated using the CONNECT . . . WITH CONCURRENT TRANSACTION statement, any thread that subsequently connects to that database environment can use an ongoing transaction. In addition, if an open cursor is associated with such a connection, the cursor remains open when the connection is made dormant.

Threads within a thread-safe ESQL/C application can use the same cursor by making the associated connection current, even though only one thread can use the connection at any given time.

Identifying the Connection

If the application did not use a connection name in the initial CONNECT statement, you must use a database environment (such as a database name or a database pathname) as the connection name. For example, the following SET CONNECTION statement uses a database environment for the connection name because the CONNECT statement does not use a connection name. For information about quoted strings that contain a database environment, see "Database Environment" on page 2-97.

```
CONNECT TO 'stores_demo'
...
CONNECT TO 'demo'
...
SET CONNECTION 'stores_demo'
```

If a connection to a database server was assigned a connection name, however, you must use the connection name to reconnect to the database server. An error is returned if you use a database environment rather than the connection name when a connection name exists.

DEFAULT Option

The DEFAULT option specifies the default connection for a SET CONNECTION statement. The default connection is one of the following connections:

- An explicit default connection (a connection established with the CONNECT TO DEFAULT statement)
- An implicit default connection (any connection established with the DATABASE or CREATE DATABASE statements)

Use SET CONNECTION without a DORMANT option to reestablish the default connection, or with that option to make the default connection dormant.

For more information, see "DEFAULT Option" on page 2-94 and "The Implicit Connection with DATABASE Statements" on page 2-94.

CURRENT Keyword

Use the CURRENT keyword with the DORMANT option of the SET CONNECTION statement as a shorthand form of identifying the current connection. The CURRENT keyword replaces the current connection name. If the current connection is `con1`, the following two statements are equivalent:

```
SET CONNECTION 'con1' DORMANT;

SET CONNECTION CURRENT DORMANT;
```

When a Transaction is Active

Without the DORMANT keyword, the SET CONNECTION statement implicitly puts the current connection in the dormant state.

When you issue a SET CONNECTION statement with the DORMANT keyword, the SET CONNECTION statement explicitly puts the current connection in the dormant state. In both cases, the statement can fail if a connection that becomes dormant has an uncommitted transaction. If the connection that becomes dormant has an uncommitted transaction, the following conditions apply:

- If the connection was established using the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, SET CONNECTION succeeds and puts the connection in a dormant state.
- If the connection was not established by the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, SET CONNECTION fails and cannot set the connection to a dormant state and the transaction in the current connection continues to be active. The statement generates an error and the application must decide whether to commit or roll back the active transaction.

Related Information

Related statements: CONNECT, DISCONNECT, and DATABASE

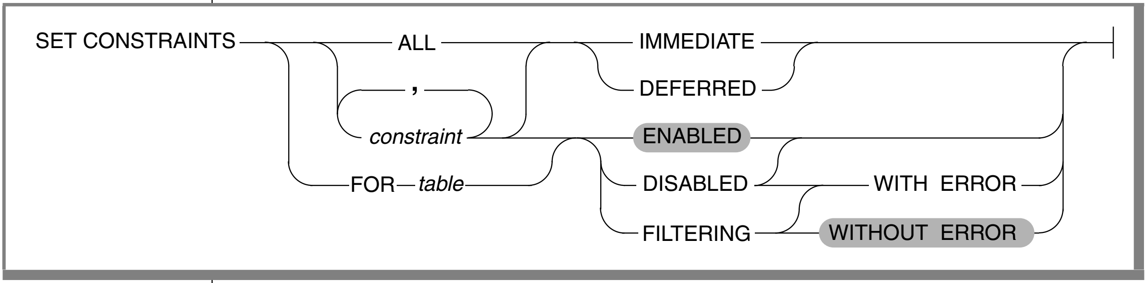
For a discussion of the SET CONNECTION statement and thread-safe applications, see the *IBM Informix ESQL/C Programmer's Manual*.

SET CONSTRAINTS

Use the SET CONSTRAINTS statements to specify how some or all of the constraints on a table are processed. Constraint-mode options include these:

- Whether constraints are checked at the statement level (IMMEDIATE) or at the transaction level (DEFERRRED)
- Whether to enable or disable constraints
- Whether to change the filtering mode of constraints.

Syntax



Element	Purpose	Restrictions	Syntax
constraint	Constraint whose mode is to be reset	All constraints must exist and must all be defined on the same table.	Database Object Name, p. 4-46
table	Table whose constraint mode is to be reset for all constraints	Table must exist in the database	Database Object Name, p. 4-46

Usage

The SET CONSTRAINTS keywords begin the SET Transaction Mode statement, which is described in "SET Transaction Mode" on page 2-725.

The SET CONSTRAINTS keywords can also begin a special case of the SET Database Object Mode statement. The SET Database Object Mode statement can also enable or disable a trigger or index, or change the filtering mode of a unique index. For the complete syntax and semantics of the SET INDEX statement, see "SET Database Object Mode" on page 2-652.

+

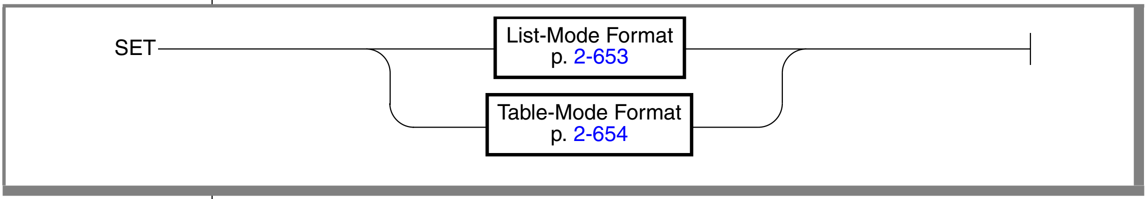
IDS

SET Database Object Mode

Use the SET Database Object Mode statement to change the filtering mode of constraints of unique indexes, or to enable or disable constraints, indexes, and triggers.

To specify whether constraints are checked at the statement level or at the transaction level, see "SET Transaction Mode" on page 2-725.

Syntax



Usage

When you change the mode of constraints, indexes, or triggers, the change remains in effect until you change the mode of the database object again.

The **sysobjstate** system catalog table lists all of the database objects in the database and the current mode of each database object. For information on the **sysobjstate** table, see the *IBM Informix Guide to SQL: Reference*.

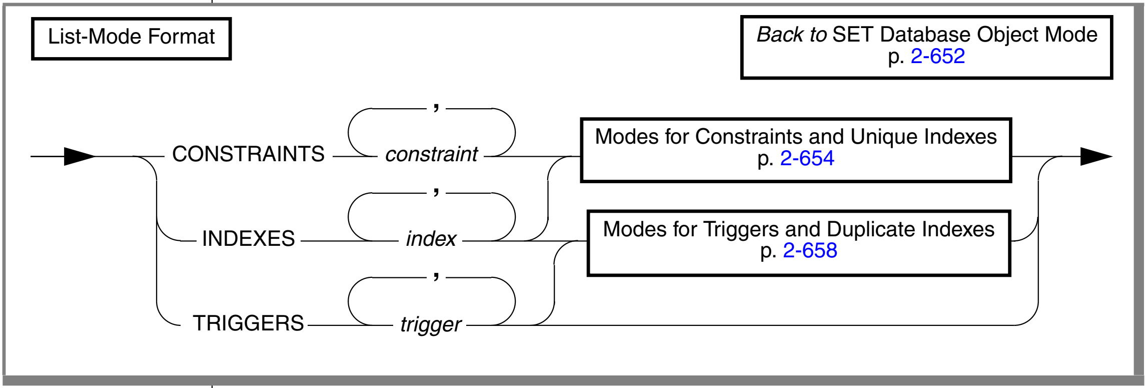
Privileges Required for Changing Database Object Modes

To change the mode of a constraint, index, or trigger, you must have the necessary privileges. You must meet at least one of these requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the table on which the database object is defined and must have the Resource privilege on the database.
- You must have the Alter privilege on the table on which the database object is defined and the Resource privilege on the database.

List-Mode Format

Use list-mode format to change the mode for one or more constraint, index, or trigger.


List-Mode Format
Back to SET Database Object Mode p. 2-652
CONSTRAINTS
constraint
,
INDEXES
index
,
TRIGGERS
trigger
,
Modes for Constraints and Unique Indexes p. 2-654
Modes for Triggers and Duplicate Indexes p. 2-658

Element	Purpose	Restrictions	Syntax
constraint	Name of a constraint whose mode is to be set	Must be a local constraint, and all constraints in the list must be defined on the same table.	Database Object Name, p. 4-44
index	Name of an index whose mode is to be set	Must be a local index, and all indexes in the list must be defined on the same table.	Database Object Name, p. 4-44
trigger	Name of a trigger whose mode is to be set	Must be a local trigger, and all triggers in the list must be defined on the same table or view	Database Object Name, p. 4-44

For example, to change the mode of the unique index **unq_ssn** on the **cust_subset** table to filtering, enter the following statement:

```
SET INDEXES unq_ssn FILTERING
```

You can also use the list-mode format to change the mode for a list of constraints, indexes, or triggers that are defined on the same table. Assume that four triggers are defined on the **cust_subset** table: **insert_trig**, **update_trig**, **delete_trig**, and **execute_trig**. Also assume that all four triggers are enabled. To disable all triggers except **execute_trig**, enter this statement:

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED
```

If **my_trig** is a disabled INSTEAD OF trigger on a view, the following statement enables that trigger:

```
SET TRIGGERS my_trig DENABLED
```

Table-Mode Format

Use table-mode format to change the mode of all database objects of a given type that have been defined on a specified table.

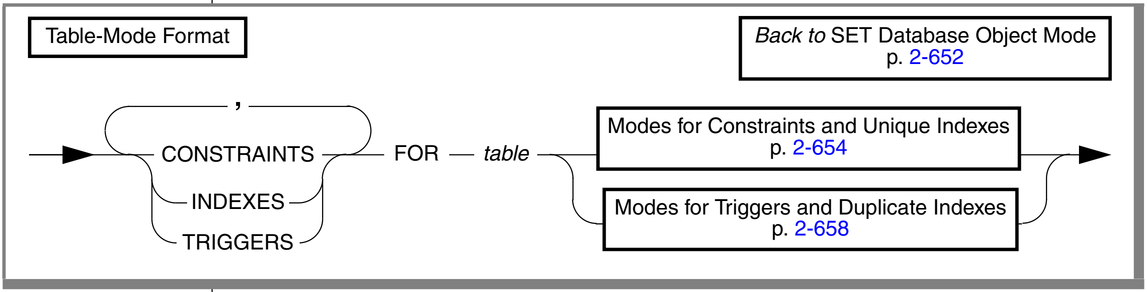


Table-Mode Format
Back to SET Database Object Mode p. 2-652
,
CONSTRAINTS
INDEXES
TRIGGERS
FOR
table
Modes for Constraints and Unique Indexes p. 2-654
Modes for Triggers and Duplicate Indexes p. 2-658

Element	Purpose	Restrictions	Syntax
table	Table on which objects are defined	Must be a local table. Objects defined on a temporary table cannot be set to disabled or filtering modes.	Database Object Name, p. 4-44

This example disables all constraints defined on the **cust_subset** table

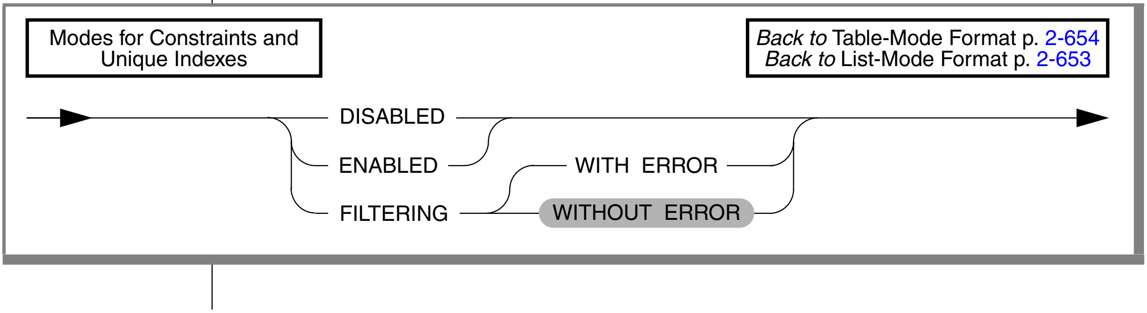
```
SET CONSTRAINTS FOR cust_subset DISABLED
```

In table-mode format, you can change the modes of more than one database object type with a single statement. For example, this enables all constraints, indexes, and triggers that are defined on the **cust_subset** table:

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset ENABLED
```

Modes for Constraints and Unique Indexes

You can specify enabled or disabled mode for a constraint or a unique index.


Modes for Constraints and Unique Indexes
Back to Table-Mode Format p. 2-654
Back to List-Mode Format p. 2-653
DISABLED
ENABLED
FILTERING
WITH ERROR
WITHOUT ERROR

If you specify no mode for a constraint in a CREATE TABLE, ALTER TABLE, or SET Database Object Mode statement, the constraint is enabled by default. If you do not specify the mode for an index in the CREATE INDEX or SET Database Object Mode statement, the index is enabled by default.

Definitions of Database Object Modes

You can use database object modes to control the effects of INSERT, DELETE, and UPDATE statements. Your choice of mode affects the tables whose data you are manipulating, the behavior of the database objects defined on those tables, and the behavior of the data manipulation statements themselves.

Enabled Mode

Constraints, indexes, and triggers are enabled by default. The CREATE TABLE, ALTER TABLE, CREATE INDEX, and CREATE TRIGGER statements create database objects in enabled mode, unless you specify another mode. When a database object is enabled, the database server recognizes the existence of the database object and takes the database object into consideration while it executes an INSERT, DELETE, or UPDATE statement. Thus, an enabled constraint is enforced, an enabled index updated, and an enabled trigger is executed when the trigger event takes place.

When you enable constraints and unique indexes, if a violating row exists, the data manipulation statement fails (that is, no rows are changed) and the database server returns an error message.

Disabled Mode

When a database object is disabled, the database server ignores it during the execution of an INSERT, DELETE, or UPDATE statement. A disabled constraint is not enforced, a disabled index is not updated, and a disabled trigger is not executed when the trigger event takes place. When you disable constraints and unique indexes, any data manipulation statement that violates the restriction of the constraint or unique index succeeds (that is, the target row is changed), and the database server does not return an error message.

You can use the disabled mode to add a new constraint or new unique index to an existing table, even if some rows in the table do not satisfy the new integrity specification. Disabling can also be efficient in LOAD operations.

For information on adding a constraint, see "Adding a Constraint When Existing Rows Violate the Constraint" in the ALTER TABLE statement. For information on adding a unique index, see "Adding a Unique Index When Duplicate Values Exist in the Column" in the CREATE INDEX statement.

Filtering Mode

When a constraint or unique index is in filtering mode, the INSERT, DELETE, OR UPDATE statement succeeds, but the database server enforces the constraint or the unique-index requirement by writing any failed rows to the violations table associated with the target table. Diagnostic information about the constraint violation or unique-index violation is written to the diagnostics table associated with the target table.

In data manipulation operations, filtering mode has the following specific effects on INSERT, UPDATE, and DELETE statements:

- A constraint violation or unique-index violation during an INSERT statement causes the database server to make a copy of the nonconforming record and write it to the violations table. The database server does not write the nonconforming record to the target table.

  If the INSERT statement is not a singleton INSERT, the rest of the insert operation proceeds with the next record.
- A constraint violation or unique-index violation during an UPDATE statement causes the database server to make a copy of the existing record that was to be updated and write it to the violations table. The database server also makes a copy of the new record and writes it to the violations table, but the actual record is not updated in the target table. If the UPDATE statement is not a singleton update, the rest of the update operation proceeds with the next record.
- A constraint violation or unique-index violation during a DELETE statement causes the database server to make a copy of the record that was to be deleted and write it to the violations table. The database server does not delete the actual record in the target table. If the DELETE statement is not a singleton delete, the rest of the delete operation proceeds with the next record.

In all of these cases, the database server sends diagnostic information about each constraint violation or unique-index violation to the diagnostics table associated with the target table.

For information on the structure of the records that the database server writes to the violations and diagnostics tables, see "Structure of the Violations Table" on page 2-734 and "Structure of the Diagnostics Table" on page 2-742.

Starting and Stopping the Violations and Diagnostics Tables

You must use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the target table on which the database objects are defined, either before you set any database objects that are defined on the table to the filtering mode, or after you set database objects to filtering, but before any users issue INSERT, DELETE, or UPDATE statements.

If you want to stop the database server from filtering bad records to the violations table and sending diagnostic information about each bad record to the diagnostics table, you must issue a STOP VIOLATIONS TABLE statement.

For further information on these statements, see "START VIOLATIONS TABLE" on page 2-729 and "STOP VIOLATIONS TABLE" on page 2-748.

Error Options for Filtering Mode

When you set the mode of a constraint or unique index to filtering, you can specify one of two error options. These error options control whether the database server displays an integrity-violation error message when it encounters bad records during execution of data manipulation statements.

- The WITH ERROR option instructs the database server to return a referential integrity-violation error message after executing an INSERT, DELETE, or UPDATE statement in which one or more of the target rows causes a constraint violation or a unique-index violation.
- The WITHOUT ERROR option is the default. This option prevents the database server from issuing a referential integrity-violation error message to the user after an INSERT, DELETE, or UPDATE statement causes a constraint violation or a unique-index violation.

Net Effect of Filtering Mode on the Database

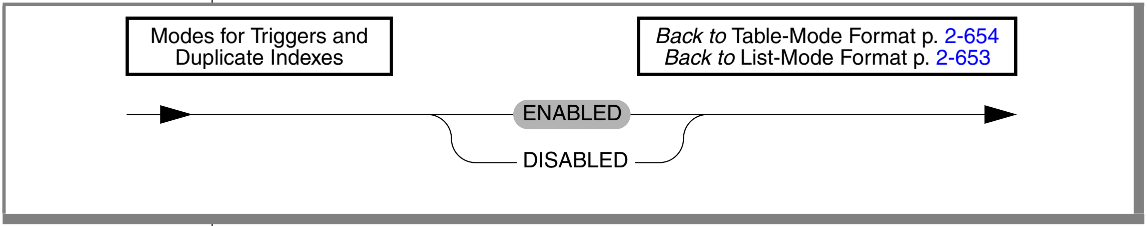
The net effect of the filtering mode is that the contents of the target table always satisfy all constraints on the table and any unique-index requirements on the table.

In addition, the database server does not lose any data values that violate a constraint or unique-index requirement because bad records are sent to the violations table and diagnostic information about those records is sent to the diagnostics table.

Furthermore, when filtering mode is in effect, INSERT, DELETE, and UPDATE operations on the target table do not fail when the database server encounters bad records. These operations succeed in adding all the good records to the target table. Thus, filtering mode is appropriate for large-scale batch updates of tables. The user can fix records that violate constraints and unique-index requirements after the fact. The user does not have to fix the bad records before the batch update or lose the bad records during the batch update.

Modes for Triggers and Duplicate Indexes

You can specify the modes for triggers or duplicate indexes.


Modes for Triggers and Duplicate Indexes
Back to Table-Mode Format p. 2-654
Back to List-Mode Format p. 2-653
ENABLED
DISABLED

If you specify no mode for an index or for a trigger when you create it or in the SET Database Object Mode statement, the object is enabled by default.

Related Information

Related statements: ALTER TABLE, CREATE TABLE, CREATE INDEX, CREATE TRIGGER, START VIOLATIONS TABLE, and STOP VIOLATIONS TABLE

For a discussion of object modes and violation detection and examples that show how database object modes work when users execute data manipulation statements on target tables or add new constraints and indexes to target tables, see the *IBM Informix Guide to SQL: Tutorial*.

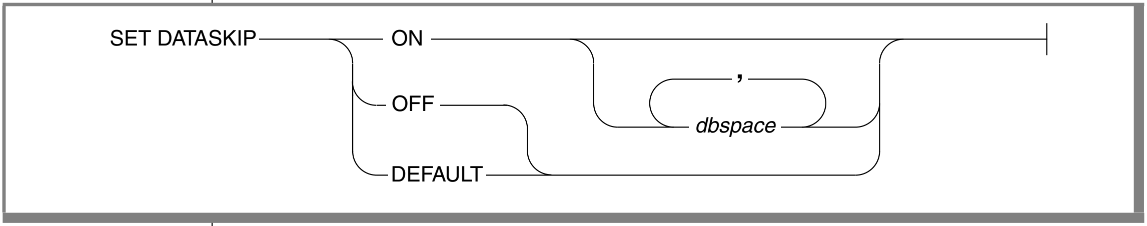
For information on the system catalog tables associated with the SET Database Object Mode statement, see the **sysobjstate** and **sysviolations** tables in the *IBM Informix Guide to SQL: Reference*.

+

SET DATASKIP

Use the SET DATASKIP statement to control whether the database server skips a dbspace that is unavailable during the processing of a transaction.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	Name of the skipped dbspace	Must exist at time of execution	Identifier, p. 4-189

Usage

SET DATASKIP allows you to reset at runtime the Dataskip feature, which controls whether the database server skips a dbspace that is unavailable (for example, due to a media failure) in the course of processing a transaction.

E/C

In ESQL/C, the warning flag **sqlca.sqlwarn.sqlwarn6** is set to `W` if a dbspace is skipped. See also the *IBM Informix ESQL/C Programmer's Manual*. ♦

IDS

This statement applies only to tables that are fragmented across dbspaces. It does not apply to blobspaces nor to sbspaces. ♦

Specifying SET DATASKIP ON without a *dbspace* instructs the database server to skip any dbspaces in the fragmentation list that are unavailable. You can use the **onstat -d** or -D utility to determine whether a dbspace is down.

When you specify SET DATASKIP ON *dbspace*, you are instructing the database server to skip the specified *dbspace* if it is unavailable.

If you specify SET DATASKIP OFF, the Dataskip feature is disabled. If you specify SET DATASKIP DEFAULT, the database server uses the setting for the Dataskip feature from the ONCONFIG file.

Circumstances When a Dbspace Cannot Be Skipped

The database server cannot skip a dbspace under certain conditions. The following list outlines those conditions:

- Referential constraint checking

  When you want to delete a parent row, the child rows must also be available for deletion, and must exist in an available fragment.

  When you want to insert a new child row, the parent row must be found in the available fragments.

- Updates

  When you perform an update that moves a record from one fragment to another, both fragments must be available.

- Inserts

  When you try to insert records in a expression-based fragmentation strategy and the dbspace is unavailable, an error is returned.

  When you try to insert records in a round-robin fragment-based strategy, and a dbspace is down, the database server inserts the rows into any available dbspace.

  When no dbspace is available, an error is returned.

- Indexing

  When you perform updates that affect the index, such as when you insert or delete records, or when you update an indexed field, the index must be available.

  When you try to create an index, the dbspace you want to use must be available.

- Serial keys

  The first fragment is used to store the current serial-key value internally. This is not visible to you except when the first fragment becomes unavailable and a new serial key value is required, which can happen during INSERT statements.

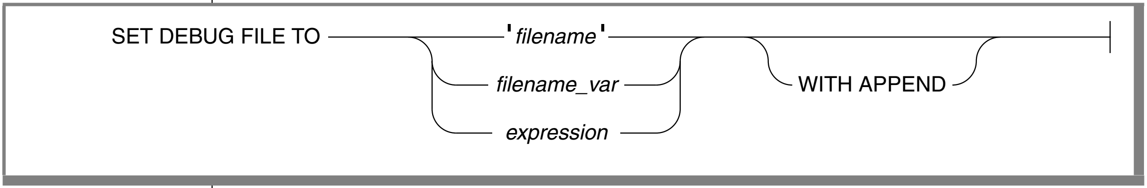
Related Information

For additional information about the Dataskip feature, see your *Administrator's Guide*.

SET DEBUG FILE TO

Use the SET DEBUG FILE TO statement to name the file that is to hold the run-time trace output of an SPL routine.

Syntax



Element	Purpose	Restrictions	Syntax
expression	Expression that returns a filename	Must be a valid filename	Expression, p. 4-67
filename	Pathname of the file that contains the output of the TRACE statement	See "Using the WITH APPEND Option" on page 2-661	Quoted String, p. 4-243.
filename_var	Host variable storing *filename* string	Must be a character data type	Language specific

Usage

This statement indicates that the output of the TRACE statement in the SPL routine goes to the file that *filename* indicates. Each time the TRACE statement is executed, the trace data is added to this output file.

Using the WITH APPEND Option

The output file that you specify in the SET DEBUG FILE TO statement can be a new file or existing file. If you specify an existing file, its current contents are purged when you issue the SET DEBUG FILE TO statement. The first execution of a TRACE command sends trace output to the beginning of the file.

If you include the WITH APPEND option, the current contents of the file are preserved when you issue the SET DEBUG FILE TO statement. The first execution of a TRACE command adds trace output to the end of the file.

If you specify a new file in the SET DEBUG FILE TO statement, it makes no difference whether you include the WITH APPEND option. The first execution of a TRACE command sends trace output to the beginning of the new file whether you include or omit the WITH APPEND option.

Closing the Output File

To close the file that the SET DEBUG FILE TO statement opened, issue another SET DEBUG FILE TO statement with another filename. You can then edit the contents of the first file.

Redirecting Trace Output

You can use the SET DEBUG FILE TO statement outside an SPL routine to direct the trace output of the SPL routine to a file. You can also use this statement inside an SPL routine to redirect its own output.

Location of the Output File

If you invoke a SET DEBUG FILE TO statement with a simple filename on a local database, the output file is located in your current directory. If your current database is on a remote database server, the output file is located in your home directory on the remote database server. If you provide a full pathname for the debug file, the file is placed in the directory and file that you specify on the remote database server. If you do not have write permissions in the directory, you get an error.

The following example sends the output of the SET DEBUG FILE TO statement to a file called **debug.out**:

```
SET DEBUG FILE TO 'debug' || '.out'
```

Related Information

Related statement: TRACE

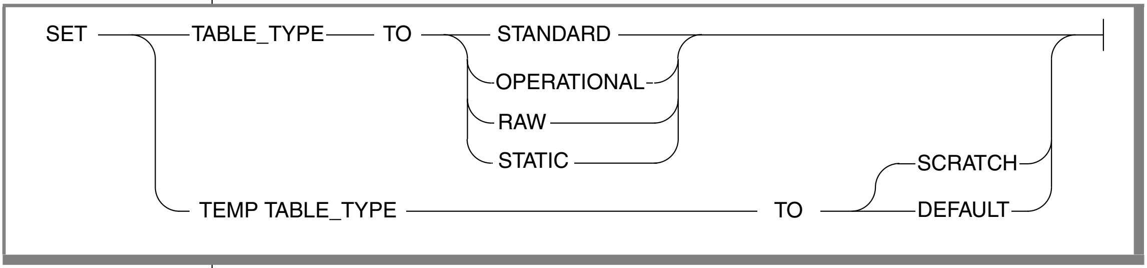
For a task-oriented discussion of SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

XPS

SET Default Table Type

Use the SET Default Table Type statement to specify the default table type for tables (or temporary tables) that you subsequently create in the same session.

Syntax



Usage

If CREATE TABLE specifies no table type, the default table type is STANDARD. The SET TABLE_TYPE statement can change this default for subsequent CREATE TABLE statements in the current session. Similarly, you can use the SET TEMP TABLE_TYPE to change the default temporary table type.

These statements have no effect on tables for which you explicitly specify a table type in the statement that creates the new table or temporary table.

Because the CREATE Temporary TABLE statement requires an explicit table type, the SET TEMP TABLE_TYPE statement only affects SQL operations that create a temporary implicitly, such as in executing join operations, SELECT statements with the GROUP BY or ORDER BY clause, and index builds.

The effect of SET Default Table Type persists until the end of the session, or until you issue another SET Default Table Type statement to specify a new default table type.

The SET TABLE_TYPE TO STANDARD statement and the SET TEMP TABLE_TYPE TO DEFAULT statements restore the default behavior.

Although the scope of these statements is the current session, they can be used to have a database-wide effect. The next example shows how to do this by using SPL routines to establish a default table type at connect time:

```
CREATE PROCEDURE public.sysdbopen()
   SET TABLE_TYPE TO RAW;
   SET TEMP TABLE_TYPE TO SCRATCH;
   SET TABLE_SPACE TO other_tables;
...
END PROCEDURE;

CREATE PROCEDURE helene.sysdbopen()
   EXECUTE PROCEDURE public.sysdbopen();
   SET ROLE marketing;
   SET TABLE_SPACE TO marketing_dbslice;
END PROCEDURE;
```

Related Information

Related statements: CREATE TABLE, CREATE TEMP TABLE, SET Default Table Space.

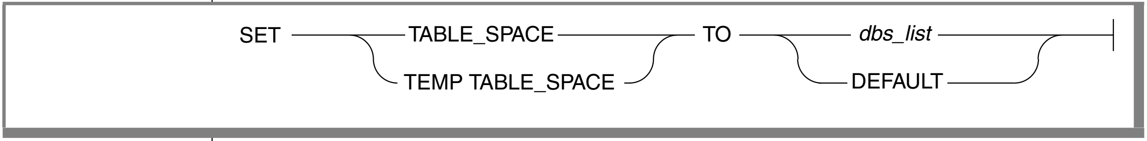
For more information on table types that can be specified in the CREATE TABLE statement, see "CREATE TABLE" on page 2-214. For more information about temporary tables see "CREATE Temporary TABLE" on page 2-261.

XPS

SET Default Table Space

Use the SET Default Table Space statement to specify the default storage space used by subsequent CREATE TABLE statements (and SELECT statements that include the INTO Table clause) in the same session. This statement can also specify a default storage location for temporary tables.

Syntax



Element	Purpose	Restrictions	Syntax
dbs_list	A dbspace, dbslice, or a comma-separated list of dbspaces	Must exist	Identifier, p. 4-189

Usage

When the CREATE TABLE or CREATE Temporary TABLE statement includes no fragmentation clause, the database server uses the dbspace of the current database as the default storage location. You can use the SET TABLE_SPACE statements to change the default to another dbslice or a list of one or more dbspaces. This statement also sets the default dbspace for SELECT statements that include the INTO Table clause. These defaults persist for the rest of the current session, or until the next SET Default Table Space statement.

Likewise, you can use the SET TEMP TABLE_SPACE statement to change the default storage location for CREATE Temporary TABLE statements that do not include the Storage Options clause.

Specifying the TO DEFAULT option restores the default behavior.

Related Information

Related statements: CREATE TABLE, CREATE Temporary TABLE, SET Default Table Type.

+

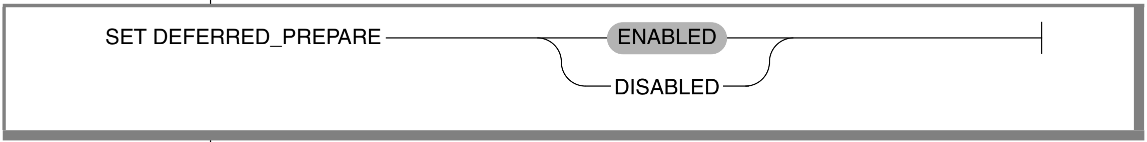
IDS

E/C

SET DEFERRED_PREPARE

Use the SET DEFERRED_PREPARE statement to control whether a client process postpones sending a PREPARE statement to the database server until the OPEN or EXECUTE statement is sent. Use this statement with ESQL/C.

Syntax



Usage

By default, the SET DEFERRED_PREPARE statement causes the application program to delay sending the PREPARE statement to the database server until the OPEN or EXECUTE statement is executed. In effect, the PREPARE statement is bundled with the other statement so that one round trip of messages instead of two is sent between the client and the server. This Deferred-Prepare feature works with the following sequences:

- PREPARE, DECLARE, OPEN statement blocks that operate with the FETCH or PUT statements
- PREPARE followed by the EXECUTE or EXECUTE IMMEDIATE statement

You can specify ENABLED or DISABLED options for SET DEFERRED_PREPARE.

If you specify no option, the default is ENABLED. This example enables the Deferred-Prepare feature by default:

```
EXEC SQL set deferred_prepare;
```

The ENABLED option enables the Deferred-Prepare feature within the application. The following example explicitly specifies the ENABLED option:

```
EXEC SQL set deferred_prepare enabled;
```

After an application issues SET DEFERRED_PREPARE ENABLED, the Deferred-Prepare feature is enabled for subsequent PREPARE statements in the application. The application then exhibits the following behavior:

- The sequence PREPARE, DECLARE, OPEN sends the PREPARE statement to the database server with the OPEN statement. If the prepared statement has syntax errors, the database server does not return error messages to the application until the application declares a cursor for the prepared statement and opens the cursor.
- The sequence PREPARE, EXECUTE sends the PREPARE statement to the database server with the EXECUTE statement. If a prepared statement contains syntax errors, the database server does not return error messages to the application until the application attempts to execute the prepared statement.

If Deferred-Prepare is enabled in a PREPARE, DECLARE, OPEN statement block that contains a DESCRIBE statement, the DESCRIBE statement must follow the OPEN statement rather than the PREPARE statement. If the DESCRIBE follows PREPARE, the DESCRIBE statement results in an error.

Use the DISABLED option to disable the Deferred-Prepare feature within the application. The following example specifies the DISABLED option:

```
EXEC SQL set deferred_prepare disabled;
```

If you specify the DISABLED option, the application sends each PREPARE statement to the database server when the PREPARE statement is executed.

Example of SET DEFERRED_PREPARE

The following code fragment shows a SET DEFERRED_PREPARE statement with a PREPARE, EXECUTE statement block. In this case, the database server executes the PREPARE and EXECUTE statements all at once.

```
EXEC SQL BEGIN DECLARE SECTION;
   int a;
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate descriptor 'desc';
EXEC SQL create database test;
EXEC SQL create table x (a int);

/* Enable Deferred-Prepare feature */
EXEC SQL set deferred_prepare enabled;
/* Prepare an INSERT statement */
EXEC SQL prepare ins_stmt from 'insert into x values(?)';
a = 2;
EXEC SQL EXECUTE ins_stmt using :a;
if (SQLCODE)
   printf("EXECUTE : SQLCODE is %d\n", SQLCODE);
```

Using Deferred-Prepare with OPTOFC

You can use the Deferred-Prepare and Open-Fetch-Close Optimization (OPTOFC) features in combination. The OPTOFC feature delays sending the OPEN message to the database server until the FETCH message is sent. The following situations occur if you enable the Deferred-Prepare and OPTOFC features at the same time:

- If the text of a prepared statement contains syntax errors, the error messages are not returned to the application until the first FETCH statement is executed.
- A DESCRIBE statement cannot be executed until after the FETCH statement.
- You must issue an ALLOCATE DESCRIPTOR statement before a DESCRIBE or GET DESCRIPTOR statement can be executed.

The database server performs an internal execution of a SET DESCRIPTOR statement which sets the TYPE, LENGTH, DATA, and other fields in the system descriptor area. You can specify a GET DESCRIPTOR statement after the FETCH statement to see the data that is returned.

Related Information

Related statements: DECLARE, DESCRIBE, EXECUTE, OPEN, and PREPARE

For a task-oriented discussion of the PREPARE statement and dynamic SQL, see the *IBM Informix Guide to SQL: Tutorial*.

For more information about concepts that relate to the SET DEFERRED_PREPARE statement, see the *IBM Informix ESQL/C Programmer's Manual*.

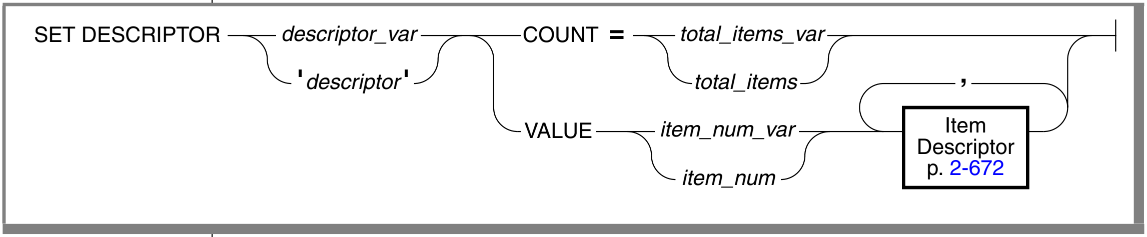
+

E/C

SET DESCRIPTOR

The SET DESCRIPTOR statement assigns values to a system-descriptor area. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	String that identifies system-descriptor area to which values are assigned	System-descriptor area (SDA) must be previously allocated.	Quoted String, p. 4-243
descriptor_var	Host variable with value of *descriptor*	Same restrictions as *descriptor.*	Language specific
item_num	Unsigned integer that specifies ordinal position of an item descriptor in the system-descriptor area (SDA)	0 < *item_num* ≤ (number of item descriptors specified when SDA was allocated).	Literal Number, p. 4-216
item_num_var	Host variable that stores *item_num*	Same restrictions as *item_num*.	Language specific
total_items	Unsigned integer that specifies how many items the SDA describes	Same restrictions as *item_num*.	Literal Number, p. 4-216
total_items_var	Host variable that stores *total_items*	Same restrictions as *total_items*.	Language specific

Usage

The SET DESCRIPTOR statement can be used after you have described SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE), and INSERT statements with the DESCRIBE...USING SQL DESCRIPTOR statement.

SET DESCRIPTOR can assign values to a system-descriptor area in these cases:

- To set the **COUNT** field of a system-descriptor area to match the number of items for which you are providing descriptions in the system-descriptor area
- To set the item descriptor for each value for which you are providing descriptions in the system-descriptor area
- To modify the contents of an item-descriptor field

If an error occurs during the assignment to any identified system-descriptor fields, the contents of all identified fields are set to `0` or NULL, depending on the data type of the variable.

Using the COUNT Clause

Use the COUNT clause to set the number of items that are to be used in the system-descriptor area. If you allocate a system-descriptor area with more items than you are using, you need to set the **COUNT** field to the number of items that you are actually using. The following example shows a fragment of an ESQL/C program:

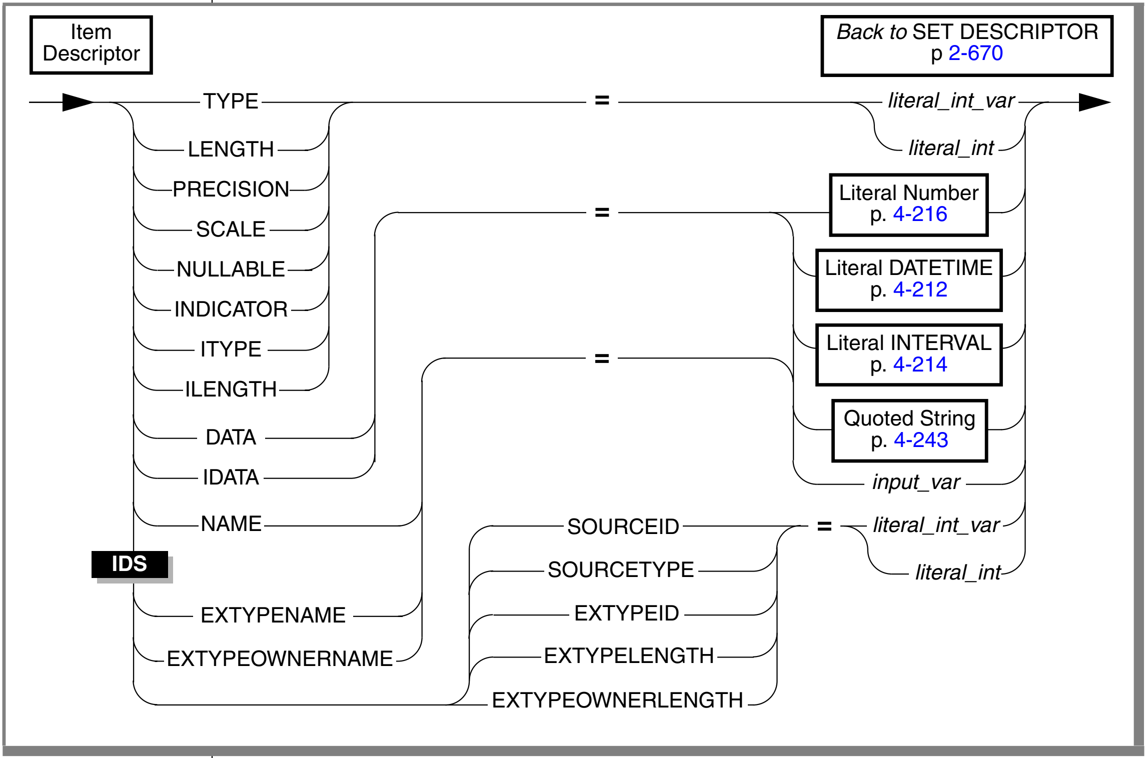
```
EXEC SQL BEGIN DECLARE SECTION;
   int count;

EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc_100'; /*allocates for 100 items*/
   count = 2;
EXEC SQL set descriptor 'desc_100' count = :count;
```

Using the VALUE Clause

Use the VALUE clause to assign values from host variables into fields of a system-descriptor area. You can assign values for items for which you are providing a description (such as parameters in a WHERE clause), or you can modify values for items after you use a DESCRIBE statement to fill the fields for a SELECT or INSERT statement.

Item Descriptor

Use the Item Descriptor portion of the SET DESCRIPTOR statement to set value for an individual field in a system-descriptor area.



Element	Purpose	Restrictions	Syntax
input_var	Host variable storing data for the specified item descriptor field	Must be appropriate for the specified field	Language-specific rules for names
literal_int	Integer value (> 0) assigned to the specified item descriptor field	Restrictions depend on the keyword to the left of **=** symbol	Literal Number, p. 4-216
literal_int_var	Variable having *literal_int* value	Same as for *literal_int*	Same as for *input_var*

For information on codes that are valid for the TYPE or ITYPE fields and their meanings, see "Setting the TYPE or ITYPE Field" on page 2-673.

For the restrictions that apply to other field types, see the individual headings for field types under "Using the VALUE Clause" on page 2-671.

Setting the TYPE or ITYPE Field

Use these integer values to set the value of **TYPE** or **ITYPE** for each item.

SQL Data Type	Integer Value	X-Open Integer Value	SQL Data Type	Integer Value	X-Open Integer Value
CHAR	0	1	MONEY	8	–
SMALLINT	1	4	DATETIME	10	–
INTEGER	2	5	BYTE	11	–
FLOAT	3	6	TEXT	12	–
SMALLFLOAT	4	–	VARCHAR	13	–
DECIMAL	5	3	INTERVAL	14	–
SERIAL	6	–	NCHAR	15	–
DATE	7	–	NVARCHAR	16	–

IDS

The following table lists integer values that represent additional data types available with Dynamic Server.

SQL Data Type	Integer Value	SQL Data Type	Integer Value
INT8	17	COLLECTION	23
SERIAL8	18	Varying-length OPAQUE type	40
SET	19	Fixed-length OPAQUE type	41
MULTISET	20	LVARCHAR (client-side only)	43
LIST	21	BOOLEAN	45
ROW	22		

♦

The same **TYPE** constants can also appear in the **syscolumns.coltype** column in the system catalog; see *IBM Informix Guide to SQL: Reference*.

For code that is easier to maintain, use the predefined constants for these SQL data types instead of their actual integer values. These constants are defined in the **$INFORMIX/incl/public/sqlstypes.h** header file. You cannot, however, use the actual constant name in the SET DESCRIPTOR statement. Instead, assign the constant to an integer host variable and specify the host variable in the SET DESCRIPTOR statement file.

The following example shows how you can set the **TYPE** field in ESQL/C:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
   int itemno, type;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL allocate descriptor 'desc1' with max 5;
...
type = SQLINT; itemno = 3;
EXEC SQL set descriptor 'desc1' value :itemno type = :type;
}
```

This information is identical for **ITYPE**. Use **ITYPE** when you create a dynamic program that does not comply with the X/Open standard.

Compiling Without the -xopen Option

If you compile without the **-xopen** option, the normal Informix SQL code is assigned for TYPE. You must be careful not to mix normal and X/Open modes because errors can result. For example, if a data type is not defined under X/Open mode but is defined under normal mode, executing a SET DESCRIPTOR statement can result in an error.

X/O

Setting the TYPE Field in X/Open Programs

In X/Open mode, you must use the X/Open set of integer codes for the data type in the **TYPE** field.

If you use the **ILENGTH**, **IDATA**, or **ITYPE** fields in a SET DESCRIPTOR statement, a warning message appears. The warning indicates that these fields are not standard X/Open fields for a system-descriptor area.

For code that is easier to maintain, use the predefined constants for these X/Open SQL data types instead of their actual integer value. These constants are defined in the **$INFORMIX/incl/public/sqlxtype.h** header file.

Using DECIMAL or MONEY Data Types

If you set the **TYPE** field for a DECIMAL or MONEY data type, and you want to use a scale or precision other than the default values, set the **SCALE** and **PRECISION** fields. You do not need to set the **LENGTH** field for a DECIMAL or MONEY item; the **LENGTH** field is set accordingly from the **SCALE** and **PRECISION** fields.

Using DATETIME or INTERVAL Data Types

If you set the **TYPE** field for a DATETIME or INTERVAL value, the **DATA** field can be a DATETIME or INTERVAL literal or a character string. If you use a character string, the **LENGTH** field must be the encoded qualifier value.

To determine the encoded qualifiers for a DATETIME or INTERVAL character string, use the datetime and interval macros in the **datetime.h** header file.

If you set **DATA** to a host variable of DATETIME or INTERVAL, you do not need to set **LENGTH** explicitly to the encoded qualifier integer.

Setting the DATA or IDATA Field

When you set the **DATA** or **IDATA** field, use the appropriate type of data (character string for CHAR or VARCHAR, integer for INTEGER, and so on).

If any value other than **DATA** is set, the value of **DATA** is undefined. You cannot set the **DATA** or **IDATA** field for an item without setting **TYPE** for that item. If you set the **TYPE** field for an item to a character type, you must also set the **LENGTH** field. If you do not set the **LENGTH** field for a character item, you receive an error.

Setting the LENGTH or ILENGTH Field

If your **DATA** or **IDATA** field contains a character string, you must specify a value for **LENGTH**. If you specify `LENGTH=0`, **LENGTH** is automatically set to the maximum length of the string. The **DATA** or **IDATA** field can contain a 368-literal character string or a character string derived from a character variable of CHAR or VARCHAR data type. This provides a method to determine dynamically the length of a string in the **DATA** or **IDATA** field.

If a DESCRIBE statement precedes a SET DESCRIPTOR statement, **LENGTH** is automatically set to the maximum length of the character field that is specified in your table.

This information is identical for **ILENGTH**. Use **ILENGTH** when you create a dynamic program that does not comply with the X/Open standard.

Setting the INDICATOR Field

If you want to put a null value into the system-descriptor area, set the **INDICATOR** field to `-1` and do not set the **DATA** field.

If you set the **INDICATOR** field to `0` to indicate that the data is not null, you must set the **DATA** field.

IDS

Setting Opaque-Type Fields

The following item-descriptor fields provide information about a column that has an opaque type as its data type:

- The **EXTYPEID** field stores the extended identifier for the opaque type. This integer value must correspond to a value in the **extended_id** column of the **sysxtdtypes** system catalog table.
- The **EXTYPENAME** field stores the name of the opaque type. This character value must correspond to a value in the **name** column of the row with the matching **extended_id** value in the **sysxtdtypes** system catalog table.
- The **EXTYPELENGTH** field stores the length of the opaque-type name. This integer value is the length, in bytes, of the string in the **EXTYPENAME** field.
- The **EXTYPEOWNERNAME** field stores the name of the opaque-type owner. This character value must correspond to a value in the **owner** column of the row with the matching **extended_id** value in the **sysxtdtypes** system catalog table.
- The **EXTYPEOWNERLENGTH** field stores the length of the value in the **EXTTYPEOWNERNAME** field. This integer value is the length, in bytes, of the string in the **EXTYPEOWNERNAME** field.

For more information on the **sysxtdtypes** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

Setting Distinct-Type Fields

The following item-descriptor fields provide information about a column that has a distinct type as its data type:

- The **SOURCEID** field stores the extended identifier for the source data type.

  Set this field if the source type of the distinct type is an opaque data type. This integer value must correspond to a value in the **source** column for the row of the **sysxtdtypes** system catalog table whose **extended_id** value matches that of the distinct type you are setting.

- The **SOURCETYPE** field stores the data type constant for the source data type.

  This value is the data type constant for the built-in data type that is the source type for the distinct type. The codes for the **SOURCETYPE** field are the same as those for the **TYPE** field (page 2-673). This integer value must correspond to the value in the **type** column for the row of the **sysxtdtypes** system catalog table whose **extended_id** value matches that of the distinct type you are setting.

For more information on the **sysxtdtypes** system catalog table, see the *IBM Informix Guide to SQL: Reference*.

Modifying Values Set by the DESCRIBE Statement

You can use a DESCRIBE statement to modify the contents of a system-descriptor area after it is set.

After you use DESCRIBE on a SELECT or an INSERT statement, you must check to determine whether the **TYPE** field is set to either `11` or `12` to indicate a TEXT or BYTE data type. If **TYPE** contains an `11` or a `12`, you must use the SET DESCRIPTOR statement to reset **TYPE** to `116`, which indicates FILE type.

Related Information

Related statements: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, and PUT

For more information on system-descriptor areas, refer to the *IBM Informix ESQL/C Programmer's Manual*.

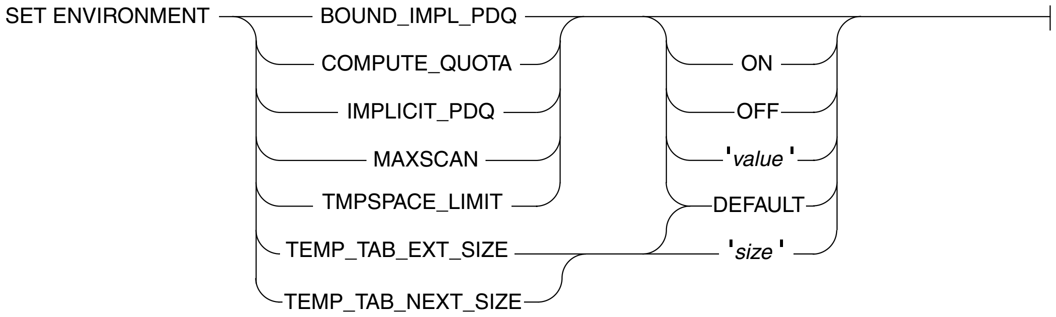
XPS

+

SET ENVIRONMENT

The SET ENVIRONMENT statement lets you specify options that affect subsequent queries submitted within the same routine.

Syntax



Element	Description	Restrictions	Syntax
value	Value to set for the specified environment option, which this statement also enables	Must be valid for the specified environment option	Quoted String, p. 4-243
size	The extent size for system-generated temporary tables, in kilobytes	Must be valid for the specified environment option	Quoted String, p. 4-243

Usage

The SET ENVIRONMENT statement specifies environment options that manage resource use by the routine in which the statement is executed. For example, the SET ENVIRONMENT IMPLICIT_PDQ ON statement enables automatic **PDQPRIORITY** allocation for queries submitted in the routine.

The OFF keyword disables the specified option.

The ON keyword enables the specified option.

The DEFAULT keyword sets the specified option to its default value.

The arguments that follow the option name depend on the syntax of the option. The option name and its ON, OFF, and DEFAULT keywords are not quoted and are not case sensitive. All other arguments must be enclosed in single or double quotation marks. If a quoted string is a valid argument for the environment option, it is case sensitive.

If you enter an undefined option name or an illegal *value* for a defined option, no error is returned. Undefined options are ignored, but they might produce unexpected results, if you intended some effect that a misspelled option name cannot produce. The SET ENVIRONMENT statement can enable only the environment options that are described in sections that follow.

For information about the performance implications of the SET ENVIRONMENT options, refer to the *Performance Guide.*

BOUND_IMPL_PDQ Environment Option

If IMPLICIT_PDQ is set to ON or to a value, use the BOUND_IMPL_PDQ environment option to specify that the allocated memory should be bounded by the current explicit **PDQPRIORITY** value or range. If IMPLICIT_PDQ is OFF, then BOUND_IMPL_PDQ is ignored. For example, you might execute the following statement to force the database server to use explicit **PDQPRIORITY** values as guidelines in allocating memory if the IMPLICIT_PDQ environment option has already been set:

```
SET ENVIRONMENT BOUND_IMPL_PDQ ON
```

If you set both IMPLICIT_PDQ and BOUND_IMPL_PDQ, then the explicit **PDQPRIORITY** value determines the upper limit of memory that can be allocated to a query. If **PDQPRIORITY** is specified as a range, the database server grants memory within the range specified.

For detailed information, see the *Performance Guide*.

COMPUTE_QUOTA Environment Option

Use the COMPUTE_QUOTA environment option to allow the optimizer to use only one CPU VP (virtual processor) on each coserver for each query operator instead of using all CPU VPs of each coserver.

To turn on this environment option, execute the following statement:

```
SET ENVIRONMENT COMPUTE_QUOTA ON
```

IMPLICIT_PDQ Environment Option

Use the IMPLICIT_PDQ environment option to allow the database server to determine the amount of memory allocated to a query. Unless BOUND_IMPL_PDQ is also set, the database server ignores the current explicit setting of **PDQPRIORITY**. It does not allocate more memory, however, than is available when **PDQPRIORITY** is set to 100, as determined by MAX_PDQPRIORITY / 100 * DS_TOTAL_MEMORY.

This environment option is OFF by default.

If *value* is set between `1` and `100`, the database server scales its estimate by the specified value. If you specify a low value, the amount of memory assigned to the query is reduced, which might increase the amount of query-operator overflow.

For example, to request the database server to determine memory allocations for queries and distribute memory among query operators according to their needs, enter the following statement:

```
SET ENVIRONMENT IMPLICIT_PDQ ON
```

To require the database server to use explicit **PDQPRIORITY** settings as the upper bound and optional lower bound of memory that it grants to a query, set the BOUND_IMPL_PDQ environment option.

MAXSCAN Environment Option

Use the MAXSCAN environment option to change the default number of scan threads on each coserver. The default is three scan threads for each CPU VP. You might want to reduce the number of scan threads on a coserver if the GROUP, JOIN, and other operators above the scan are not producing rows quickly enough to keep the default number of scan threads busy.

For some queries, you might want to increase the number of scan threads on each coserver. For example, if each coserver has three CPU VPs, the database server can create nine scan threads on each coserver. To increase the number to four threads for each CPU VP, execute the following statement:

```
SET ENVIRONMENT MAXSCAN  12
```

MAXSCAN is automatically set to `1` if the COMPUTE_QUOTA environment option is enabled or if the isolation level is set to Cursor Stability and the database server can use a pipe operator.

TMPSPACE_LIMIT Environment Option

Use the TMPSPACE_LIMIT environment option to specify the amount of temporary space on each coserver that a query can use for query operator overflow. Temporary space limits do not affect the creation of temporary tables. The limits apply only to the query-operator overflow that occurs when a query cannot get enough memory to complete execution.

By default TMPSPACE_LIMIT is OFF, and a query can use all available temporary space for operator overflow. If the DS_TOTAL_TMPSPACE configuration parameter is not set, then setting TMPSPACE_LIMIT has no effect.

If you enter a value between `0` and `100` as an argument to the TMPSPACE_LIMIT option, the database server sets the temporary space quota to the specified percent of available temporary space that the **ONCONFIG** parameter, DS_TOTAL_TMPSPACE, specifies.

- If the value is `100`, queries can use only the amount of temporary space on each coserver that DS_TOTAL_TMPSPACE. specifies.
- If the value is `0`, query operators cannot overflow to temporary space.
- If you do not specify a value, a query can use all available temporary space on each coserver, as limited by DS_TOTAL_TMPSPACE, for query operator overflow.

To require queries to use only the amount of temporary space specified by the setting of DS_TOTAL_TMPSPACE on each coserver, execute this statement:

```
SET ENVIRONMENT TMPSPACE_LIMIT ON
```

To limit queries to 50 percent of DS_TOTAL_TMPSPACE on each coserver, execute the following statement:

```
SET ENVIRONMENT TMPSPACE_LIMIT "50";
```

TEMP_TAB_EXT_SIZE and TEMP_TAB_NEXT_SIZE Options

Use the TEMP_TAB_EXT_SIZE and TEMP_TAB_NEXT_SIZE environment options to specify the number of kilobytes used as the first and next extent size for a system-generated temporary table.

The following example shows the syntax for setting the first and next extent size using these environment options:

```
SET ENVIRONMENT TEMP_TAB_EXT_SIZE '64'
SET ENVIRONMENT TEMP_TAB_NEXT_SIZE '128'
```

This example sets the first extent size of a generated temporary table to 64 and the next extent size to 128 kilobytes.

The minimum value of these environment options is four times the page size on your system. If you specify a size below the minimum, the server will default the page size to four pages. For flex inserts, the server will default to 32 pages or 128 kilobytes.

The maximum value for TEMP_TAB_EXT_SIZE and TEMP_TAB_NEXT_SIZE is the maximum value of a chunk size.

Use the DEFAULT keyword to reset the values of these environments options back to the system defaults.

Important: *The extent sizes for temporary tables created for hash tables and sorts are calculated by the system and they will not be overridden by these SET ENVIRONMENT variables.*

Local Scope of SET ENVIRONMENT

The scope of these changes to environment options is local to the routine that executes SET ENVIRONMENT, rather than the entire session. Exceptions to this local scope are the **sysdbopen( )** and **sysdbclose( )** SPL routines, which can set the initial environment for a session. For more information on these built-in routines, see "Using sysbdopen( ) and sysdbclose( ) Stored Procedures" on page 2-189.

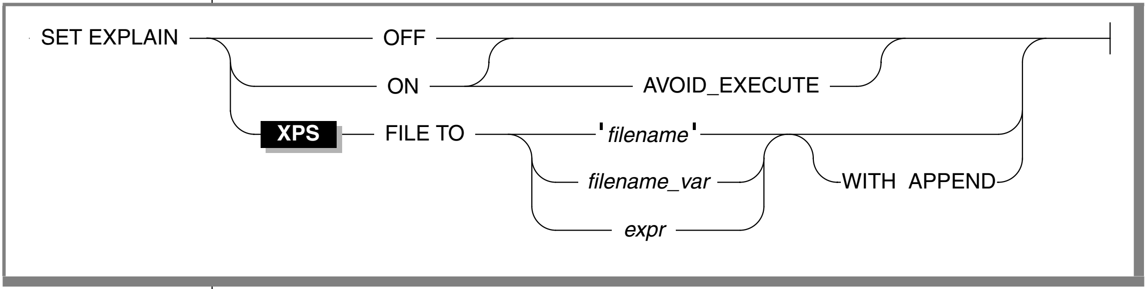
Related Information

Related statements: SET PDQPRIORITY

SET EXPLAIN

Use the SET EXPLAIN statement to display the query plan of optimizer, an estimate of the number of rows returned, and the relative cost of the query.

Syntax



Element	Purpose	Restrictions	Syntax
expr	Expression that returns a *filename* specification	Must return a string satisfying the restrictions on *filename*.	Expression, p. 4-67
filename	Path and filename of the file to receive the output. For the default, see "Location of the Output File" on page 2-662.	Must conform to operating-system rules. If an existing file, see "Using the WITH APPEND Option" on page 2-661.	Quoted String, p. 4-243
filename_var	Host variable that stores *filename*	Must be a character data type.	Language specific

Usage

Output from a SET EXPLAIN ON statement is directed to the appropriate file until you issue a SET EXPLAIN OFF statement or until the program ends. If you do not enter a SET EXPLAIN statement, the default behavior is OFF. The database server does not generate measurements for queries.

The SET EXPLAIN statement executes during the database server optimization phase, which occurs when you initiate a query. For queries that are associated with a cursor, if the query is prepared and does not have host variables, optimization occurs when you prepare it. Otherwise, optimization occurs when you open the cursor.

The SET EXPLAIN statement provides various measurements of the work involved in performing a query.

Option	Purpose
ON	Generates measurements for each subsequent query and writes the results to an output file in the current directory. If the file already exists, new explain output is appended to the existing file.
AVOID_EXECUTE	Prevents a SELECT, INSERT, UPDATE, or DELETE statement from executing while the database server prints the query plan to an output file
OFF	Terminates activity of the SET EXPLAIN statement, so that measurements for subsequent queries are no longer generated or written to the output file
FILE TO	Generates measurements for each subsequent query and allows you to specify the location for the explain output file. If the file already exists, new explain output overwrites the contents of the file unless you use the WITH APPEND option.

Using the AVOID_EXECUTE Option

The SET EXPLAIN ON AVOID_EXECUTE statement activates the Avoid Execute option for a session, or until the next SET EXPLAIN OFF (or ON) without AVOID_EXECUTE. The AVOID_EXECUTE keyword prevents DML statements from executing; the database server prints the query plan to an output file. If you activate AVOID_EXECUTE for a query that contains a remote table, the query does not execute at either the local or remote site.

Use the SET EXPLAIN ON or the SET EXPLAIN OFF statement to turn off the AVOID_EXECUTE option. The SET EXPLAIN ON statement turns off the AVOID_EXECUTE option but continues to generate a query plan and writes the results to an output file.

If you use the SET EXPLAIN ON AVOID_EXECUTE statement inside an SPL routine, the SPL routine and any DDL statements still execute, but the DML statements inside the SPL routine do not execute. The database server prints the query plan of the SPL routine to an output file. To turn off this option, you must execute the SET EXPLAIN ON or the SET EXPLAIN OFF statement outside the SPL routine. If you execute the SET EXPLAIN ON AVOID_EXECUTE statement before you execute an SPL routine, the DML statements inside the SPL routine do not execute, and the database server does not print a query plan of the SPL routine to an output file.

E/C

If you execute the SET EXPLAIN ON AVOID_EXECUTE statement before you open a cursor in an ESQL/C program, each FETCH operation returns the message that the row was not found. If you execute SET EXPLAIN ON AVOID_EXECUTE after an ESQL/C program opens a cursor, however, this statement has no effect on the cursor, which continues to return rows. ♦

Nonvariant functions in a query are still evaluated when AVOID_EXECUTE is in effect, because the database server calculates these functions before optimization.

For example, the **func( )** function is evaluated, even though the following SELECT statement is not executed:

```
SELECT * FROM orders WHERE func(10) > 5
```

For other performance implications of the AVOID_EXECUTE option, see your *Performance Guide*.

Using the FILE TO Option

When you execute a SET EXPLAIN FILE TO statement, explain output is implicitly turned on. The default filename for the output is **sqexplain.out** until changed by a SET EXPLAIN FILE TO statement. Once changed, the filename remains set until the end of the session or until changed by another SET EXPLAIN FILE TO statement.

The filename can be any valid combination of optional path and filename. If no path component is specified, the file is placed in your current directory. The permissions for the file are owned by the current user.

Using the WITH APPEND Option

The output file that you specify in the SETEXPLAIN statement can be a new file or an existing file.

If you specify an existing file, the current contents of the file are purged when you issue the SET EXPLAIN FILE TO statement. The first execution of a FILE TO command sends output to the beginning of the file.

If you include the WITH APPEND option, the current contents of the file are preserved when you issue the SET EXPLAIN FILE TO statement. The execution of a WITH APPEND command appends output to the end of the file.

If you specify a new file in the SET EXPLAIN FILE TO statement, it makes no difference whether you include the WITH APPEND option. The first execution of the command sends output to the beginning of the new file.

Default Name and Location of the Output File

UNIX

On UNIX, when you issue a SET EXPLAIN ON statement, the plan that the optimizer chooses for each subsequent query is written to the sqexplain.out file by default.

If the output file does not exist when you issue the SET EXPLAIN ON statement, the database server creates the output file. If the output file already exists when you issue the SET EXPLAIN ON statement, subsequent output is appended to the file.

If the client application and the database server are on the same computer, the **sqexplain.out** file is stored in your current directory. If you are using a Version 5.x or earlier client application and the **sqexplain.out** file does not appear in the current directory, check your home directory for the file. When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. ♦

Windows

On Windows, SET EXPLAIN ON writes the plan that the optimizer chooses for each subsequent query to file **%INFORMIXDIR%\sqexpln\*username*.out** where *username* is the user login. ♦

SET EXPLAIN Output

By examining the SET EXPLAIN output file, you can determine if steps can be taken to improve the performance of the query. The following table lists terms that can appear in the output file and their significance.

Term	Significance
Query	Displays the executed query and indicates whether SET OPTIMIZATION was set to HIGH or LOW. If you SET OPTIMIZATION to LOW, the output displays the following uppercase string as the first line: `QUERY:{LOW}` If you SET OPTIMIZATION to HIGH, the output of SET EXPLAIN displays the following uppercase string as the first line: `QUERY:`
Directives followed	Lists the directives set for the query If the syntax for a directive is incorrect, the query is processed without the directive. In that case, the output shows DIRECTIVES NOT FOLLOWED in addition to DIRECTIVES FOLLOWED. For more information on the directives specified after this term, see the "Optimizer Directives" on page 4-222 or "SET OPTIMIZATION" on page 2-700.
Estimated cost	An estimate of the amount of work for the query The optimizer uses an estimate to compare the cost of one path with another. The estimate is a number the optimizer assigns to the selected access method. This number does not translate directly into time and cannot be used to compare different queries. It can be used, however, to compare changes made for the same query. When data distributions are used, a query with a higher estimate generally takes longer to run than one with a smaller estimate. In the case of a query and a subquery, two estimated cost figures are returned; the query figure also contains the subquery cost. The subquery cost is shown only so you can see the cost that is associated with the subquery.
Estimated number of rows returned	An estimate of the number of rows to be returned This number is based on information in the system catalog tables.

(1 of 2)

Term	Significance
Numbered list	The order in which tables are accessed, followed by the access method used (index path or sequential scan) When a query involves table inheritance, all the tables are listed under the supertable in the order they were accessed.
Index keys	The columns used as filters or indexes; the column name used for the index path or filter is indicated The notation *(Key Only)* indicates that all the desired columns are part of the index key, so a key-only read of the index could be substituted for a read of the actual table. The *Lower Index Filter* shows the key value where the index read begins. If the filter condition contains more than one value, an *Upper Index Filter* is shown for the key value where the index read stops.
Join method	When the query involves a join between two tables, the join method the optimizer used (Nested Loop or Dynamic Hash) is shown at the bottom of the output for that query. When the query involves a dynamic join of two tables, if the output contains the words *Build Outer*, the hash table is built on the first table listed (called the build table). If the words *Build Outer* do not appear, the hash table is built on the second table listed.

(2 of 2)

IDS

GLS

If the query uses a collating order other than the default for the **DB_LOCALE** setting, then the **DB_LOCALE** setting and the name of the other locale that is the basis for the collation in the query (as specified by the SET COLLATION statement) are both included in the output file. Similarly, if an index is not used because of its collation, the output file indicates this. ♦

Related Information

Related statements: SET OPTIMIZATION and UPDATE STATISTICS

For a description of the EXPLAIN and AVOID_EXECUTE optimizer directives, see "Explain-Mode Directives" on page 4-231.

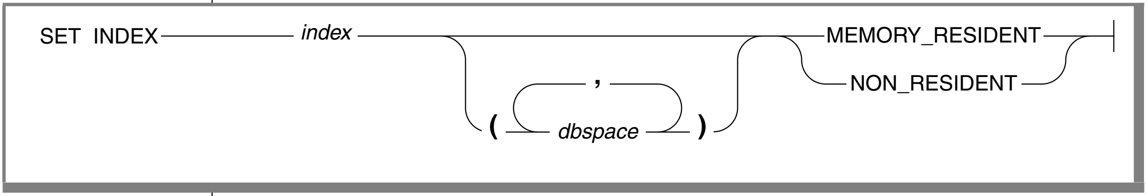
For discussions of SET EXPLAIN and of analyzing the output of the optimizer, see your *Performance Guide*.

+
XPS

SET INDEX

Use the SET INDEX statement to specify that one or more fragments of an index be resident in shared memory as long as possible.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	Dbspace to store the fragment	Must exist.	Identifier, p. 4-189
index	Index for which to change residency state	Must exist.	Database Object Name, p. 4-46

Usage

IDS

This statement was formerly supported by Dynamic Server, but it is ignored in current releases. Beginning with Version 9.40, Dynamic Server determines the residency status of indexes and tables automatically. ♦

The SET INDEX statement is a special case of the SET Residency statement. The SET Residency statement can also specify how long a table fragment remains resident in shared memory.

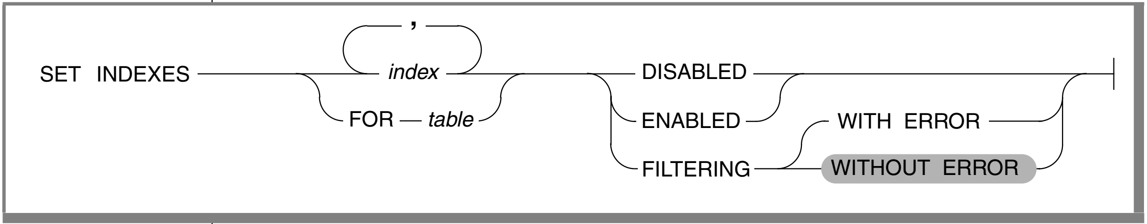
For the complete syntax and semantics of the SET INDEX statement, see "SET Residency" on page 2-708.

SET INDEXES

Use the SET INDEXES statement to enable or disable an index, or to change the filtering mode of a unique index.

Do not confuse the SET INDEXES statement with the SET INDEX statement.

Syntax



Element	Purpose	Restrictions	Syntax
table	Table whose indexes are all to be enabled, disabled, or changed in their filtering mode	Must exist	Database Object Name, p. 4-46
index	Index to be enabled, disabled, or changed in its filtering mode	Must exist	Database Object Name, p. 4-46

Usage

The SET INDEXES statement is a special case of the SET Database Object Mode statement. The SET Database Object Mode statement can also enable or disable a trigger or constraint, or change the filtering mode of a constraint.

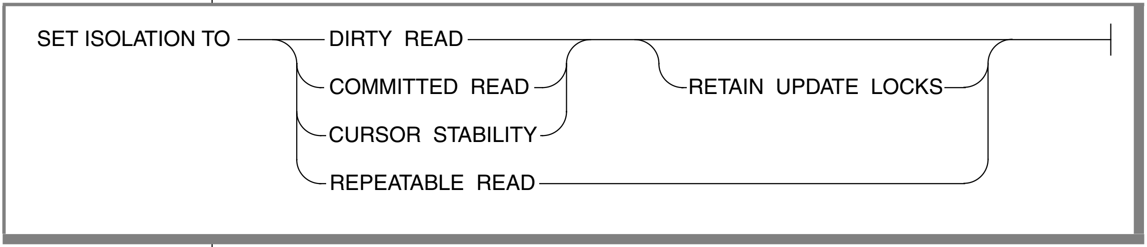
For the complete syntax and semantics of the SET INDEX statement, see "SET Database Object Mode" on page 2-652.

+

SET ISOLATION

Use the SET ISOLATION statement to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

Syntax



Usage

The SET ISOLATION statement is an Informix extension to the ANSI SQL-92 standard. If you want to set isolation levels through an ANSI-compliant statement, use the SET TRANSACTION statement instead. For a comparison of these two statements, see "SET TRANSACTION" on page 2-720.

The database isolation level affects read concurrency when rows are retrieved from the database. The database server uses shared locks to support different levels of isolation among processes attempting to access data.

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with rows that you are updating or deleting. If another process attempts to update or delete rows that you are reading with an isolation level of Repeatable Read, that process is denied access to those rows.

E/C

In ESQL/C, cursors that are open when SET ISOLATION executes might or might not use the new isolation level when rows are retrieved. Any isolation level that was set from the time the cursor was opened until the application fetches a row might be in effect. The database server might have read rows into internal buffers and internal temporary tables using the isolation level that was in effect at that time. To ensure consistency and reproducible results, close any open cursors before you execute the SET ISOLATION statement. ♦

Informix Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Using the Dirty Read Option

Use the Dirty Read option to copy rows from the database whether or not there are locks on them. The program that fetches a row places no locks and it respects none. Dirty Read is the only isolation level available to databases that do not have transactions.

This isolation level is most appropriate for static tables that are used for queries, that is, tables where data is not being modified, because it provides no isolation. With Dirty Read, the program might return a phantom row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row.

Using the Committed Read Option

Use the Committed Read option to guarantee that every retrieved row is committed in the table at the time that the row is retrieved. This option does not place a lock on the fetched row. Committed Read is the default level of isolation in a database with logging that is not ANSI compliant.

Committed Read is appropriate to use when each row of data is processed as an independent unit, without reference to other rows in the same table or in other tables.

Using the Cursor Stability Option

Use the Cursor Stability option to place a shared lock on the fetched row, which is released when you fetch another row or close the cursor. Another process can also place a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row. Such row stability is important when the program updates another table based on the data it reads from the row.

If you set the isolation level to Cursor Stability, but you are not using a transaction, the Cursor Stability isolation level acts like the Committed Read isolation level.

Using the Repeatable Read Option

Use the Repeatable Read option to place a shared lock on every row that is selected during the transaction. Another process can also place a shared lock on a selected row, but no other process can modify any selected row during your transaction or insert a row that meets the search criteria of your query during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Repeatable Read is the default isolation level in an ANSI-compliant database.

Repeatable Read isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most.

Default Isolation Levels

The default isolation level for a particular database is established when you create the database according to database type. The following list describes the default isolation level for each database type.

Isolation Level	Database Type
Dirty Read	Default level in a database without logging
Committed Read	Default level in a logged database that is not ANSI compliant
Repeatable Read	Default level in an ANSI-compliant database

The default level remains in effect until you issue a SET ISOLATION statement. After a SET ISOLATION statement executes, the new isolation level remains in effect until one of the following events occurs:

- You enter another SET ISOLATION statement.
- You open another database that has a default isolation level different from the level that your last SET ISOLATION statement specified.
- The program ends.

Using the RETAIN UPDATE LOCKS Option

Use the RETAIN UPDATE LOCKS option to affect the behavior of the database server when it handles a SELECT ... FOR UPDATE statement.

In a database with the isolation level set to Dirty Read, Committed Read, or Cursor Stability, the database server places an update lock on a fetched row of a SELECT ... FOR UPDATE statement. When you turn on the RETAIN UPDATE LOCKS option, the database server retains the update lock until the end of the transaction rather than release it at the next subsequent FETCH or when the cursor is closed. This option prevents other users from placing an exclusive lock on the updated row before the current user reaches the end of the transaction.

You can use this option to achieve the same locking effects but avoid the overhead of dummy updates or the repeatable read isolation level.

You can turn this option on or off at any time during the current session.

You can turn the option off by resetting the isolation level without using the RETAIN UPDATE LOCKS keywords.

For more information on update locks, see "Locking Considerations" on page 2-765.

Turning the Option Off In the Middle of a Transaction

If you set the RETAIN UPDATE LOCKS option to OFF after a transaction has begun, but before the transaction has been committed or rolled back, several update locks might still exist.

Switching OFF the feature does not directly release any update lock. When you turn this option off, the database server reverts to normal behavior for the three isolation levels. That is, a FETCH statement releases the update lock placed on a row by the immediately preceding FETCH statement, and a closed cursor releases the update lock on the current row.

Update locks placed by earlier FETCH statements are not released unless multiple update cursors are present within the same transaction. In this case, a subsequent FETCH could also release older update locks of other cursors.

Effects of Isolation Levels

You cannot set the database isolation level in a database that does not have logging. Every retrieval in such a database occurs as a Dirty Read.

You can issue a SET ISOLATION statement from a client computer only after a database is opened.

The data retrieved from a BYTE or TEXT column can vary, depending on the database isolation level. Under Dirty Read or Committed Read levels of isolation, a process can read a BYTE or TEXT column that is either deleted (if the DELETE is not yet committed) or in the process of being deleted. Under these isolation levels, deleted data is readable under certain conditions. For information about these conditions, see the *Administrator's Guide*.

DB

When you use DB-Access, as you use higher levels of isolation, lock conflicts occur more frequently. For example, if you use Cursor Stability, more lock conflicts occur than if you use Committed Read. ♦

E/C

Using a scroll cursor in an ESQL/C transaction, you can force consistency between your temporary table and the database table either by setting the level to Repeatable Read or by locking the entire table during the transaction.

If you use a scroll cursor WITH HOLD in a transaction, you cannot force consistency between your temporary table and the database table. A table-level lock or locks that are set by Repeatable Read are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, but the retrieved data in the temporary table might be inconsistent with the actual data. ♦

Warning: *Do not use nonlogging tables within a transaction. If you need to use a nonlogging table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.*

Related Information

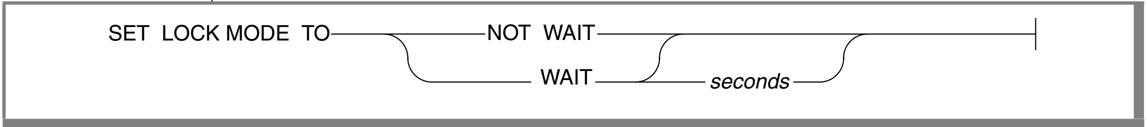
Related statements: CREATE DATABASE, SET LOCK MODE, and SET TRANSACTION

For a discussion on how to set the isolation level, see the *IBM Informix Guide to SQL: Tutorial*.

SET LOCK MODE

Use the SET LOCK MODE statement to define how the database server handles a process that tries to access a locked row or table.

Syntax



Element	Purpose	Restrictions	Syntax
seconds	Maximum number of seconds that a process waits for a lock to be released before issuing an error	Valid only if shorter than system default.	Literal Number, p. 4-216

Usage

This statement overrides any setting of the **IFX_TABLE_LOCKMODE** environment variable or DEF_TABLE_LOCKMODE setting in ONCONFIG. You can direct the response of the database server in the following ways when a process tries to access a locked row or table.

Lock Mode	Effect
NOT WAIT	Database server ends the operation immediately and returns an error code. This condition is the default.
WAIT	Database server suspends the process until the lock releases.
WAIT *seconds*	Database server suspends the process until the lock releases or until the waiting period ends. If the lock remains after the waiting period, the operation ends and an error code is returned.

In the following example, the user specifies that if the process requests a locked row, the operation should end immediately and an error code should be returned:

```
SET LOCK MODE TO NOT WAIT
```

In the following example, the user specifies that the process should be suspended until the lock is released:

```
SET LOCK MODE TO WAIT
```

The next example sets an upper limit of 17 seconds on the length of any wait:

```
SET LOCK MODE TO WAIT 17
```

WAIT Clause

The WAIT clause causes the database server to suspend the process until the lock is released or until a specified number of seconds have passed without the lock being released.

The database server protects against the possibility of a deadlock when you request the WAIT option. Before the database server suspends a process, it checks whether suspending the process could create a deadlock. If the database server discovers that a deadlock could occur, it ends the operation (overruling your instruction to wait) and returns an error code. In the case of either a suspected or actual deadlock, the database server returns an error.

Cautiously use the unlimited waiting period that was created when you specify the WAIT option without *seconds*. If you do not specify an upper limit, and the process that placed the lock somehow fails to release it, suspended processes could wait indefinitely. Because a true deadlock situation does not exist, the database server does not take corrective action.

In a networked environment, the DBA uses the ONCONFIG parameter DEADLOCK_TIMEOUT to establish a default value for *seconds*. If you use a SET LOCK MODE statement to set an upper limit, your value applies only when your waiting period is shorter than the system default. The number of seconds that the process waits applies only if you acquire locks within the current database server and a remote database server within the same transaction.

Related Information

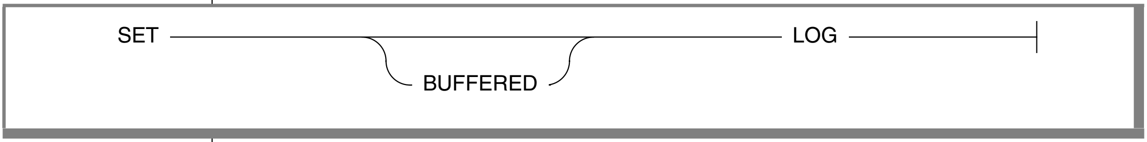
Related statements: LOCK TABLE, SET ISOLATION, SET TRANSACTION, and UNLOCK TABLE

For a discussion on how to set the lock mode, see the *IBM Informix Guide to SQL: Tutorial*.

SET LOG

Use the SET LOG statement to change your database logging mode from buffered transaction logging to unbuffered transaction logging or vice versa.

Syntax



Usage

You activate transaction logging when you create a database or add logging to an existing database. These transaction logs can be buffered or unbuffered.

Buffered logging is a type of logging that holds transactions in a memory buffer until the buffer is full, regardless of when the transaction is committed or rolled back. The database server provides this option to speed up operations by reducing the number of disk writes.

Warning: *You gain a marginal increase in efficiency with buffered logging, but you incur some risk. In the event of a system failure, the database server cannot recover any completed transactions in the memory buffer that had not been written to disk.*

The SET LOG statement in the following example changes the transaction logging mode to buffered logging:

```
SET BUFFERED LOG
```

Unbuffered logging is a type of logging that does not hold transactions in a memory buffer. As soon as a transaction ends, the database server writes the transaction to disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions, but not those still in the buffer. The default condition for transaction logs is unbuffered logging.

The SET LOG statement in the following example changes the transaction logging mode to unbuffered logging:

```
SET LOG
```

The SET LOG statement redefines the mode for the current session only. The default mode, which the database administrator sets with the **ondblog** utility, remains unchanged.

The buffering option does not affect retrievals from external tables. For distributed queries, a database with logging can retrieve only from databases with logging, but it makes no difference whether the databases use buffered or unbuffered logging.

ANSI

An ANSI-compliant database cannot use buffered logging.

You cannot change the logging mode of ANSI-compliant databases. If you created a database with the WITH LOG MODE ANSI keywords, you cannot later use the SET LOG statement to change the logging mode to buffered or unbuffered transaction logging. ♦

Related Information

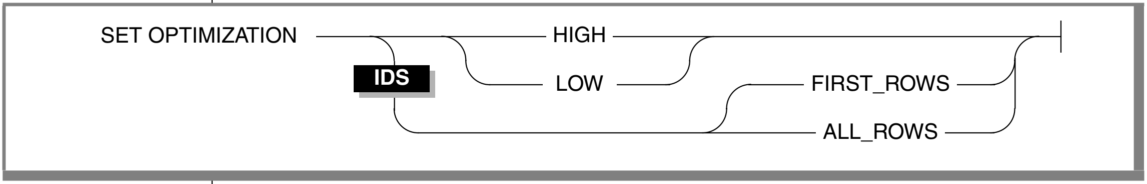
Related statement: CREATE DATABASE

+

SET OPTIMIZATION

Use the SET OPTIMIZATION statement to specify how much time the optimizer spends developing a query plan or specifying optimization goals.

Syntax



Usage

You can execute a SET OPTIMIZATION statement at any time. The specified optimization level carries across databases on the current database server. The option that you specify remains in effect until you issue another SET OPTIMIZATION statement or until the program ends. The default database server optimization level for the amount of time that the query optimizer spends determining the query plan is HIGH.

IDS

The default optimization goal is ALL_ROWS. Although you can set only one option at a time, you can issue two SET OPTIMIZATION statements: one that specifies the time the optimizer spends to determine the query plan and one that specifies the optimization goal of the query. ♦

HIGH and LOW Options

The HIGH and LOW options determine how much time the query optimizer spends to determine the query plan:

- HIGH

  This option directs the optimizer to use a sophisticated, cost-based algorithm that examines all reasonable query-plan choices and selects the best overall alternative.

  For large joins, this algorithm can incur more overhead than you desire. In extreme cases, you can run out of memory.
- LOW

  This option directs the optimizer to use a less sophisticated, but faster, optimization algorithm. This algorithm eliminates unlikely join strategies during the early stages of optimization and reduces the time and resources spent during optimization.

  When you specify a low level of optimization, the database server might not select the optimal strategy because the strategy was eliminated from consideration during the early stages of the algorithm.

IDS

FIRST_ROWS and ALL_ROWS Options

The FIRST_ROWS and ALL_ROWS options relate to the optimization goal of the query:

- FIRST_ROWS

  This option directs the optimizer to choose the query plan that returns the first result record as soon as possible.
- ALL_ROWS

  This option directs the optimizer to choose the query plan that returns all the records as quickly as possible.

You can also specify the optimization goal of a specific query with the optimization-goal directive. For more information, see "Optimizer Directives" on page 4-222.

Optimizing SPL Routines

For SPL routines that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to HIGH when you create the SPL routine. This step stores the best query plans for the SPL routine. Then execute a SET OPTIMIZATION LOW statement before you execute the SPL routine. The SPL routine then uses the optimal query plans and runs at the more cost-effective rate.

Examples

The following example shows optimization across a network. The **central** database (on the **midstate** database server) is to have LOW optimization; the **western** database (on the **rockies** database server) is to have HIGH optimization.

```
CONNECT TO 'central@midstate';
SET OPTIMIZATION LOW;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'western@rockies';
SET OPTIMIZATION HIGH;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'wyoming@rockies';
SELECT * FROM customer;
```

The **wyoming** database is to have HIGH optimization because it resides on the same database server as the **western** database. The code does not need to re-specify the optimization level for the **wyoming** database because the **wyoming** database resides on the **rockies** database server like the **western** database.

IDS

The following example directs the optimizer to use the most time to determine a query plan and to then return the first rows of the result as soon as possible:

```
SET OPTIMIZATION LOW;
SET OPTIMIZATION FIRST_ROWS;
SELECT lname, fname, bonus
FROM sales_emp, sales
WHERE sales.empid = sales_emp.empid AND bonus > 5,000
ORDER BY bonus DESC
```

♦

Related Information

Related statements: SET EXPLAIN and UPDATE STATISTICS

IDS

For information on other methods by which you can alter the query plan of the optimizer, see "Optimizer Directives" on page 4-222. ♦

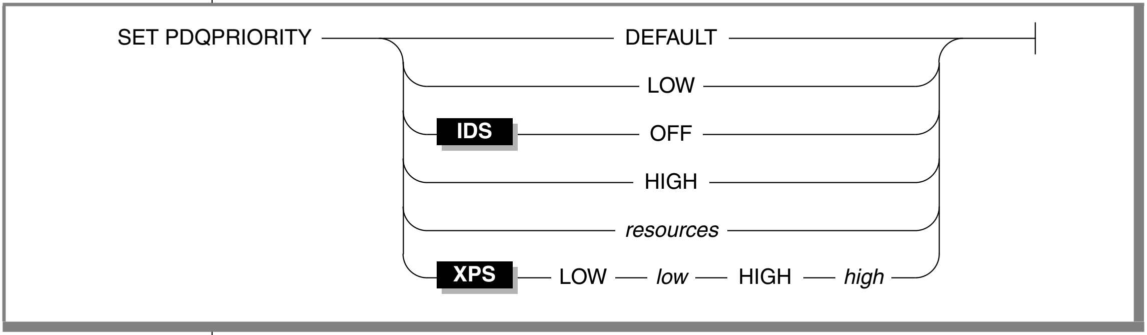
For more information on how to optimize queries, see your *Performance Guide*.

+

SET PDQPRIORITY

The SET PDQPRIORITY statement allows an application to set the query priority level dynamically within a routine.

Syntax



Element	Purpose	Restrictions	Syntax
high	Integer value that specifies the desired resource allocation	Must be in the range `1` to `100`, and not less than the *low* value.	Literal Number, p. 4-216
low	Integer that specifies the minimum resource allocation	Must be in the range `1` to `100`, and not greater than the *high* value.	Literal Number, p. 4-216
resources	Integer that specifies the query priority level and the percent of resources to process the query	Can range from `-1` to `100`. See also "Allocating Database Server Resources" on page 2-705.	Literal Number, p. 4-216

Usage

The SET PDQPRIORITY statement overrides the **PDQPRIORITY** environment variable (but has lower precedence than the ONCONFIG configuration parameter MAX_PDQPRIORITY). The scope of SET PDQPRIORITY is local to the routine, and does not affect other routines within the same session.

IDS

Set PDQ priority to a value that is less than the quotient of 100 divided by the maximum number of prepared statements. For example, if two prepared statements are active, you should set the PDQ priority to less than `50`. ♦

SPL

The SET PDQPRIORITY statement is not supported in SPL routines. ♦

XPS

In Extended Parallel Server, you can use SET PDQPRIORITY to set PDQ priority at runtime to a value greater than `0` when you need more memory for operations such as sorts, forming groups, and index builds.
For guidelines on which values to use, see your *Performance Guide*. ♦

Allocating Database Server Resources

For example, assume that the DBA sets the MAX_PDQPRIORITY parameter to `50`. Then a user enters the following SET PDQPRIORITY statement to set the query priority level to 80 percent of resources:

```
SET PDQPRIORITY 80
```

When it processes the query, the database server uses the MAX_PDQPRIORITY value to factor the query priority level set by the user. The database server silently processes the query with a priority level of `40`. This priority level represents 50 percent of the 80 percent of resources that the user specifies.

The following keywords are supported by the SET PDQPRIORITY statement.

Keyword	Effect
DEFAULT	Uses the setting of the **PDQPRIORITY** environment variable
LOW	Data values are fetched from fragmented tables in parallel. (In Dynamic Server, when you specify LOW, the database server uses no other forms of parallelism.)
OFF	PDQ is turned off (Dynamic Server only). The database server uses no parallelism. OFF is the default if you use neither the **PDQPRIORITY** environment variable nor the SET PDQPRIORITY statement.
HIGH	The database server determines an appropriate PDQPRIORITY value, based on factors that include the number of available processors, the fragmentation of the tables being queried, the complexity of the query, and others. IBM reserves the right to change the performance behavior of queries when HIGH is specified in future releases.

You can specify an integer in the range from `-1` to `100` to indicate a query priority level as the percent of database server resources to process the query. Resources include the amount of memory and the number of processors. The higher the number you specify, the more resources the database server uses.

Use of more resources usually indicates better performance for a given query. Using excessive resources, however, can cause contention for resources and remove resources from other queries, so that degraded performance results. With the *resources* option, the following values are numeric equivalents of the keywords that indicate query priority level.

Value	Equivalent Keyword-Priority Level
`-1`	DEFAULT
`0`	OFF (Dynamic Server only)
`1`	LOW

The following statements are equivalent. The first statement uses the keyword LOW to establish a low query-priority level. The second uses a value of `1` in the *resources* parameter to establish a low query-priority level.

```
SET PDQPRIORITY LOW;

SET PDQPRIORITY 1;
```

XPS

Using a Range of Values

In Extended Parallel Server, when you specify a range of values in SET PDQPRIORITY, you allow the Resource Grant Manager (RGM) some discretion in allocating resources.

The *high* value in the range is the desired resource allocation, while the *low* value is the minimum acceptable resource allocation for the query. If the *low* value exceeds the available system resources, the RGM blocks the query. Otherwise, the RGM chooses the largest PDQ priority in the specified range that does not exceed available resources.

Related Information

For information about configuration parameters and about the Resource Grant Manager, see your *Administrator's Guide* and your *Performance Guide*.

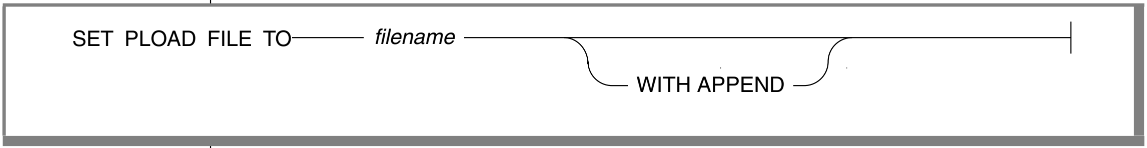
For information about the **PDQPRIORITY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

+
XPS

SET PLOAD FILE

Use the SET PLOAD FILE statement to prepare a log file for a session of loading or unloading data from or to an external table. The log file records summary statistics about each load or unload job. The log file also lists any reject files created during a load job.

Syntax



Element	Purpose	Restrictions	Syntax
filename	Name of the log file. If you specify no log *filename*, then log information is written to **/dev/null**.	If the file cannot be opened for writing, an error results.	Platform-dependent

Usage

The WITH APPEND option allows you to append new log information to the existing log file.

Each time a session closes, the log file for that session also closes. If you issue more than one SET PLOAD FILE statement within a session, each new statement closes a previously opened log file and opens a new log file.

If you invoke a SET PLOAD FILE statement with a simple filename on a local database, the output file is located in your current directory. If your current database is on a remote database server, then the output file is located in your home directory on the remote database server, on the coserver where the initial connection was made. If you provide a full pathname for the file, it is placed in the directory and file specified on the remote server.

Related Information

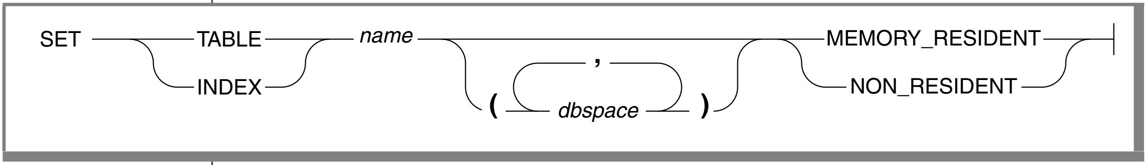
Related statement: CREATE EXTERNAL TABLE

+
XPS

SET Residency

Use the SET Residency statement to specify that one or more fragments of a table or index be resident in shared memory as long as possible.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The dbspace where fragment resides	The dbspace must exist	Identifier, p. 4-189
name	Table or index for which the residency state will be changed	Table or index must exist	Database Object Name, p. 4-46

Usage

IDS

This statement was formerly supported by Dynamic Server, but it is ignored in current releases. Beginning with Version 9.40, Dynamic Server determines the residency status of indexes and tables automatically. ♦

The SET Residency statement allows you to specify the tables, indexes, and data fragments that you want to remain in the buffer as long as possible. When a free buffer is requested, pages that are declared MEMORY_RESIDENT are considered last for page replacement.

The default state is nonresident. The residency state is persistent while the database server is up. That is, each time the database server is started, you must specify the database objects that you want to remain in shared memory.

After a table, index, or data fragment is set to MEMORY_RESIDENT, the residency state remains in effect until one of the following events occurs:

- You use SET Residency to set the database object to NON_RESIDENT.
- The database object is dropped.
- The database server is taken offline.

Only user **informix** can set or change the residency state of a database object.

Residency and the Changing Status of Fragments

If new fragments are added to a resident table, the fragments are not marked automatically as resident. You must issue the SET Residency statement for each new fragment or reissue the statement for the entire table.

Similarly, if a resident fragment is detached from a table, the residency state of the fragment remains unchanged. If you want the residency state to change to nonresident, you must issue the SET Residency statement to declare the specific fragment (or the entire table) as nonresident.

Examples

The next example shows how to set the residency status of an entire table:

```
SET TABLE tab1 MEMORY_RESIDENT
```

For fragmented tables or indexes, you can specify residency for individual fragments as the following example shows:

```
SET INDEX index1 (dbspace1, dbspace2) MEMORY_RESIDENT;
SET TABLE tab1 (dbspace1) NON_RESIDENT
```

This example specifies that the **tab1** fragment in **dbspace1** is not to remain in shared memory while the **index1** fragments in **dbspace1** and **dbspace2** are to remain in shared memory as long as possible.

Related Information

Related statement: ALTER FRAGMENT

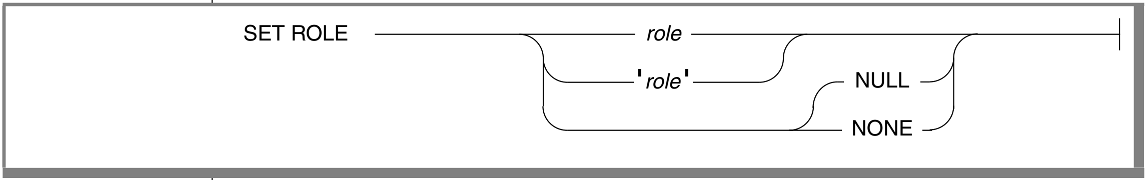
For information on how to monitor the residency status of tables, indexes, and fragments, refer to your *Administrator's Guide*.

+

SET ROLE

Use the SET ROLE statement to enable the privileges of a role.

Syntax



Element	Purpose	Restrictions	Syntax
role	The role to be enabled	Must have been created with the CREATE ROLE statement. A role name that is enclosed between quotation marks is case sensitive.	Identifier, p. 4-189

Usage

Any user who is granted a role can enable the role using the SET ROLE statement. You can only enable one role at a time. If you execute the SET ROLE statement after a role is already set, the new role replaces the old role.

All users are, by default, assigned the role NULL or NONE. In this context, NULL and NONE are synonymous. The roles NULL and NONE have no privileges. When you set the role to NULL or NONE, you disable the current role.

When you set a role, you gain the privileges of the role, in addition to the privileges of PUBLIC and your own privileges. If a role is granted to another role, you gain the privileges of both roles, in addition to those of PUBLIC and your own privileges.

After the SET ROLE statement executes successfully, the role remains effective until the current database is closed or the user executes another SET ROLE statement. Additionally, only the user, not the role, retains ownership of any database objects, such as tables, that were created during the session.

A role is on scope only within the current database. You cannot use privileges that you acquire from a role to access data in another database.

For example, if you have privileges from a role in the database named **acctg**, and you execute a distributed query over the databases named **acctg** and **inventory**, your query cannot access the data in the **inventory** database unless you were granted privileges in the **inventory** database.

You cannot execute the SET ROLE statement while in a transaction. If the SET ROLE statement is executed while a transaction is active, an error occurs.

If the SET ROLE statement is executed as a part of a trigger or SPL routine, and the owner of the trigger or SPL routine was granted the role with the WITH GRANT OPTION, the role is enabled even if you are not granted the role. For example, this code fragment sets a role and then relinquishes it after a query:

```
EXEC SQL set role engineer;
EXEC SQL select fname, lname, project
      INTO :efname, :elname, :eproject FROM projects
      WHERE project_num > 100 AND lname = 'Larkin';
printf ("%s is working on %s\n", efname, eproject);
EXEC SQL set role null;
```

XPS

Setting the Default Role

In Extended Parallel Server, you can be define a default role for a user without having to set it explicitly each time a session is initialized. After a role has been created, it can be granted to a user using the **sysdbopen()** procedure. The next example sets the default role to **Engineer** for all users:

```
CREATE PROCEDURE PUBLIC.sysdbopen()
   SET ROLE Engineer;
END PROCEDURE;
```

You can also define a default role for individual users, as in this example:

```
CREATE PROCEDURE newuser.sysdbopen()
   SET ROLE DBmaster;
END PROCEDURE;
```

For more information on using the **sysdbopen()** procedure, see "Using sysbdopen( ) and sysdbclose( ) Stored Procedures" on page 2-189.

Related Information

Related statements: CREATE ROLE, DROP ROLE, GRANT, and REVOKE

For a discussion of how to use roles, see the *IBM Informix Guide to SQL: Tutorial*.

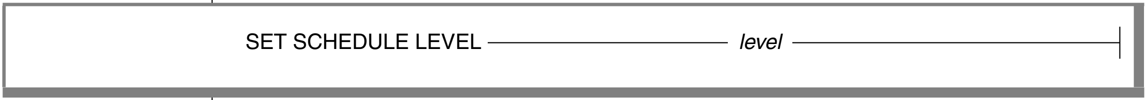
+

XPS

SET SCHEDULE LEVEL

The SET SCHEDULE LEVEL statement specifies the scheduling level of a query when queries are waiting to be processed.

Syntax



Element	Purpose	Restrictions	Syntax
level	Integer value that specifies the scheduling priority of a query	Must be between `1` and `100`. If the value falls outside the range of `1` and `100`, the database server uses the default value of `50`.	Literal Number, p. 4-216

Usage

The highest priority level is `100`. That is, a query at level `100` is more important than a query at level `1`. In general, the Resource Grant Manager (RGM) processes a query with a higher scheduling level before a query with a lower scheduling level. The exact behavior of the RGM is also influenced by the setting of the DS_ADM_POLICY configuration parameter.

Related Information

Related statement: SET PDQPRIORITY

For information about the Resource Grant Manager, see your *Administrator's Guide*.

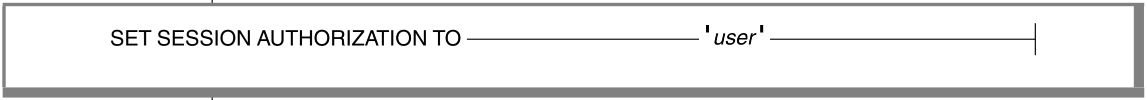
For information about the DS_ADM_POLICY configuration parameter, see your *Administrator's Reference*.

IDS

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement lets you change the user name under which database operations are performed in the current session.

Syntax



Element	Purpose	Restrictions	Syntax
user	User name by which database operations will be performed in the current session	You must specify a valid user name and put quotation marks around that name.	Identifier, p. 4-189

Usage

This statement allows a user with the DBA privilege to bypass the privileges that protect database objects. You can use this statement to gain access to a table and adopt the identity of a table owner to grant access privileges. You must obtain the DBA privilege before you start a session in which you use this statement. Otherwise, this statement returns an error.

This statement requires the DBA privilege, which you must obtain from the DBA before the start of your current session. The new identity remains in effect in the current database until you execute another SET SESSION AUTHORIZATION statement or until you close the current database. When you use this statement, the specified *user* must have the Connect privilege on the current database. Additionally, the DBA cannot set the authorization to PUBLIC or to any defined role in the current database.

Setting a session to another user causes a change in a user name in the current active database server. These users are, as far as this database server process is concerned, completely dispossessed of any privileges that they might have while accessing the database server through some administrative utility. Additionally, the new session user is not able to initiate an administrative operation (execute a utility, for example) by virtue of the acquired identity.

After the SET SESSION AUTHORIZATION statement successfully executes, the user must use the SET ROLE statement to assume a role granted to the current user. Any role enabled by a previous user is relinquished.

After SET SESSION AUTHORIZATION successfully executes, all owner-privileged UDRs created while using the new identity are given RESTRICTED mode. For more information on RESTRICTED mode, see the **sysprocedures** system catalog table in the *IBM Informix Guide to SQL: Reference*.

When you assume the identity of another user by executing the SET SESSION AUTHORIZATION statement, you can perform operations in the current database only. You cannot perform an operation on a database object outside the current database, such as a remote table. In addition, you cannot execute a DROP DATABASE or RENAME DATABASE statement, even if the database is owned by the real or effective user.

You can use this statement either to obtain access to the data directly or to grant the database-level or table-level privileges needed for the database operation to proceed. The following example shows how to use the SET SESSION AUTHORIZATION statement to obtain table-level privileges:

```
SET SESSION AUTHORIZATION TO 'cathl';
GRANT ALL ON customer TO mary;
SET SESSION AUTHORIZATION TO 'mary';
UPDATE customer SET fname = 'Carl' WHERE lname = 'Pauli';
```

SET SESSION AUTHORIZATION and Transactions

If your database is not ANSI compliant, you must issue the SET SESSION AUTHORIZATION statement outside a transaction. If you issue the statement within a transaction, you receive an error message.

ANSI

In an ANSI-compliant database, you can execute the SET SESSION AUTHORIZATION statement as long as you have not executed a statement that initiates an implicit transaction (for example, CREATE TABLE or SELECT). Statements that do not initiate an implicit transaction are statements that do not acquire locks or log data (for example, SET EXPLAIN and SET ISOLATION). You can execute the SET SESSION AUTHORIZATION statement immediately after a DATABASE statement or a COMMIT WORK statement. ♦

Related Information

Related statements: CONNECT, DATABASE, GRANT, and SET ROLE

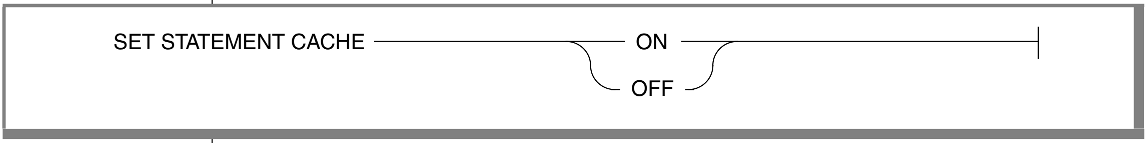
+

IDS

SET STATEMENT CACHE

Use the SET STATEMENT CACHE statement to turn caching on or off for the current session.

Syntax



Usage

You can use the SET STATEMENT CACHE statement to turn caching in the SQL statement cache ON or OFF for the current session. The statement cache stores in a buffer identical statements that are repeatedly executed in a session. Only data manipulation language (DML) statements (DELETE, INSERT, UPDATE, or SELECT) can be stored in the statement cache.

This mechanism allows qualifying statements to bypass the optimization stage and parsing stage, and avoid recompiling, which can reduce memory consumption and can improve query processing time.

Precedence and Default Behavior

SET STATEMENT CACHE takes precedence over the **STMT_CACHE** environment variable and the STMT_CACHE configuration parameter. You must enable the SQL statement cache, however, either by setting the STMT_CACHE configuration parameter or by using the **onmode** utility, before the SET STATEMENT CACHE statement can execute successfully.

When you issue a SET STATEMENT CACHE ON statement, the SQL statement cache remains in effect until you issue a SET STATEMENT CACHE OFF statement or until the program ends. If you do not use SET STATEMENT CACHE, the default behavior depends on the setting of the **STMT_CACHE** environment variable or the STMT_CACHE configuration parameter.

Turning the Cache ON

Use the ON option to enable the SQL statement cache. When the SQL statement cache is enabled, each statement that you execute passes through the SQL statement cache to determine if a matching cache entry is present. If so, the database server uses the cached entry to execute the statement.

If the statement has no matching entry, the database server tests to see if it qualifies for entry into the cache. For the conditions a statement must meet to enter into the cache, see "Statement Qualification" on page 2-717.

Restrictions on Matching Entries in the SQL Statement Cache

When the database server considers whether or not a statement is identical to a statement in the SQL statement cache, the following items must match:

- Lettercase
- Comments
- White space
- Optimization settings
  - SET OPTIMIZATION statement options
  - Optimizer directives
  - Settings of the **OPTCOMPIND** environment variable or the OPTCOMPIND configuration parameter in the ONCONFIG file
- Parallelism settings
  - SET PDQPRIORITY statement options or settings of the **PDQPRIORITY** environment variable
- Query text strings
- Literals

If an SQL statement is semantically equivalent to a statement in the SQL statement cache but has different literals, the statement is not considered identical and qualifies for entry into the cache. For example, the following SELECT statements are not identical:

```
SELECT col1, col2 FROM tab1 WHERE col1=3;

SELECT col1, col2 FROM tab1 WHERE col1=5;
```

In this example, both statements are entered into the SQL statement cache.

Host-variable names, however, are insignificant. For example, the following select statements are considered identical:

```
SELECT * FROM tab1 WHERE x = :x AND y = :y;

SELECT * FROM tab1 WHERE x = :p AND y = :q;
```

In the previous example, although the host names are different, the statements qualify, because the case, query text strings, and white space match. Performance does not improve, however, because each statement has already been parsed and optimized by the PREPARE statement.

Turning the Cache OFF

The OFF option disables the SQL statement cache. When you turn caching off for your session, no SQL statement cache code is executed for that session.

The SQL statement cache is designed to save memory in environments where identical queries are executed repeatedly and where schema changes are infrequent. If this is not the case, you might want to turn the SQL statement cache off to avoid the overhead of caching. For example, if you have little cache cohesion, that is, when relatively few matches but many new entries into the cache exist, the cache management overhead is high. In this case, turn the SQL statement cache off.

If you know that you are executing many statements that do not qualify for the SQL statement cache, you might want to disable it and avoid the overhead of testing to see if each statement qualifies for entry into the cache.

Statement Qualification

A statement that can be cached in the SQL statement cache (and consequently, one that can match a statement that already appears in the SQL statement cache) must meet the following conditions:

- It must be a SELECT, INSERT, UPDATE, or DELETE statement.
- It must contain only built-in data types (excluding BLOB, BOOLEAN, BYTE, CLOB, LVARCHAR, or TEXT).
- It must contain only built-in operators.
- It cannot contain user-defined routines.
- It cannot contain temporary or remote tables.

- It cannot contain subqueries in the select list.
- It cannot be part of a multistatement PREPARE.
- It cannot have user-permission restrictions on target columns.
- In an ANSI-compliant database, it must contain fully qualified object names.
- It cannot require re-optimization.

Cache Insertion After "n" Hits

A qualified SQL statement is fully inserted into the SQL statement cache only after the database server counts a configured number of references, or hits, to that statement.

The database administrator (DBA) can specify a minimum number of hits for cache insertion using the configuration parameter, STMT_CACHE_HITS. By doing so, the DBA excludes one-time-only ad hoc queries from full insertion into the SQL statement cache, thus lowering cache-management overhead.

Enabling or Disabling Insertions After Size Exceeds Configured Limit

The DBA can prevent the insertion of qualified SQL statements into the SQL statement cache when the cache size reaches its configured size (STMT_CACHE_SIZE configuration parameter) by setting the configuration parameter, STMT_CACHE_NOLIMIT.

Related Information

For information on optimization settings, see "SET OPTIMIZATION" on page 2-700 and "Optimizer Directives" on page 4-222.

For information about the **STMT_CACHE** environment variable, see the *IBM Informix Guide to SQL: Reference*.

For more information about the configuration parameters and the **onmode** utility, see your *Administrator's Reference*.

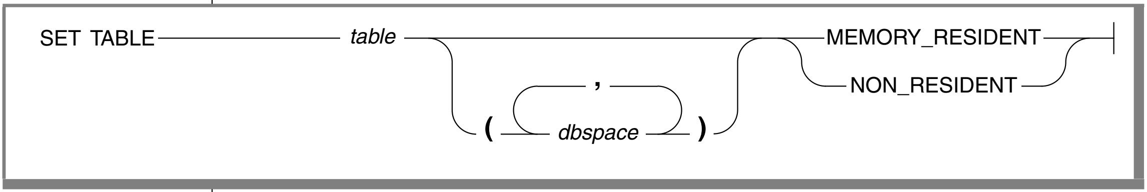
For more information on the performance implications of this feature, on when and how to use the SQL statement cache, on how to monitor the cache with the **onstat** options, and on how to tune the configuration parameters, see your *Performance Guide*.

+
XPS

SET TABLE

Use the SET TABLE statement to specify that one or more fragments of a table be resident in shared memory as long as possible.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	Name of the dbspace in which the fragment resides	The dbspace must exist.	Identifier, p. 4-189
table	Name of the table for which you want to change the residency state	The table must exist.	Database Object Name, p. 4-46

Usage

IDS

This statement was formerly supported by Dynamic Server, but it is ignored in current releases. Beginning with Version 9.40, Dynamic Server determines the residency status of indexes and tables automatically. ♦

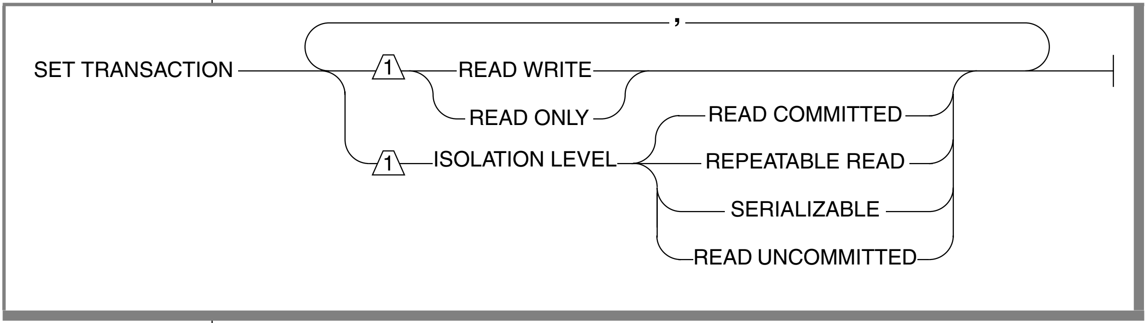
The SET TABLE statement is a special case of the SET Residency statement. The SET Residency statement can also specify how long an index fragment remains resident in shared memory.

For the complete syntax and semantics of the SET TABLE statement, see "SET Residency" on page 2-708.

SET TRANSACTION

Use the SET TRANSACTION statement to define isolation levels and to define the access mode of a transaction (read-only or read-write).

Syntax



Usage

SET TRANSACTION is valid only in databases with logging. You can issue a this statement from a client computer only after a database is opened. The database isolation level affects concurrency among processes that attempt to access the same rows simultaneously from the database. The database server uses shared locks to support different levels of isolation among processes that are attempting to read data, as the following list shows:

- Read Uncommitted
- Read Committed
- (ANSI) Repeatable Read
- Serializable

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with such rows, but the access mode does affect whether you can update or delete rows.

If another process attempts to update or delete rows that you are reading with an isolation level of Serializable or (ANSI) Repeatable Read, that process will be denied access to those rows.

Comparing SET TRANSACTION with SET ISOLATION

The SET TRANSACTION statement complies with ANSI SQL-92. This statement is similar to the Informix SET ISOLATION statement; however, the SET ISOLATION statement is not ANSI compliant and does not provide access modes. In fact, the isolation levels that you can set with the SET TRANSACTION statement are almost parallel to the isolation levels that you can set with the SET ISOLATION statement, as the following table shows.

SET TRANSACTION Isolation Level	SET ISOLATION Isolation Level
Read Uncommitted	Dirty Read
Read Committed	Committed Read
[*Not supported*]	Cursor Stability
(ANSI) Repeatable Read	(Informix) Repeatable Read
Serializable	(Informix) Repeatable Read

Another difference between SET TRANSACTION and SET ISOLATION is the behavior of the isolation levels within transactions. You can issue SET TRANSACTION only once for a transaction. Any cursors that are opened during that transaction are guaranteed that isolation level (or access mode, if you are defining an access mode). With SET ISOLATION, after a transaction is started, you can change the isolation level more than once within the transaction.

The following examples illustrate this difference in the behavior of the SET ISOLATION and SET TRANSACTION statements:

```
EXEC SQL BEGIN WORK;
EXEC SQL SET ISOLATION TO DIRTY READ;
EXEC SQL SELECT ... ;
EXEC SQL SET ISOLATION TO REPEATABLE READ;
EXEC SQL INSERT ... ;
EXEC SQL COMMIT WORK;-- Executes without error
```

Compare the previous example to these SET TRANSACTION statements:

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
   -- Produces error 876: Cannot issue SET TRANSACTION
   -- in an active transaction.
```

Another difference between SET ISOLATION and SET TRANSACTION is the duration of isolation levels. The isolation level set by SET ISOLATION remains in effect until another SET ISOLATION statement is issued. The isolation level set by SET TRANSACTION only remains in effect until the transaction terminates. Then the isolation level is reset to the default for the database type.

Informix Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Using the Read Uncommitted Option

Use the Read Uncommitted option to copy rows from the database whether or not locks are present on them. The program that fetches a row places no locks and it respects none. Read Uncommitted is the only isolation level available to databases that do not have transactions.

This isolation level is most appropriate for static tables that are used for queries, that is, tables where data is not being modified, since it provides no isolation. With Read Uncommitted, the program might return a phantom row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row.

Using the Read Committed Option

Use the Read Committed option to guarantee that every retrieved row is committed in the table at the time that the row is retrieved. This option does not place a lock on the fetched row. Read Committed is the default level of isolation in a database with logging that is not ANSI compliant.

Read Committed is appropriate when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

Using the Repeatable Read and Serializable Options

The Informix implementation of Repeatable Read and of Serializable are equivalent. The Serializable (or Repeatable Read) option places a shared lock on every row that is selected during the transaction.

Another process can also place a shared lock on a selected row, but no other process can modify any selected row during your transaction or insert a row that meets the search criteria of your query during your transaction.

If you repeat the query during the transaction, you reread the same data. The shared locks are released only when the transaction commits or rolls back. Serializable is the default isolation level in an ANSI-compliant database. Serializable isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most.

Default Isolation Levels

The default isolation level is established when you create the database.

Informix Name	ANSI Name	When This Is the Default Level of Isolation
Dirty Read	Read Uncommitted	Database without transaction logging
Committed Read	Read Committed	Databases with logging that are not ANSI- compliant
Repeatable Read	Serializable	ANSI-compliant databases

The default isolation level remains in effect until you issue a SET TRANSACTION statement within a transaction. After a COMMIT WORK statement completes the transaction or a ROLLBACK WORK statement cancels the transaction, the isolation level is reset to the default.

Access Modes

Access modes affect read and write concurrency for rows within transactions. Use access modes to control data modification. SET TRANSACTION can specify that a transaction is read-only or read-write. By default, transactions are read-write. When you specify a read-only transaction, certain limitations apply. Read-only transactions cannot perform the following actions:

- Insert, delete, or update rows of a table
- Create, alter, or drop any database object such as schemas, tables, temporary tables, indexes, or SPL routines
- Grant or revoke privileges

- Update statistics
- Rename columns or tables

You can execute SPL routines in a read-only transaction as long as the SPL routine does not try to perform any restricted statement.

Effects of Isolation Levels

You cannot set the database isolation level in a database that does not have logging. Every retrieval in such a database occurs as a Read Uncommitted.

The data that is obtained during retrieval of BYTE or TEXT data can vary, depending on the database isolation levels. Under Read Uncommitted or Read Committed isolation levels, a process is permitted to read a BYTE or TEXT column that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted BYTE or TEXT column when certain conditions exist. For information about these conditions, see the *Administrator's Guide*.

E/C

In ESQL/C, if you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to Serializable or by locking the entire table. A scroll cursor with hold, however, cannot guarantee the same consistency between the two tables. A table-level lock or locks set by Serializable are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, so the retrieved data in the temporary table might be inconsistent with the actual data. ♦

Warning: *Do not use nonlogging tables within a transaction. If you need to use a nonlogging table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.*

Related Information

Related statements: CREATE DATABASE, SET ISOLATION, and SET LOCK MODE

For a discussion of isolation levels and concurrency issues, see the *IBM Informix Guide to SQL: Tutorial*.

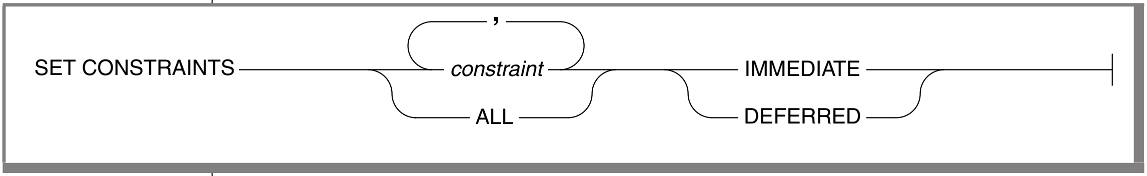
SET Transaction Mode

Use the SET Transaction Mode statement to specify whether constraints are checked at the statement level or at the transaction level.

IDS

To enable or disable constraints, or to change their filtering mode, see "SET Database Object Mode" on page 2-652. ♦

Syntax



Element	Purpose	Restrictions	Syntax
constraint	A constraint whose transaction mode is to be changed	All constraints must exist in the same database, which must support logging.	Database Object Name, p. 4-46

Usage

The SET Transaction Mode statement is valid only in a database with logging. The effect of the SET Transaction Mode statement is limited to the transaction in which it is executed. Use the IMMEDIATE keyword to set the transaction mode of constraints to statement-level checking. IMMEDIATE is the default transaction mode of constraints.

Use the DEFERRED keyword to set the transaction mode to transaction-level checking. You cannot change the transaction mode of a constraint to DEFERRED mode unless the constraint is currently enabled.

Statement-Level Checking

When you set the transaction mode to IMMEDIATE, statement-level checking is turned on, and all specified constraints are checked at the end of each INSERT, UPDATE, or DELETE statement. If a constraint violation occurs, the statement is not executed.

Transaction-Level Checking

When you set the transaction mode of constraints to DEFERRED, statement-level checking is turned off, and all (or the specified) constraints are not checked until the transaction is committed. If a constraint violation occurs while the transaction is being committed, the transaction is rolled back.

Tip: *If you defer checking a primary-key constraint, checking the not-null constraint for that column or set of columns is also deferred.*

Duration of Transaction Modes

The duration of the transaction mode that the SET Transaction Mode statement specifies is the transaction in which the SET Transaction Mode statement is executed. You cannot execute this statement outside a transaction. Once a COMMIT WORK or ROLLBACK WORK statement is successfully completed, the transaction mode of all constraints reverts to IMMEDIATE.

To switch from transaction-level checking to statement-level checking, you can use the SET Transaction Mode statement to set the transaction mode to IMMEDIATE, or you can use a COMMIT WORK or ROLLBACK WORK statement to terminate your transaction.

Specifying All Constraints or a List of Constraints

You can specify all constraints in the database in your SET Transaction Mode statement, or you can specify a single constraint or list of constraints.

Specifying All Constraints

If you specify the ALL keyword, the SET Transaction Mode statement sets the transaction mode for all constraints in the database. If any statement in the transaction requires that any constraint on any table in the database be checked, the database server performs the checks at the statement level or the transaction level, depending on the setting that you specify in the SET Transaction Mode statement.

Specifying a List of Constraints

If you specify a single constraint name or a list of constraints, the SET Transaction Mode statement sets the transaction mode for the specified constraints only. If any statement in the transaction requires checking of a constraint that you did not specify in the SET Transaction Mode statement, that constraint is checked at the statement level regardless of the setting that you specified in the SET Transaction Mode statement for other constraints.

When you specify a list of constraints, the constraints do not have to be defined on the same table, but they must exist in the same database.

Specifying Remote Constraints

You can set the transaction mode of local constraints or remote constraints. That is, the constraints that are specified in the SET Transaction Mode statement can be constraints that are defined on local tables or constraints that are defined on remote tables.

Examples of Setting the Transaction Mode for Constraints

The following example shows how to defer checking constraints within a transaction until the transaction is complete. The SET Transaction Mode statement in the example specifies that any constraints on any tables in the database are not checked until the COMMIT WORK statement is encountered.

```
BEGIN WORK
SET CONSTRAINTS ALL DEFERRED
...
COMMIT WORK
```

The following example specifies that a list of constraints is not checked until the transaction is complete:

```
BEGIN WORK
SET CONSTRAINTS update_const, insert_const DEFERRED
...
COMMIT WORK
```

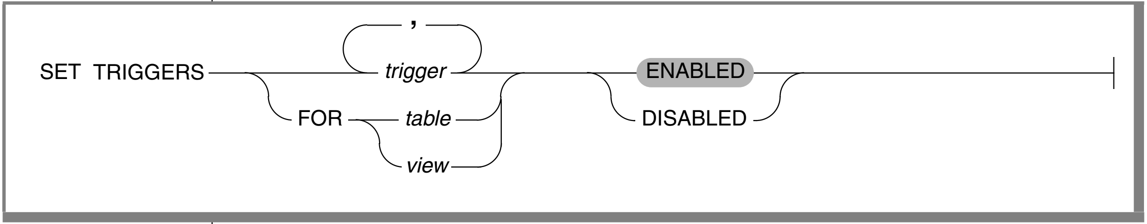
Related Information

Related statements: ALTER TABLE and CREATE TABLE

SET TRIGGERS

Use the SET TRIGGERS statement to enable or disable all or some of the triggers on a table, or all or some of the INSTEAD OF triggers on a view.

Syntax



Element	Purpose	Restrictions	Syntax
table	Table whose triggers are all to be enabled or disabled	Must exist	Database Object Name, p. 4-46
trigger	Trigger to be enabled or disabled	Must exist	Database Object Name, p. 4-46
view	View whose INSTEAD OF triggers are all to be enabled or disabled	Must exist	Database Object Name, p. 4-46

Usage

The SET TRIGGERS statement is a special case of the SET Database Object Mode statement. The SET Database Object Mode statement can also enable or disable an index or a constraint, or change the filtering mode of a unique index or of a constraint.

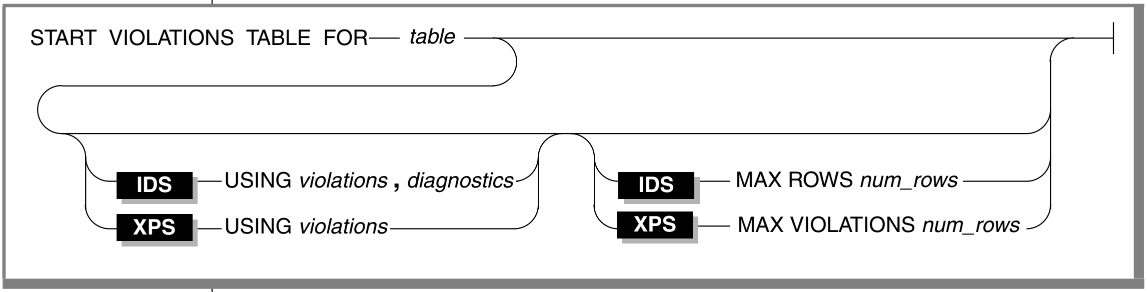
For the complete syntax and semantics of the SET TRIGGERS statement, see "SET Database Object Mode" on page 2-652.

+

START VIOLATIONS TABLE

Use the START VIOLATIONS TABLE statement to create a violations table and (for Dynamic Server only) a diagnostics table for a specified target table.

Syntax



Element	Purpose	Restrictions	Syntax
diagnostics	Diagnostics table to be associated with the target *table*. Default name is ***table*_dia**.	Must be unique among tables, views, and synonyms.	Database Object Name, p. 4-46
num_rows	Maximum number of rows the database server (IDS) or any single coserver (XPS) can insert into *violations* when a single statement is executed on *table*	Must be an integer in range from 1 to the maximum value of the INTEGER data type.	Literal Number, p. 4-216
table	Target table for which *violations* and (for IDS only) *diagnostics* are to be created	No more than 124 bytes (IDS) or no more than 14 bytes (XPS) if USING clause omitted	Database Object Name, p. 4-46
violations	Violations table to be associated with *table*. Default name is ***table*_vio**.	Same restrictions as *diagnostics*.	Database Object Name, p. 4-46

Usage

The database server associates the *violations* table and (for Dynamic Server only) the *diagnostics* table with the target *table* that you specify after the FOR keyword by recording the relationship among the three tables in the **sysviolations** system catalog table.

XPS

In Extended Parallel Server, the START VIOLATIONS TABLE statement creates a violations table, but no diagnostics table is created. ♦

A target table must satisfy these requirements:

- It cannot be external to the database.
- It cannot already be associated with a violations or diagnostics table.
- It cannot be a system catalog table.

The START VIOLATIONS TABLE statement creates the special violations table that holds rows that fail to satisfy constraints and unique indexes during INSERT, UPDATE, and DELETE operations on target tables. This statement also creates the special diagnostics table that contains information about the integrity violations that each row causes in the violations table.

Relationship to SET Database Object Mode Statement

The START VIOLATIONS TABLE statement is closely related to the SET Database Object Mode statement. If you use SET Database Object Mode to set the constraints or unique indexes defined on a table to the FILTERING mode, without also using START VIOLATIONS TABLE, any rows that violate a constraint or unique-index requirement in data manipulation operations are not filtered out to a violations table. Instead you receive an error message that indicates that you must start a violations table for the target table.

Similarly, if you use the SET Database Object Mode statement to set a disabled constraint or disabled unique index to the ENABLED or FILTERING mode, but you do not use START VIOLATIONS TABLE for the table on which the database objects are defined, any rows that do not satisfy the constraint or unique-index requirement are not filtered out to a violations table.

In these cases, to identify the rows that do not satisfy the constraint or unique-index requirement, issue the START VIOLATIONS TABLE statement to start the violations and diagnostics tables. Do this before you use the SET Database Object Mode statement to set the database objects to the ENABLED or FILTERING database object mode.

XPS

Extended Parallel Server does not support the SET Database Object Mode, and the concept of database object modes does not exist. Once you use the START VIOLATIONS TABLE statement to create a violations table and associate it with a target table, the existence of this violations table causes all violations of constraints and unique-index requirements by insert, delete, and update operations to be recorded in the violations table.

In other words, once you issue a START VIOLATIONS TABLE statement, all constraints and unique indexes in a database on Extended Parallel Server behave like filtering-mode constraints and filtering-mode unique indexes in a database on Dynamic Server. For an explanation of the behavior of filtering-mode constraints and filtering-mode unique indexes, see "Filtering Mode" on page 2-656. ♦

Effect on Concurrent Transactions

If athe database has transaction logging, you must issue START VIOLATIONS TABLE in isolation. That is, no other transaction can be in progress on a target table when you issues START VIOLATIONS TABLE on that table within a transaction. Any transactions that start on the target table after the first transaction has issued the START VIOLATIONS TABLE statement will behave the same way as the first transaction with respect to the violations and diagnostics tables. That is, any constraint and unique-index violations by these subsequent transactions will be recorded in the violations and diagnostics tables.

For example, if transaction A operates on table **tab1** and issues a START VIOLATIONS TABLE statement on table **tab1**, the database server starts a violations table named **tab1_vio** and filters any constraint or unique-index violations on table **tab1** by transaction A to table **tab1_vio**. If transactions B and C start on table **tab1** after transaction A has issued the START VIOLATIONS TABLE statement, the database server also filters any constraint and unique-index violations by transactions B and C to table **tab1_vio**.

The result is that all three transactions do not receive error messages about constraint and unique-index violations, even though transactions B and C do not expect this behavior. For example, if transaction B issues an INSERT or UPDATE statement that violates a check constraint on table **tab1**, the database server does not issue a constraint violation error to transaction B. Instead, the database server filters the bad row to the violations table without notifying transaction B that a data-integrity violation occurred.

IDS

You can prevent this situation from arising in Dynamic Server by specifying WITH ERRORS when you specify the FILTERING mode in a SET Database Object Mode, CREATE TABLE, ALTER TABLE, or CREATE INDEX statement. When multiple transactions operate on a table and the WITH ERRORS option is in effect, any transaction that violates a constraint or unique-index requirement on a target table receives a data-integrity error message. ♦

IDS

In Extended Parallel Server, once a transaction issues a START VIOLATIONS TABLE statement, you have no way to make the database server issue data-integrity violation messages to that transaction or to any other transactions that start subsequently on the same target table. ♦

Stopping the Violations and Diagnostics Tables

After you use START VIOLATIONS TABLE to create an association between a target table and the violations and diagnostics tables, the only way to drop the association between the target table and the violations and diagnostics tables is to issue a STOP VIOLATIONS TABLE statement for the target table. For more information, see "STOP VIOLATIONS TABLE" on page 2-748.

USING Clause

IDS

Use the USING clause to assign explicit names to the violations and diagnostics tables.

If you omit the USING clause, the database server assigns names to the violations and diagnostics tables. The system-assigned name of the violations table consists of the name of the target table followed by the string `vio`. The system-assigned name of the diagnostics table consists of the name of the target table followed by the string `_dia`.

If you omit the USING clause, the maximum length of the name of the target table is 124 bytes. ♦

XPS

Use the USING clause to assign an explicit name to the violations table.

If you omit the USING clause, the database server assigns a name to the violations table. The system-assigned name of the violations table consists of the name of the target table followed by the string `_vio`. If you omit the USING clause, the maximum length of the name of the target table is 14 bytes. ♦

IDS

Using the MAX ROWS Clause

The MAX ROWS clause specifies the maximum number of rows that the database server can insert into the diagnostics table when a single statement is executed on the target table. If you omit the MAX ROWS clause, no upper limit is imposed on the number of rows that can be inserted into the diagnostics table when a single statement is executed on the target table.

XPS

Using the MAX VIOLATIONS Clause

Use the MAX VIOLATIONS clause to specify the maximum number of rows that any single coserver can insert into the violations table when a single statement is executed on the target table. Each coserver where the violations table resides has this limit. The first coserver to reach this limit raises an error and causes the statement to fail.

If you omit the MAX VIOLATIONS clause of START VIOLATIONS TABLE, no upper limit exists on the number of rows that can be inserted into the violations table when a single statement is executed on the target table.

IDS

Specifying the Maximum Number of Rows in the Diagnostics Table

The following statement starts violations and diagnostics tables for the target table named **orders**. The MAX ROWS clause specifies the maximum number of rows that can be inserted into the diagnostics table when a single statement, such as an INSERT statement, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX ROWS 50000
```

XPS

Specifying the Maximum Number of Rows in the Violations Table

The following statement starts a violations table for the target table named **orders**. The MAX VIOLATIONS clause specifies the maximum number of rows that any single coserver can insert into the violations table when a single statement, such as an INSERT statement, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX VIOLATIONS 50000
```

Privileges Required for Starting Violations Tables

To start a violations and diagnostics table for a target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and have the Resource privilege on the database.
- You must have the Alter privilege on the target table and the Resource privilege on the database.

Structure of the Violations Table

When you issue START VIOLATIONS TABLE for a target table, the violations table that the statement creates has a predefined structure. This structure consists of the columns of the target table and three additional columns.

Serial columns in the target table are converted to integer data types in the violations table.

Users can examine these bad rows in the violations table, analyze the related rows that contain diagnostics information in the diagnostics table, and take corrective actions.

The following table shows the structure of the violations table.

Column Name	Data Type	Column Description
Same columns (in the same order) that appear in the target table	Same types as corresponding columns in the target table.	The violations table has the same schema as the target table, so that rows violating constraints or a unique-index during insert, update, and delete operations can be filtered to the violations table.
informix_tupleid	SERIAL	Unique serial code for the nonconforming row
informix_optype	CHAR(1)	The type of operation that caused this bad row. This column can have the following values: `I` = Insert `D` = Delete `O` = Update (with original values in this row) `N` = Update (with new values in this row) `S` = SET Database Object Mode statement (IDS)
informix_recowner	CHAR(8)	User who issued the statement that created this bad row

Examples of START VIOLATIONS TABLE Statements

The following examples show different ways to execute the START VIOLATIONS TABLE statement.

Violations and Diagnostics Tables with Default Names

The following statement starts violations and diagnostics tables for the target table named **cust_subset**. The violations table is named **cust_subset_vio** by default, and the diagnostics table is named **cust_subset_dia** by default.

```
START VIOLATIONS TABLE FOR cust_subset
```

Violations and Diagnostics Tables with Explicit Names

The following statement starts a violations and diagnostics table for the target table named **items**. The USING clause assigns explicit names to the violations and diagnostics tables. The violations table is to be named **exceptions**, and the diagnostics table is to be named **reasons**.

```
START VIOLATIONS TABLE FOR items USING exceptions, reasons
```

Relationships Among the Target, Violations, and Diagnostics Tables

Users can take advantage of the relationships among the target, violations, and diagnostics tables to obtain diagnostic information about rows that cause data-integrity violations during INSERT, DELETE, and UPDATE statements. Each row of the violations table has at least one corresponding row in the diagnostics table.

- One row in the violations table is a copy of any row in the target table for which a data-integrity violation was detected. A row in the diagnostics table contains information about the nature of the data-integrity violation caused by the bad row in the violations table.
- One row in the violations table has a unique serial identifier in the **informix_tupleid** column. A row in the diagnostics table has the same serial identifier in its **informix_tupleid** column.

A given row in the violations table can have more than one corresponding row in the diagnostics table. The multiple rows in the diagnostics table all have the same serial identifier in their **informix_tupleid** column so that they are all linked to the same row in the violations table. Multiple rows can exist in the diagnostics table for the same row in the violations table because a bad row in the violations table can cause more than one data-integrity violation.

For example, a bad row can violate a unique-index requirement for one column, a not-NULL constraint for another column, and a check constraint for yet another column. In this case, the diagnostics table contains three rows for the single bad row in the violations table. Each of these diagnostic rows identifies a different data-integrity violation that the nonconforming row in the violations table caused.

By joining the violations and diagnostics tables, the DBA or target-table owner can obtain diagnostic information about any or all bad rows in the violations table. SELECT statements can perform these joins interactively, or you can write a program to perform them within transactions.

Initial Privileges on the Violations Table

When you issue the START VIOLATIONS TABLE statement to create the violations table, the database server uses the set of privileges granted on the target table as a basis for granting privileges on the violations table. The database server follows different rules, however, when it grants each type of privilege.

The following table summarizes the circumstances under which the database server grants each type of privilege on the violations table.

Privilege	Condition for Granting the Privilege
Alter	Alter privilege is not granted on the violations table. (Users cannot alter violations tables.)
Index	User has Index privilege on the violations table if the user has the Index privilege on the target table. The user cannot create a globally detached index on the violations table even if the user has the Index privilege on the violations table (XPS).
Insert	User has the Insert privilege on the violations table if the user has the Insert, Delete, or Update privilege on any column of the target table.

(1 of 2)

Privilege	Condition for Granting the Privilege
Delete	User has the Delete privilege on the violations table if the user has the Insert, Delete, or Update privilege on any column of the target table.
Select	User has the Select privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Select privilege on any column of the target table. User has the Select privilege on any other column of the violations table if the user has the Select privilege on the same column in the target table.
Update	User has the Update privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Update privilege on any column of the target table. Even if the user has the Update privilege on the **informix_tupleid** column, however, the user cannot update this SERIAL column. User has the Update privilege on any other violations table column if the user has the Update privilege on the same column in the target table.
References	The References privilege is not granted on the violations table. (Users cannot add referential constraints to violations tables.)

(2 of 2)

The following rules apply to ownership of the violations table and privileges on the violations table:

- When the violations table is created, the owner of the target table becomes the owner of the violations table.
- The owner of the violations table automatically receives all table-level privileges on the violations table, including the Alter and References privileges. The database server, however, prevents the owner of the violations table from altering the violations table or adding a referential constraint to the violations table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the violations table.

- When you issue an INSERT, DELETE, or UPDATE statement on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the INSERT, DELETE, or UPDATE statement.

  Similarly, when you issue a SET Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET Database Object Mode statement.
- The grantor of the initial set of privileges on the violations table is the same as the grantor of the privileges on the target table.

  For example, if user **henry** was granted the Insert privilege on the target table by both user **jill** and user **albert**, then the Insert privilege on the violations table is granted to **henry** both by **jill** and by **albert**.
- After the violations table is started, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the violations table from that user. Instead, you must explicitly revoke the privilege on the violations table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding fragment-level privileges on the violations table.

Example of Privileges on the Violations Table

The following example illustrates how the initial set of privileges on a violations table is derived from the current set of privileges on the target table. Assume that a table named **cust_subset** consists of the following columns: **ssn** (customer social security number), **fname** (customer first name), **lname** (customer last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.
- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags
```

The database server grants the following set of initial privileges on the **cust_subset_viols** violations table:

- User **alvin** is the owner of the violations table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, and Index privileges on the table.

  User **barbara** has the Select privilege on five columns of the violations table: the **ssn**, the **lname**, the **informix_tupleid**, the **informix_optype**, and the **informix_recowner** columns.
- User **carrie** has Insert and Delete privileges on the violations table.

  User **carrie** has the Update privilege on four columns of the violations table: the **city**, the **informix_tupleid**, the **informix_optype**, and the **informix_recowner** columns. She cannot, however, update the **informix_tupleid** column (because this is a SERIAL column).

  User **carrie** has the Select privilege on four columns of the violations table: the **ssn** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **danny** has no privileges on the violations table.

Using the Violations Table

The following rules concern the structure and use of the violations table:

- Every pair of update rows in the violations table has the same value in the **informix_tupleid** column to indicate that both rows refer to the same row in the target table.
- If the target table has columns named **informix_tupleid**, **informix_optype**, or **informix_recowner**, the database server attempts to generate alternative names for these columns in the violations table by appending a digit to the end of the column name (for example, **informix_tupleid1**). If this fails, an error is returned, and no violations table is started for the target table.
- When a table functions as a violations table, it cannot have triggers or constraints defined on it.
- When a table functions as a violations table, users can create indexes on the table, even though the existence of an index affects performance. Unique indexes on the violations table cannot be set to the filtering database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also. If the target table is dropped in the restricted mode, the DROP TABLE operation fails (because the violations and diagnostics tables exist).
- After a violations table is started for a target table, ALTER TABLE cannot add, modify, or drop columns of the violations, diagnostics, or target tables. Before you can alter any of these tables, you must issue a STOP VIOLATIONS TABLE statement for the target table.
- The database server does not clear out the contents of the violations table before or after it uses the violations table during an Insert, Update, Delete, or SET Database Object Mode operation.

- If a target table has a filtering-mode constraint or unique index defined on it and a violations table associated with it, users cannot insert into the target table by selecting from the violations table. Before you insert rows into the target table by selecting from the violations table, you must take one of the following steps:
  - ❑ You can set the constraint or unique index to DISABLED mode.
  - ❑ You can issue STOP VIOLATIONS TABLE for the target table.

  If it is inconvenient to take either of these steps, but you still want to copy records from the violations table into the target table, a third option is to select from the violations table into a temporary table and then insert the contents of the temporary table into the target table.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the violations table has the same fragmentation strategy as the target table. Each fragment of the violations table is stored in the same dbspace as the corresponding fragment of the target table.
- Once a violations table is started for a target table, you cannot use the ALTER FRAGMENT statement to alter the fragmentation strategy of the target table or the violations table.
- If the target table specified in the START VIOLATIONS TABLE statement is not fragmented, the database server places the violations table in the same dbspace as the target table.
- If the target table has BYTE or TEXT columns, BYTE or TEXT data in the violations table is created in the same blobspace as the BYTE or TEXT data in the target table.

Example of a Violations Table

To start a violations and diagnostics table for the target table named **customer** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer
```

Because you include no USING clause, the violations table is named **customer_vio** by default. The **customer_vio** table includes these columns:

customer_num
fname
lname
company
address1
address2
city
state
zipcode
phone
informix_tupleid
informix_optype
informix_recowner

The **customer_vio** table has the same table definition as the **customer** table except that the **customer_vio** table has three additional columns that contain information about the operation that caused the bad row.

IDS

Structure of the Diagnostics Table

When you issue a START VIOLATIONS TABLE statement for a target table, the diagnostics table that the statement creates has a predefined structure. This structure is independent of the structure of the target table.

The following table shows the structure of the diagnostics table.

Column Name	Type	Purpose
informix_tupleid	INTEGER	Implicitly refers to the values in the **informix_tupleid** column in the violations table This relationship, however, is not declared as a foreign-key to primary-key relationship.
objtype	CHAR(1)	Identifies the type of the violation This column can have the following values: `C` = Constraint violation `I` = Unique-index violation
objowner	CHAR(8)	Identifies the owner of the constraint or index for which an integrity violation was detected
objname	CHAR(18)	Contains the name of the constraint or index for which an integrity violation was detected

Initial Privileges on the Diagnostics Table

When the START VIOLATIONS TABLE statement creates the diagnostics table, the set of privileges granted on the target table are a basis for granting privileges on the diagnostics table. The database server follows different rules, however, when it grants each type of privilege.

The following table explains the circumstances under which the database server grants each privilege on the diagnostics table.

Privilege	Condition for Granting the Privilege
Insert	User has the Insert privilege on the diagnostics table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Delete	User has the Delete privilege on the diagnostics table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Select	User has the Select privilege on the diagnostics table if the user has the Select privilege on any column in the target table.
Update	User has the Update privilege on the diagnostics table if the user has the Update privilege on any column in the target table.
Index	User has the Index privilege on the diagnostics table if the user has the Index privilege on the target table.
Alter	Alter privilege is not granted on the diagnostics table. (Users cannot alter diagnostics tables.)
References	References privilege is not granted on the diagnostics table. (Users cannot add referential constraints to diagnostics tables.)

The following rules concern privileges on the diagnostics table:

- When the diagnostics table is created, the owner of the target table becomes the owner of the diagnostics table.
- The owner of the diagnostics table automatically receives all table-level privileges on the diagnostics table, including the Alter and References privileges. The database server, however, prevents the owner of the diagnostics table from altering the diagnostics table or adding a referential constraint to the diagnostics table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the diagnostics table.
- For INSERT, DELETE, or UPDATE operations on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the INSERT, DELETE, or UPDATE statement.

  Similarly, when you issue a SET Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET Database Object Mode statement.
- The grantor of the initial set of privileges on the diagnostics table is the same as the grantor of the privileges on the target table. For example, if the user **jenny** was granted the Insert privilege on the target table by both the user **wayne** and the user **laurie**, both user **wayne** and user **laurie** grant the Insert privilege on the diagnostics table to user **jenny**.

- Once a diagnostics table is started for a target table, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the diagnostics table from that user. Instead you must explicitly revoke the privilege on the diagnostics table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding table-level privileges on the diagnostics table.

The next example illustrates how the initial set of privileges on a diagnostics table is derived from the current privileges on the target table. Assume that you have a table called **cust_subset** that holds **customer data.** This table consists of the following columns: **ssn** (social security number), **fname** (first name), **lname** (last name), and **city** (city in which the customer lives). The following set of privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.
- User **danny** has the Alter privilege on the table.
- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags
```

The database server grants the following set of initial privileges on the **cust_subset_diags** diagnostics table:

- User **alvin** is the owner of the diagnostics table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, Select, and Index privileges on the diagnostics table.
- User **carrie** has the Insert, Delete, Select, and Update privileges on the diagnostics table.
- User **danny** has no privileges on the diagnostics table.

Using the Diagnostics Table

For information on the relationship between the diagnostics table and the violations table, see "Relationships Among the Target, Violations, and Diagnostics Tables" on page 2-735.

The following issues concern the structure and use of the diagnostics table:

- The MAX ROWS clause of the START VIOLATIONS TABLE statement sets a limit on the number of rows that can be inserted into the diagnostics table when you execute a single statement, such as an INSERT or SET Database Object Mode statement, on the target table.
- The MAX ROWS clause limits the number of rows only for operations in which the table functions as a diagnostics table.
- When a table functions as a diagnostics table, it cannot have triggers or constraints defined on it.
- When a table functions as a diagnostics table, users can create indexes on the table, even though the existence of an index affects performance. You cannot set unique indexes on the diagnostics table to the filtering database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in CASCADE mode (the default mode) causes the violations and diagnostics tables to be dropped also.
- If the target table is dropped in RESTRICTED mode, the DROP TABLE operation fails (because the violations and diagnostics tables exist).
- Once a violations table is started for a target table, you cannot use the ALTER TABLE statement to add, modify, or drop columns in the target table, violations table, or diagnostics table. Before you can alter any of these tables, you must issue a STOP TABLE VIOLATIONS statement for the target table.
- The database server does not clear out the contents of the diagnostics table before or after it uses the diagnostics table during an Insert, Update, Delete, or Set operation.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the diagnostics table is fragmented with a round-robin strategy over the same dbspaces in which the target table is fragmented.

To start a violations and diagnostics table for the target table named **stock** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR stock
```

Because your START VIOLATIONS TABLE statement does not include a USING clause, the diagnostics table is named **stock_dia** by default. The **stock_dia** table includes the following columns:

informix_tupleid
objtype
objowner
objname

This list of columns shows an important difference between the diagnostics table and violations table for a target table. Whereas the violations table has a matching column for every column in the target table, the columns of the diagnostics table do not match any columns in the target table. The diagnostics table created by any START VIOLATIONS TABLE statement always has the same columns with the same column names and data types.

Related Information

Related statements: SET Database Object Mode and STOP VIOLATIONS TABLE

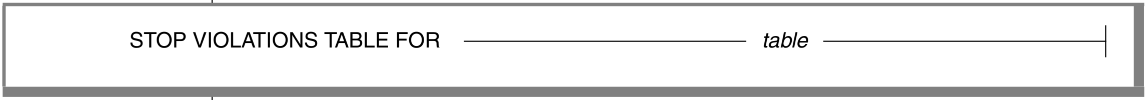
For a discussion of object modes and violation detection, see the *IBM Informix Guide to SQL: Tutorial*.

+

STOP VIOLATIONS TABLE

Use the STOP VIOLATIONS TABLE statement to drop the association between a target table and its violations table (and for IDS only, its diagnostics tables).

Syntax



Element	Purpose	Restrictions	Syntax
table	Name of target table whose association with the violations and diagnostics table is to be dropped. No default value exists.	Must be a local table that has an associated violations table and (for IDS only) a diagnostics table	Database Object Name, p. 4-46

Usage

The STOP VIOLATIONS TABLE statement drops the association between the target table and the violations and diagnostics tables. After you issue this statement, the former violations and diagnostics tables continue to exist, but they no longer function as violations and diagnostics tables for the target table. They now have the status of regular database tables instead of violations and diagnostics tables for the target table. You must issue the DROP TABLE statement to drop these two tables explicitly.

When INSERT, DELETE, or UPDATE operations cause data-integrity violations for rows of the target table, the nonconforming rows are no longer filtered to the former violations table, and diagnostics information about the data-integrity violations is not placed in the former diagnostics table.

XPS

In Extended Parallel Server, the diagnostics table does not exist. The STOP VIOLATIONS TABLE statement drops the association between the target table and the violations table. ♦

Example of Stopping a Violations and Diagnostics Table

Assume that a target table named **cust_subset** has an associated violations table named **cust_subset_vio** and an associated diagnostics table named **cust_subset_dia**. To drop the association between the target table and the violations and diagnostics tables, enter the following statement:

```
STOP VIOLATIONS TABLE FOR cust_subset
```

Example of Dropping a Violations and Diagnostics Table

After you execute the STOP VIOLATIONS TABLE statement in the preceding example, the **cust_subset_vio** and **cust_subset_dia** tables continue to exist, but they are no longer associated with the **cust_subset** table. Instead they now have the status of regular database tables. To drop these two tables, enter the following statements:

```
DROP TABLE cust_subset_vio;
DROP TABLE cust_subset_dia;
```

Privileges Required for Stopping a Violations Table

To stop a violations and diagnostics table for a target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and have the Resource privilege on the database.
- You must have the Alter privilege on the target table and the Resource privilege on the database.

Related Information

Related statements: SET Database Object Mode and START VIOLATIONS TABLE

For a discussion of database object modes and violation detection, see the *IBM Informix Guide to SQL: Tutorial*.

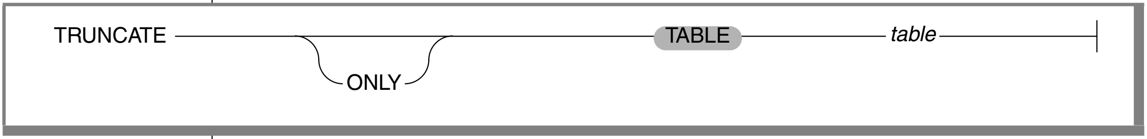
+

XPS

TRUNCATE

Use the TRUNCATE statement for quick removal of all rows from a table and all corresponding index data.

Syntax



Element	Purpose	Restrictions	Syntax
table	Name of table from which to remove all data	Must exist	Database Object Name, p. 4-46

Usage

You must be the owner of the table or have the DBA privilege to use this statement.

The TRUNCATE statement does not automatically reset the serial value of a column. To reset the serial value of a column, you must do so explicitly, either before or after you run the TRUNCATE statement.

TRUNCATE is not equivalent to DROP TABLE. After TRUNCATE successfully executes, the specified *table* (and all its columns, synonyms, views, indexes and permissions) still exists in the database schema, but with no rows of data.

Restrictions

The statement will not succeed if any of the following conditions exist:

- One or more cursors are open on the table.
- Referential constraints exist on the table and any of the referencing tables has at least one row.
- A shared or exclusive lock on the table already exists.
- The statement references a view.

- The statement references any of the following types of tables:
  - ❑ External
  - ❑ System catalog
  - ❑ Violations
- The statement is issued inside a transaction.

Using the ONLY and TABLE Keyword

The ONLY and TABLE keywords have no effect on this statement. They can be included to make your code more legible for human readers.

All of the following statements have the same effect, deleting all rows and any related index data from the **customer** table:

```
TRUNCATE ONLY TABLE customer

TRUNCATE TABLE customer

TRUNCATE ONLY customer

TRUNCATE customer
```

After the TRUNCATE Statement Executes

Information about the success of this statement appears in the logical-log files. For information about logical-log files, see your *Administrator's Guide*.

If the table was fragmented, after the statement executes, each fragment has a space allocated for it that is the same size as that of the first extent size. The fragment size of any indexes also corresponds to the size of the first extents.

Because the TRUNCATE statement does not alter the schema, the database server does not automatically update statistics. After you use this statement, you might want to issue an UPDATE STATISTICS statement.

When You Might Use The TRUNCATE Statement

TRUNCATE performs similar operations to those that you can perform with the DELETE statement or a combination of DROP TABLE and CREATE TABLE.

Using TRUNCATE can be faster than removing all rows from a table with the DELETE statement, because it does not activate any DELETE triggers. In addition, when you use this statement, the database server creates a log entry for the entire TRUNCATE statement rather than for each deleted row.

You might also use this statement instead of dropping a table and then re-creating it. When you drop and re-create a table, you must regrant any privileges that you want to preserve on the table. In addition, you must re-create any indexes, constraints, and triggers that were defined on the table. The TRUNCATE statement leaves these database objects and privileges intact.

Related Information

Related statements: DELETE and DROP TABLE

For more information about the performance implications of this statement, see your *Performance Guide*.

+

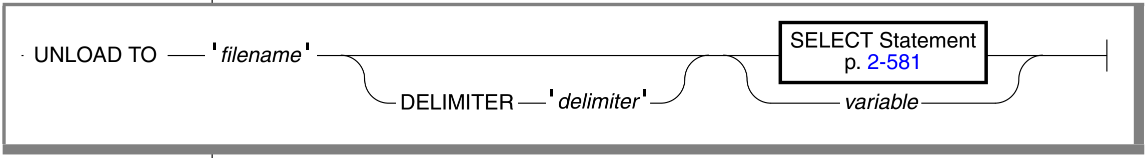
DB

SQLE

UNLOAD

Use the UNLOAD statement to write the rows retrieved in a SELECT statement to an operating-system file. Use this with DB-Access.

Syntax



Element	Purpose	Restrictions	Syntax
delimiter	Quoted string to specify the field delimiter character in *filename* file	See "DELIMITER Clause" on page 2-758	Quoted String, p. 4-243
filename	Operating-system file to receive the rows. Default pathname is the current directory.	See "UNLOAD TO File" on page 2-754.	Quoted String, p. 4-243
variable	Host variable that contains the text of a valid SELECT statement	Must have been declared as a character data type.	Language-specific

Usage

UNLOAD copies to a file the rows that a query retrieves. You must have the Select privilege on all columns specified in the SELECT statement. For information on database- and table-level privileges, see "GRANT" on page 2-459.

You can specify a literal SELECT statement or a character variable that contains the text of a SELECT statement. (See "SELECT" on page 2-581.)

This example unloads rows where the value of **customer.customer_num** is greater than or equal to 138, and writes them to a file named **cust_file**:

```
UNLOAD TO 'cust_file' DELIMITER '!'
   SELECT * FROM customer WHERE customer_num> = 138
```

The resulting output file, **cust_file**, contains two rows of data values:

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite 10!Palo
Alto!CA!94306!!
139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo
Alto!CA!94301!(415)323-5400
```

UNLOAD TO File

The UNLOAD TO file, as specified by the *filename* parameter, receives the retrieved rows. You can use an UNLOAD TO file as input to a LOAD statement.

In the default locale, data values have these formats in the UNLOAD TO file.

Data Type	Output Format
BOOLEAN	IDS BOOLEAN values appear as either `t` for TRUE or `f` for FALSE. ♦
Character	If a character field contains the delimiter, IBM Informix products automatically escape it with a backslash (\) to prevent interpretation as a special character. (If you use a LOAD statement to insert the rows into a table, backslashes are automatically stripped.)
Collections	A collection is unloaded with its values enclosed between braces ({ }) and a field delimiter separating each element. For more information, see "Unloading Complex Types" on page 2-758.
DATE	DATE values are represented as *mm*/*dd*/*yyyy* (or the default format for the database locale), where *mm* is the month (January = `1`, and so on), *dd* is the day, and *yyyy* is the year. If you have set the **GL_DATE** or **DBDATE** environment variable, the UNLOAD statement uses the specified date format for DATE values.
DATETIME, INTERVAL	Literal DATETIME and INTERVAL values appear as digits and delimiters, without keyword qualifiers, in the default format *yyyy-mm-dd hh:mi:ss.fff*. Time units outside the declared precision are omitted. If the **GL_DATETIME** or **DBTIME** environment variable is set, DATETIME values appear in the specified format.
DECIMAL, MONEY	Values are unloaded with no leading currency symbol. In the default locale, comma (,) is the *thousands* separator and period (.) is the decimal separator. If **DBMONEY** is set, UNLOAD uses its specified separators (and its currency format for MONEY values).
NULL	NULL appears as two delimiters with no characters between them.
Number	Values appear as literals, with no leading blanks. INTEGER, INT8, or SMALLINT zero appear as `0`, and MONEY, FLOAT, SMALLFLOAT, or DECIMAL zero is represented as `0.0`.
ROW types (named and unnamed)	GLID A ROW type is unloaded with its values surrounded by parentheses and a field delimiter separating each element. For more information, see "Unloading Complex Types" on page 2-758. ♦

(1 of 2)

IDS

Data Type	Output Format
Simple large objects (TEXT, BYTE)	TEXT and BYTE columns are unloaded directly into the UNLOAD TO file. BYTE values appear in ASCII hexadecimal form, with no added whitespace or newline characters. For more information, see "Unloading Simple Large Objects" on page 2-756.
Smart large objects (CLOB, BLOB)	CLOB and BLOB columns are unloaded into a separate operating-system file on the client computer. The CLOB or BLOB field in the UNLOAD TO file contains the name of this separate file. For more information, see "Unloading Smart Large Objects" on page 2-756.
User-defined data types (opaque types)	Opaque types must have an **export** support function defined. They need special processing to copy data from the internal format of the opaque type to the UNLOAD TO file format. An export binary support function might also be required for data in binary format. The data in the UNLOAD TO file would correspond to the format that the **export** or **exportbinary** support function returns. ♦

(2 of 2)

For more information on **DB** environment variables, refer to the *IBM Informix Guide to SQL: Reference*. For more information on **GL** environment variables, refer to the *IBM Informix GLS User's Guide*.

GLS

In a nondefault locale, DATE, DATETIME, MONEY, and numeric column values have formats that the locale supports for these data types. For more information, see the *IBM Informix GLS User's Guide*. ♦

Unloading Character Columns

In unloading files that contain VARCHAR or NVARCHAR columns, trailing blanks are retained in VARCHAR or NVARCHAR fields. Trailing blanks are discarded when CHAR or NCHAR columns are unloaded.

For CHAR, VARCHAR, NCHAR, and NVARCHAR columns, an empty string (that is, a data string of zero length, containing no characters) is represented in the UNLOAD TO file as the four bytes "|\ |" (delimiter, backslash, blank space, delimiter).

Some earlier releases of Informix database servers used "||" (consecutive delimiters) to represent the empty string in LOAD and UNLOAD operations. In this release, however, "||" only represents NULL values in CHAR, VARCHAR, NCHAR, and NVARCHAR columns.

Unloading Simple Large Objects

The database server writes BYTE and TEXT values directly into the UNLOAD TO file. BYTE values are written in hexadecimal dump format with no added blank spaces or new line characters. The logical length of an UNLOAD TO file containing BYTE data can therefore be long and difficult to print or edit.

If you are unloading files that contain simple-large-object data types, do not use characters that can appear in BYTE or TEXT values as delimiters in the UNLOAD TO file. See also the section "DELIMITER Clause" on page 2-758.

GLS

The database server handles any required code-set conversions for TEXT data. For more information, see the *IBM Informix GLS User's Guide*. ♦

If you are unloading files that contain simple-large-object data types, objects smaller than 10 kilobytes are stored temporarily in memory. You can adjust the 10-kilobyte setting to a larger setting with the **DBBLOBBUF** environment variable. Simple large objects that are larger than the default or the setting of the **DBBLOBBUF** environment variable are stored in a temporary file. For additional information about the **DBBLOBBUF** environment variable, see the *IBM Informix Guide to SQL: Reference*.

IDS

Unloading Smart Large Objects

The database server unloads smart large objects (BLOB and CLOB columns) into a separate operating-system file on the client computer, in the same directory as the UNLOAD TO file. The file has a name in one of these formats:

- For a BLOB value: `blob########`
- For a CLOB value: `clob########`

In the preceding formats, the pound (`#`) symbols represent the digits of the unique hexadecimal smart-large-object identifier. The database server uses the hexadecimal ID for the first smart large object in the file. The maximum number of digits for a smart-large-object identifier is 17. Most smart large objects, however, would have an identifier with fewer digits.

When the database server unloads the first smart large object, it creates the appropriate BLOB or CLOB client file with the hexadecimal identifier of the smart large object. If additional smart-large-object values are present, the database server creates another BLOB or CLOB client file whose filename contains the hexadecimal identifier of the next smart large object to unload.

In an UNLOAD TO file, a BLOB or CLOB column value appears as follows:

```
start_off,length,client_path
```

In this format, *start_off* is the starting offset (in hexadecimal) of the smart-large-object value within the client file, *length* is the length (in hexadecimal) of the BLOB or CLOB value, and *client_path* is the pathname for the client file. No blank spaces can appear between these values. If a CLOB value is 512 bytes long and is at offset 256 in the **/usr/apps/clob9ce7.318** file, for example, then the CLOB value appears as follows in the UNLOAD TO file:

```
|100,200,/usr/apps/clob9ce7.318|
```

If a BLOB or CLOB column value occupies an entire client file, the CLOB or BLOB column value appears as follows in the UNLOAD TO file:

```
client_path
```

For example, if a CLOB value occupies the entire file **/usr/apps/clob9ce7.318**, the CLOB value appears as follows in the UNLOAD TO file:

```
|/usr/apps/clob9ce7.318|
```

GLS

For CLOB columns, the database server handles any required code-set conversions for the data. For more information, see the *IBM Informix GLS User's Guide*. ♦

IDS

Unloading Complex Types

In an UNLOAD TO file, complex types appear as follows:

- Collections are introduced with the appropriate constructor (SET, MULTISET, LIST), and have their elements enclosed in braces ({}) and separated with a comma, as follows:

  ```
  constructor{val1 , val2 , ... }
  ```

  For example, to unload the SET values {1, 3, 4} from a column of the SET (INTEGER NOT NULL) data type, the corresponding field of the UNLOAD TO file appears as follows:

  ```
  |SET{1 , 3 , 4}|
  ```

- Row types (named and unnamed) are introduced with the ROW constructor and have their fields enclosed between parentheses and separated with a comma, as follows:

  ```
  ROW(val1 , val2 , ... )
  ```

  For example, to unload the ROW values (`1, 'abc'`), the corresponding field of the UNLOAD TO file appears as follows:

  ```
  |ROW(1 , abc)|
  ```

DELIMITER Clause

Use the DELIMITER clause to identify the delimiter that separates the data contained in each column in a row in the output file. If you omit this clause, DB-Access checks the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the pipe (|).

You can specify the TAB (CTRL-I) or a blank space (ASCII 32) as the delimiter symbol. You cannot use the following symbols as delimiter symbols:

- Backslash (\ )
- Newline character (CTRL-J)
- Hexadecimal digits (`0` to `9`, `a` to `f`, `A` to `F`)

The backslash (\ ) is not a valid field separator or record delimiter because it serves as an escape character, indicating that the next character is a literal character in the data, rather than a special character.

The following statement specifies the semicolon (;) as the delimiter:

```
UNLOAD TO 'cust.out' DELIMITER ';'
   SELECT fname, lname, company, city
      FROM customer
```

Related Information

Related statements: LOAD and SELECT

For information about how to set the **DBDELIMITER** environment variable, see the *IBM Informix Guide to SQL: Reference*.

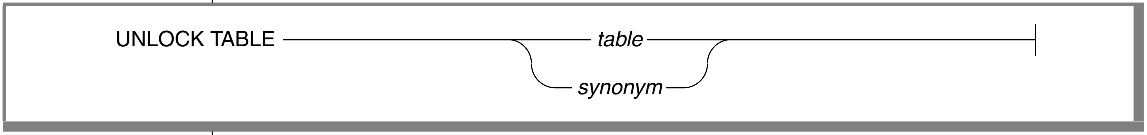
For a discussion of the GLS aspects of the UNLOAD statement, see the *IBM Informix GLS User's Guide*.

For a task-oriented discussion of the UNLOAD statement and other utilities for moving data, see the *IBM Informix Migration Guide*.

UNLOCK TABLE

Use the UNLOCK TABLE statement in a database without transactions to unlock a table that you previously locked with the LOCK TABLE statement. The UNLOCK TABLE statement fails in a database that uses transactions.

Syntax



Element	Purpose	Restrictions	Syntax
synonym	Synonym for a table to unlock	The synonym and the table to which it points must exist.	Database Object Name, p. 4-46
table	Table to unlock	Must be in a database without transactions and must be a table that you previously locked	Database Object Name, p. 4-46

Usage

You can lock a table if you own the table or if you have the Select privilege on the table, either from a direct grant to yourself or from a grant to **public**. You can only unlock a table that you locked. You cannot unlock a table that another process locked. Only one lock can apply to a table at a time.

You must specify the name or synonym of the table that you are unlocking. Do not specify the name of a view, or a synonym for a view.

To change the lock mode of a table in a database without transactions, use the UNLOCK TABLE statement to unlock the table, then issue a new LOCK TABLE statement. The following example shows how to change the lock mode of a table in a database that was created without transactions:

```
LOCK TABLE items IN EXCLUSIVE MODE
...
UNLOCK TABLE items
...
LOCK TABLE items IN SHARE MODE
```

The UNLOCK TABLE statement fails if it is issued within a transaction. Table locks set within a transaction are released automatically when the transaction completes.

ANSI

If you are using an ANSI-compliant database, do not issue an UNLOCK TABLE statement. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database. ♦

Related Information

Related statements: BEGIN WORK, COMMIT WORK, LOCK TABLE, and ROLLBACK WORK

For a discussion of concurrency and locks, see the *IBM Informix Guide to SQL: Tutorial*.

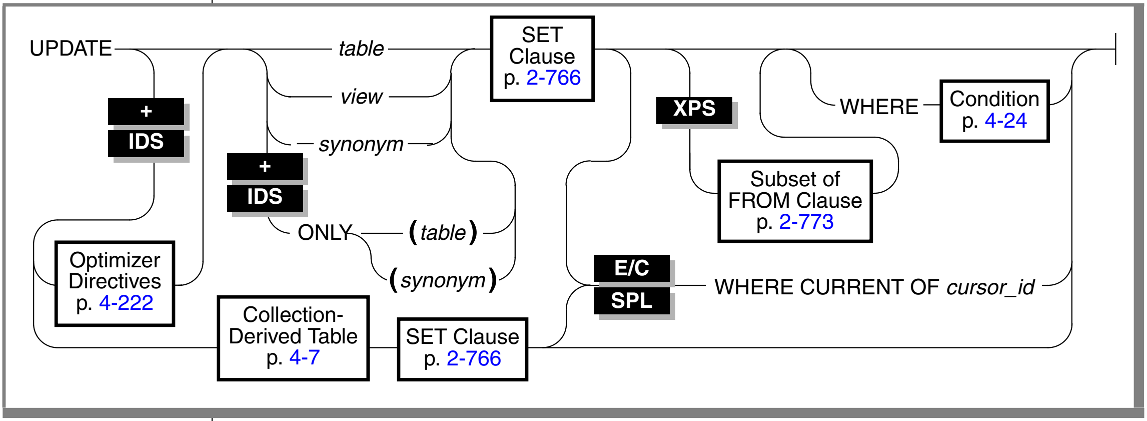
UPDATE

Use the UPDATE statement to change the values in one or more columns of one or more rows in a table or view.

IDS

With Dynamic Server, you can also use this statement to change the values in one or more elements in an ESQL/C collection variable. ♦

Syntax



Element	Purpose	Restrictions	Syntax
cursor_id	Name of a cursor whose current row is to be updated	You cannot update a row that includes aggregates.	Identifier, p. 4-189
synonym, table, view	Synonym, table, or view that contains the rows to be updated	The synonym and the table or view to which it points must exist.	Database Object Name, p. 4-46

Usage

Use the UPDATE statement to update any of the following types of objects:

- A row in a table: a single row, a group of rows, or all rows in a table
- An element in a collection variable
- An ESQL/C **row** variable: a field or all fields

IDS

For information on how to update elements of a collection variable, see "Collection-Derived Table" on page 4-7. The other sections of this UPDATE statement describe how to update a row in a table. ♦

You must either own the table or have the Update privilege for the table; see "GRANT" on page 2-459. To update data in a view, you must have the Update privilege, and the view must meet the requirements that are explained in "Updating Rows Through a View" on page 2-764.

The cursor (as defined in the SELECT...FOR UPDATE portion of a DECLARE statement) can contain only column names. If you omit the WHERE clause, all rows of the target table are updated.

If you are using effective checking and the checking mode is set to IMMEDIATE, all specified constraints are checked at the end of each UPDATE statement. If the checking mode is set to DEFERRED, all specified constraints are *not* checked until the transaction is committed.

XPS

In Extended Parallel Server, if UPDATE is constructed in such a way that a single row might be updated more than once, the database server returns an error. If the new value is the same in every update, however, the database server allows the UPDATE to take place without reporting an error. ♦

DB

If you omit the WHERE clause and are in interactive mode, DB-Access does not run the UPDATE statement until you confirm that you want to change all rows. If the statement is in a command file, however, and you are running at the command line, the statement executes immediately. ♦

IDS

Using the ONLY Keyword

If you use the UPDATE statement to update rows of a supertable, rows from its subtables can also be updated. To update rows from the supertable only, use the ONLY keyword prior to the table name, as this example shows:

```
UPDATE ONLY(am_studies_super)
WHERE advisor = "johnson"
SET advisor = "camarillo"
```

Warning: *If you use the UPDATE statement on a supertable without the ONLY keyword and without a WHERE clause, all rows of the supertable and its subtables are updated.*

You cannot use the ONLY keyword if you plan to use the WHERE CURRENT OF clause to update the current row of the active set of a cursor.

Updating Rows Through a View

You can update data through a *single-table* view if you have the Update privilege on the view (see "GRANT" on page 2-459). Certain restrictions exist. For a view to be updatable, the SELECT statement that defines the view must not contain any of the following items:

- Columns in the select list that are aggregate values
- Columns in the select list that use the UNIQUE or DISTINCT keyword
- A GROUP BY clause
- A UNION operator

In addition, if a view is built on a table that has a derived value for a column, that column cannot be updated through the view. Other columns in the view, however, can be updated. In an updatable view, you can update the values in the underlying table by inserting values into the view. See also the section "INSTEAD OF Triggers on Views" on page 2-305, which describes how you can create INSTEAD OF triggers to update tables through a view.

You can use data-integrity constraints to prevent users from updating values in the underlying table when the update values do not fit the SELECT statement that defined the view. For more information, see "WITH CHECK OPTION Keywords" on page 2-314.

Because duplicate rows can occur in a view even though the underlying table has unique rows, be careful when you update a table through a view. For example, if a view is defined on the **items** table and contains only the **order_num** and **total_price** columns, and if two items from the same order have the same total price, the view contains duplicate rows. In this case, if you update one of the two duplicate total price values, you have no way to know which item price is updated.

Important: *If you are using a view with a check option, you cannot update rows to a remote table.*

Updating Rows in a Database Without Transactions

If you are updating rows in a database without transactions, you must take explicit action to restore updated rows. For example, if the UPDATE statement fails after updating some rows, the successfully updated rows remain in the table. You cannot automatically recover from a failed update.

Updating Rows in a Database with Transactions

If you are updating rows in a database with transactions, and you are using transactions, you can undo the update using the ROLLBACK WORK statement. If you do not execute a BEGIN WORK statement before the update, and the update fails, the database server automatically rolls back any database modifications made since the beginning of the update.

You can create temporary tables with the WITH NO LOG option. These tables are never logged and are not recoverable.

Tables that you create with the RAW logging type are never logged. Thus, RAW tables are not recoverable, even if the database uses logging. For information about RAW tables, refer to the *IBM Informix Guide to SQL: Reference*.

ANSI

In an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an UPDATE statement fails, you can use ROLLBACK WORK to undo the update.

If you are within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

Locking Considerations

When a row is selected with the intent to update, the update process acquires an update lock. Update locks permit other processes to read, or *share*, a row that is about to be updated, but they do not allow those processes to update or delete it. Just before the update occurs, the update process *promotes* the shared lock to an exclusive lock. An exclusive lock prevents other processes from reading or modifying the contents of the row until the lock is released.

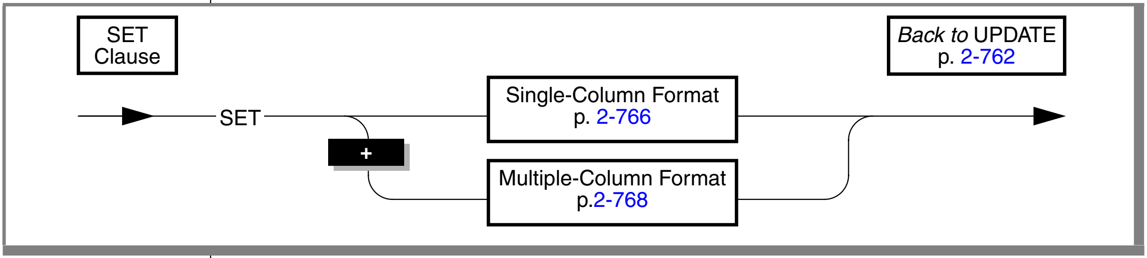
An update process can acquire an update lock on a row or on a page that has a shared lock from another process, but you cannot promote the update lock from shared to exclusive (and the update cannot occur) until the other process releases its lock.

If the number of rows that a single update affects is large, you can exceed the limits placed on the maximum number of simultaneous locks. If this occurs, you can reduce the number of transactions per UPDATE statement, or you can lock the page or the entire table before you execute the statement.

SET Clause

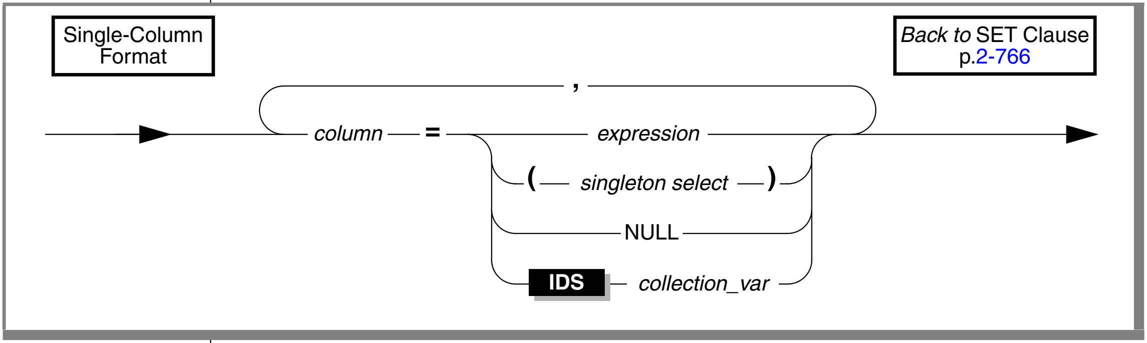
Use the SET clause to identify the columns to update and assign values to each column. The clause supports the following formats:

- A single-column format, which pairs a column to a single expression
- A multiple-column format, which lists multiple columns and sets them equal to corresponding expressions



Single-Column Format

Use the single-column format to pair one column with a single expression.



Element	Purpose	Restrictions	Syntax
column	Column to be updated	Cannot be a serial data type.	Identifier, p. 4-189
collection_var	Host or program variable	Must be declared as a collection data type.	Language specific
expression	Returns a value for *column*	Cannot contain aggregate functions.	Expression, p. 4-67
singleton select	Subquery that returns exactly one row	Returned subquery values must have a 1-to-1 correspondence with *column* list.	SELECT, p. 2-581

You can use this syntax to update a ROW column.

An expression list can include an SQL subquery that returns a single row of multiple values, provided that the number of columns named in the column list equals the number of values that the expression list returns.

You can include any number of single columns to single expressions in the UPDATE statement. For information on how to specify values of a row-type column in a SET clause, see "Updating ROW-Type Columns" on page 2-770. The following examples illustrate the single-column format of the SET clause.

```
UPDATE customer
   SET address1 = '1111 Alder Court', city = 'Palo Alto',
      zipcode = '94301' WHERE customer_num = 103;

UPDATE stock
   SET unit_price = unit_price * 1.07;
```

Using a Subquery to Update a Column

You can update a column with the value that a subquery returns.

```
UPDATE orders
   SET ship_charge =
      (SELECT SUM(total_price) * .07 FROM items
         WHERE orders.order_num = items.order_num)
      WHERE orders.order_num = 1001
```

IDS

If you are updating a supertable in a table hierarchy, the SET clause cannot include a subquery that references a subtable.

If you are updating a subtable in a table hierarchy, a subquery in the SET clause can reference the supertable if it references only the supertable. That is, the subquery must use the SELECT…FROM ONLY (*supertable*)… syntax. ♦

Updating a Column to NULL

Use the NULL keyword to modify a column value when you use the UPDATE statement. For example, for a customer whose previous address required two address lines but now requires only one, you would use the following entry:

```
UPDATE customer
   SET address1 = '123 New Street',
   SET address2 = null,
   city = 'Palo Alto',
   zipcode = '94303'
   WHERE customer_num = 134
```

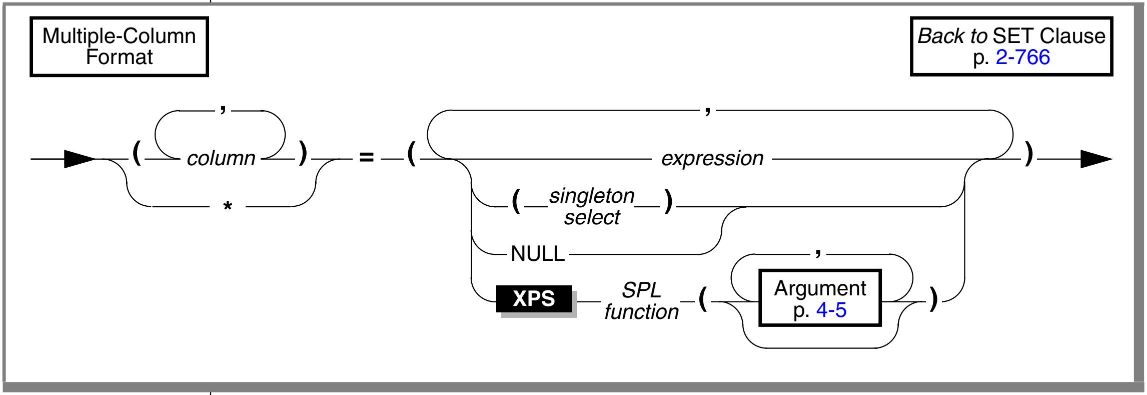
Updating the Same Column Twice

You can specify the same column more than once in the SET clause. If you do so, the column is set to the last value that you specified for the column. In the following example, the user specifies the **fname** column twice in the SET clause. For the row where the customer number is `101`, the user sets **fname** first to `gary` and then to `harry`. After the UPDATE statement executes, the value of **fname** is `harry`.

```
UPDATE customer
   SET fname = "gary", fname = "harry"
      WHERE customer_num = 101
```

Multiple-Column Format

Use the multiple-column format of the SET clause to list multiple columns and set them equal to corresponding expressions.



Element	Purpose	Restrictions	Syntax
column	Name of a column to be updated	Cannot be a serial or ROW column. The number of *column* names must equal the number of values returned to the right of the = sign.	Identifier, p. 4-189
expression	Expression that returns a value for a *column*	Cannot include aggregate functions.	Expression, p. 4-67
singleton select	Subquery that returns exactly one row	Values that the subquery returns must correspond to the columns named in the column list.	SELECT, p. 2-581
SPL function	An SPL routine that returns one or more values	Returned values must have a 1-to-1 correspondence to the columns named in the column list.	Identifier, p. 4-189

The multiple-column format of the SET clause offers the following options for listing a set of columns that you intend to update:

- Explicitly list each column, placing commas between columns and enclosing the set of columns between parentheses.
- Implicitly list all columns in the table by using an asterisk (*).

You must list each expression explicitly, placing comma (,) separators between expressions and enclosing the set of expressions between parentheses. The number of columns must equal the number of values returned by the expression list, unless the expression list includes an SQL subquery.

The following examples show the multiple-column format of the SET clause:

```
UPDATE customer
   SET (fname, lname) = ('John', 'Doe') WHERE customer_num = 101

UPDATE manufact
   SET * = ('HNT', 'Hunter') WHERE manu_code = 'ANZ'
```

Using a Subquery to Update Column Values

An expression list can include an SQL subquery that returns a single row of multiple values, provided that the number of columns that you specify, explicitly or implicitly, equals the number of values produced by the expression or expressions that follow the equal (=) sign.

The following examples show the use of subqueries:

```
UPDATE items
   SET (stock_num, manu_code, quantity) =
      ( (SELECT stock_num, manu_code FROM stock
         WHERE description = 'baseball'), 2)
   WHERE item_num = 1 AND order_num = 1001

UPDATE table1
   SET (col1, col2, col3) =
      ((SELECT MIN (ship_charge),
         MAX (ship_charge) FROM orders),
         '07/01/1997')
   WHERE col4 = 1001
```

IDS

If you are updating the supertable in a table hierarchy, the SET clause cannot include a subquery that references one of its subtables. If you are updating a subtable in a table hierarchy, a subquery in the SET clause can reference the supertable if it references only the supertable. That is, the subquery must use the SELECT… FROM ONLY (*supertable*) syntax. ♦

XPS

Using an SPL Function to Update Column Values

When you use an SPL function to update column values, the returned values of the function must have a one-to-one correspondence with the listed columns. That is, each value that the SPL function returns must be of the data type expected by the corresponding column in the column list.

If the called SPL routine contains certain SQL statements, a runtime error occurs. For information on which SQL statements cannot be used in an SPL routine that is called in a data-manipulation statement, see "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279. In the next example, the SPL function **p2( )** updates the **i2** and **c2** columns of the **t2** table:

```
CREATE PROCEDURE p2()
   RETURNING int, char(20);
   RETURN 3, ‘three’;
END PROCEDURE;
UPDATE t2 SET (i2, c2) = (p2()) WHERE i2 = 2;
```

In Extended Parallel Server, you create an SPL function with the CREATE PROCEDURE statement. The CREATE FUNCTION statement is not available.

IDS

Updating ROW-Type Columns

Use the SET clause to update a named or unnamed ROW-type column. For example, suppose you define the following named ROW type and a table that contains columns of both named and unnamed ROW types:

```
CREATE ROW TYPE address_t
(
   street CHAR(20), city CHAR(15), state CHAR(2)
);
CREATE TABLE empinfo
(
   emp_id INT
   name ROW ( fname CHAR(20), lname CHAR(20)),
   address address_t
);
```

To update an unnamed ROW type, specify the ROW constructor before the parenthesized list of field values.

The following statement updates the **name** column (an unnamed ROW type) of the **empinfo** table:

```
UPDATE empinfo SET name = ROW('John','Williams') WHERE emp_id =455
```

To update a named ROW type, specify the ROW constructor before the parenthesized list of field values and use the cast (**::**) operator to cast the ROW value as a named ROW type. The following statement updates the **address** column (a named ROW type) of the **empinfo** table:

```
UPDATE empinfo
SET address = ROW('103 Baker St','Tracy','CA')::address_t
WHERE emp_id = 3568
```

For more information on the syntax for ROW constructors, see "Constructor Expressions" on page 4-106. See also "Literal Row" on page 4-218.

E/C

The ROW-column SET clause can only support literal values for fields. To use a variable to specify a field value, you must select the ROW data into a **row** variable, use host variables for the individual field values, then update the ROW column with the **row** variable. For more information, see "Updating a Row Variable" on page 2-775. ♦

E/C

You can use ESQL/C host variables to insert *non-literal* values as:

- An entire row type into a column

  Use a **row** variable as a variable name in the SET clause to update all fields in a ROW column at one time.
- Individual fields of a ROW type

  To insert non-literal values into a ROW-type column, you can first update the elements in a **row** variable and then specify the **collection** variable in the SET clause of an UPDATE statement.

When you use a **row** variable in the SET clause, the **row** variable must contain values for each field value. For information on how to insert values into a **row** variable, see "Updating a Row Variable" on page 2-775. ♦

You can use the UPDATE to modify only some of the fields in a row:

- Specify the field names with field projection for all fields whose values remain unchanged.

  For example, the following UPDATE statement changes only the **street** and **city** fields of the **address** column of the **empinfo** table:

  ```
  UPDATE empinfo
  SET address = ROW('23 Elm St', 'Sacramento',
                 address.state)
     WHERE emp_id = 433
  ```

  The **address.state** field remains unchanged.

E/C

- Select the row into a **row** variable and update the desired fields.

  For more information, see "Updating a Row Variable" on page 2-775. ♦

IDS

Updating Collection Columns

You can use the SET clause to update values in a collection column. For more information, see "Collection Constructors" on page 4-108.

E/C

SPL

A collection variable can update a collection-type column. With a collection variable, you can insert one or more individual elements of a collection. For more information, see "Collection-Derived Table" on page 4-7. ♦

For example, suppose you define the **tab1** table as follows:

```
CREATE TABLE tab1
(
   int1 INTEGER,
   list1 LIST(ROW(a INTEGER, b CHAR(5)) NOT NULL),
   dec1 DECIMAL(5,2)
)
```

The following UPDATE statement updates a row in **tab1**:

```
UPDATE tab1
   SET list1 = LIST{ROW(2, 'zyxwv'),
      ROW(POW(2,6), ‘=64’),
      ROW(ROUND(ROOT(146)), ‘=12’)},
   where int1 = 10
```

Collection column **list1** in this example has three elements. Each element is an unnamed row type with an INTEGER field and a CHAR(5) field. The first element includes two literal values: an integer (`2`) and a quoted string (`'zyxwv'`).

The second and third elements also use a quoted string to indicate the value for the second field. They each designate the value for the first field with an expression, however, rather than with a literal value.

IDS

Updating Values in Opaque-Type Columns

Some opaque data types require special processing when they are updated. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign( )**. When you execute UPDATE on a table whose rows contain one of these opaque types, the database server automatically invokes the **assign( )** function for the type. This function can make the decision of how to store the data. For more information about the **assign( )** support function, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

XPS

Subset of FROM Clause

In Extended Parallel Server, you can use a join to determine which column values to update by specifying a FROM clause. Columns from any table in the FROM clause can appear in the WHERE clause to provide values for the columns and rows to update. For example, in the following UPDATE statement, a FROM clause introduces tables to be joined in the WHERE clause:

```
UPDATE tab1 SET tab1.a = tab2.a FROM tab1, tab2, tab3
   WHERE tab1.b = tab2.b AND tab2.c =tab3.c
```

UPDATE supports only a subset of the syntax listed in "FROM Clause" on page 2-594. You cannot include the LOCAL or the SAMPLES OF keywords.

WHERE Clause

The WHERE clause lets you specify search criteria to limit the rows to be updated. If you omit the WHERE clause, every row in the table is updated. For more information, see the "WHERE Clause" on page 2-613.

The next example uses WHERE and FROM clauses to update three columns (**state**, **zipcode**, and **phone**) in each row of the **customer** table that has a corresponding entry in a table of new addresses called **new_address**:

```
UPDATE customer
   SET (state, zipcode, phone) =
      ((SELECT state, zipcode, phone FROM new_address N
         WHERE N.cust_num = customer.customer_num))
      WHERE customer_num IN
         (SELECT cust_num FROM new_address)
```

ANSI

SQLSTATE VALUES When Updating an ANSI Database

If you update a table in an ANSI-compliant database with an UPDATE statement that contains the WHERE clause and no rows are found, the database server issues a warning.

You can detect this warning condition in either of the following ways:

- The GET DIAGNOSTICS statement sets the **RETURNED_SQLSTATE** field to the value `02000`. In an SQL API application, the **SQLSTATE** variable contains this same value.
- In an SQL API application, the **sqlca.sqlcode** and **SQLCODE** variables contain the value `100`.

The database server also sets **SQLSTATE** and **SQLCODE** to these values if the UPDATE... WHERE... is a part of a multistatement PREPARE and the database server returns no rows.

SQLSTATE VALUES When Updating a Non-ANSI Database

In a database that is not ANSI compliant, the database server does not return a warning when it finds no matching rows for the WHERE clause of an UPDATE statement. The **SQLSTATE** code is `00000` and the **SQLCODE** code is zero (`0`). If the UPDATE... WHERE... is a part of a multistatement PREPARE, however, and no rows are returned, the database server issues a warning, and sets **SQLSTATE** to `02000` and sets **SQLCODE** to `100`.

E/C

SPL

Using the WHERE CURRENT OF Clause

Use the WHERE CURRENT OF clause to update the current row of the active set of a cursor in the current element of a collection cursor.

The UPDATE statement does not advance the cursor to the next row, so the current row position remains unchanged.

IDS

You cannot use this clause if you are selecting from only one table in a table hierarchy. That is, you cannot use this option if you use the ONLY keyword. ♦

To use the WHERE CURRENT OF keywords, you must have previously used the DECLARE statement to define the *cursor* with the FOR UPDATE option.

If the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you are restricted to updating only those columns in a subsequent UPDATE...WHERE CURRENT OF statement. The advantage to specifying columns in the FOR UPDATE clause of a DECLARE statement is speed. The database server can usually perform updates more quickly if columns are specified in the DECLARE statement.

Before you can use the CURRENT OF keywords, you must declare a cursor with the FOREACH statement.

Tip: *An update cursor can perform updates that are not possible with the UPDATE statement.*

The following ESQL/C example illustrates the CURRENT OF form of the WHERE clause. In this example, updates are performed on a range of customers who receive 10-percent discounts (assume that a new column, **discount**, is added to the **customer** table). The UPDATE statement is prepared outside the WHILE loop to ensure that parsing is done only once.

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
   char fname[32],lname[32];
   int low,high;
EXEC SQL END DECLARE SECTION;
main()
{
   EXEC SQL connect to 'stores_demo';
   EXEC SQL prepare sel_stmt from
      'select fname, lname from customer \
       where cust_num between ? and ? for update';
   EXEC SQL declare x cursor for sel_stmt;
   printf("\nEnter lower limit customer number: ");
   scanf("%d", &low);
   printf("\nEnter upper limit customer number: ");
   scanf("%d", &high);
   EXEC SQL open x using :low, :high;
   EXEC SQL prepare u from
      'update customer set discount = 0.1  where current of x';
   while (1)
      {
      EXEC SQL fetch x into :fname, :lname;
      if ( SQLCODE == SQLNOTFOUND) break;
      }
   printf("\nUpdate %.10s %.10s (y/n)?", fname, lname);
   if (answer = getch() == 'y')
      EXEC SQL execute u;
   EXEC SQL close x;
}
```

IDS

E/C

Updating a Row Variable

The UPDATE statement with the Collection-Derived-Table segment allows you to update fields in a **row** variable. The Collection-Derived-Table segment identifies the **row** variable in which to update the fields. For more information, see "Collection-Derived Table" on page 4-7.

To update fields

1. Create a **row** variable in your ESQL/C program.
2. Optionally, select a row-type column into the **row** variable with the SELECT statement (without the Collection-Derived-Table segment).
3. Update fields of the **row** variable with the UPDATE statement and the Collection-Derived-Table segment.
4. After the **row** variable contains the correct fields, you then use the UPDATE or INSERT statement on a table or view name to save the **row** variable in the ROW column (named or unnamed).

The UPDATE statement and the Collection-Derived-Table segment allow you to update a particular field or group of fields in the **row** variable. Specify the new field values in the SET clause. For example, the following UPDATE changes the **x** and **y** fields in the **myrect** ESQL/C **row** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL select into :myrect from rectangles where area = 64;
EXEC SQL update table(:myrect) set x=3, y=4;
```

Suppose that after the SELECT statement, the **myrect2** variable has the values `x=0`, `y=0`, `length=8`, and `width=8`. After the UPDATE statement, the **myrect2** variable has field values of `x=3`, `y=4`, `length=8`, and `width=8`. You cannot use a **row** variable in the Collection-Derived-Table segment of an INSERT statement.

You can, however, use the UPDATE statement and the Collection-Derived-Table segment to insert new field values into a **row** host variable, if you specify a value for every field in the row.

For example, the following code fragment inserts new field values into the **row** variable **myrect** and then inserts this **row** variable into the database:

```
EXEC SQL update table(:myrect)
   set x=3, y=4, length=12, width=6;
EXEC SQL insert into rectangles
   values (72, :myrect);
```

If the **row** variable is an untyped variable, you must use a SELECT statement *before* the UPDATE so that ESQL/C can determine the data types of the fields. An UPDATE of fields in a **row** variable cannot include a WHERE clause.

The **row** variable can store the field values of the row, but it has no intrinsic connection with a database column. Once the **row** variable contains the correct field values, you must then save the variable into the ROW column with one of the following SQL statements:

- To update the ROW column in the table with contents of the **row** variable, use an UPDATE statement on a table or view name and specify the **row** variable in the SET clause. (For more information, see "Updating ROW-Type Columns" on page 2-770.)
- To insert a **row** into a column, use the INSERT statement on a table or view name and specify the **row** variable in the VALUES clause. (For more information, see "Inserting Values into ROW-Type Columns" on page 2-497.)

For examples of SPL ROW variables, see the *IBM Informix Guide to SQL: Tutorial*. For more information on using ESQL/C **row** variables, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

Related Information

Related statements: DECLARE, INSERT, OPEN, SELECT, and FOREACH

For a task-oriented discussion of the UPDATE statement, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion of the GLS aspects of the UPDATE statement, see the *IBM Informix GLS User's Guide*.

For information on how to access row and collections with ESQL/C host variables, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

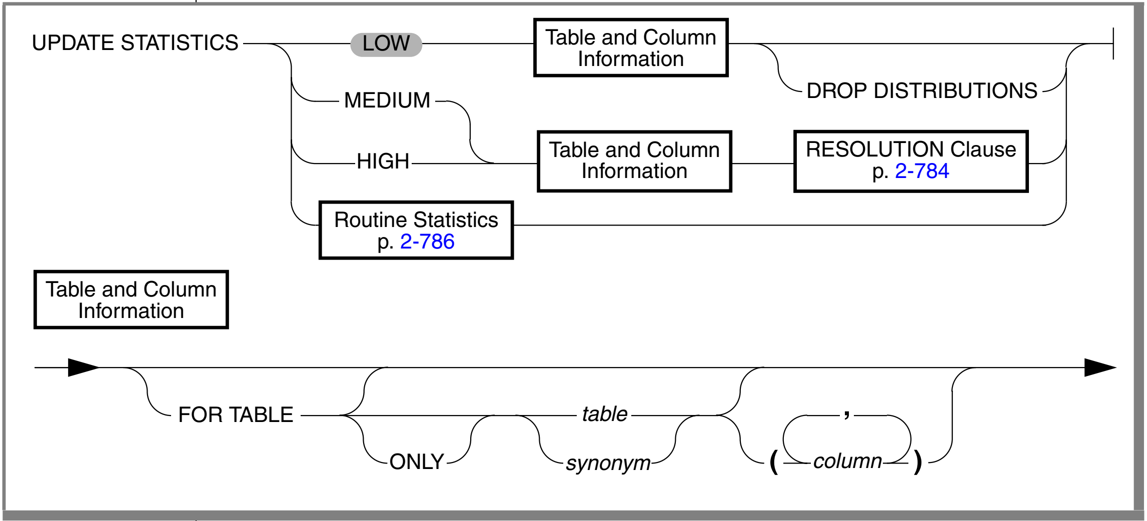
+

UPDATE STATISTICS

Use the UPDATE STATISTICS statement to perform any of the following tasks:

- Determine the distribution of column values.
- Update system catalog tables that the database server uses to optimize queries.
- Force reoptimization of SPL routines.
- Convert existing indexes when you upgrade the database server.

Syntax


UPDATE STATISTICS
LOW
MEDIUM
HIGH
Table and Column Information
DROP DISTRIBUTIONS
Table and Column Information
RESOLUTION Clause p. 2-784
Routine Statistics p. 2-786
Table and Column Information
FOR TABLE
ONLY
table
synonym
,
(
column
)

Element	Purpose	Restrictions	Syntax
column	A column in *table* or *synonym*	Must exist. With MEDIUM or HIGH keywords, *column* cannot be of BYTE or TEXT data type.	Identifier, p. 4-189
synonym	Synonym for a table whose statistics are to be updated	The *synonym* and the table to which it points must exist in the current database.	Database Object Name, p. 4-46
table	Table for which statistics are to be updated	Must exist in the current database.	Database Object Name, p. 4-46

Usage

You cannot update the statistics used by the optimizer for a table or UDR that is external to the current database. That is, you cannot update statistics on remote database objects.

Scope of UPDATE STATISTICS

If you do not specify any clause that begins with the FOR keyword, statistics are updated for every table and SPL routine in the current database, including the system catalog tables. Similarly, if you use a clause that begins with the FOR keyword, but do not specify a table or SPL routine name, the database server updates the statistics for all tables, including temporary tables, or all SPL routines in the current database.

If you use the FOR TABLE clause without a specific table name to build distributions on all of the tables in the database, distributions are also built on all of the temporary tables in your session.

Updating Statistics for Tables

Although a change to the database might obsolete the corresponding statistics in the **systables**, **syscolumns**, **sysindexes**, and **sysdistrib** system catalog tables, the database server does not automatically update them.

Issue an UPDATE STATISTICS statement in the following situations to ensure that the stored distribution information reflects the state of the database:

- You perform extensive modifications to a table.
- An application changes the distribution of column values.

  The UPDATE STATISTICS statement reoptimizes queries on the modified objects.
- You upgrade a database for use with a newer database server.

  The UPDATE STATISTICS statement converts the old indexes to conform to the newer database server index format and implicitly drops the old indexes.

  You can choose to convert the indexes table by table or for the entire database at one time. Follow the conversion guidelines in the *IBM Informix Migration Guide*.

If your application makes many modifications to the data in a particular table, update the system catalog data for that table routinely with the UPDATE STATISTICS statement to improve query efficiency. The term *many modifications* is relative to the resolution of the distributions. If the data changes do not change the distribution of column values, you do not need to execute UPDATE STATISTICS.

XPS

In Extended Parallel Server, the UPDATE STATISTICS statement does not update, maintain, or collect statistics on indexes. The statement does not update the **syscolumns** and **sysindexes** tables. Any information about indexes, the **syscolumns,** and the **sysindexes** tables in the following pages does not apply to Extended Parallel Server. ♦

IDS

Using the ONLY Keyword

Use the ONLY keyword to collect data for one table in a hierarchy of typed tables. If you do not specify the ONLY keyword and the table that you specify has subtables, the database server creates distributions for that table and every table under it in the hierarchy.

For example, assume your database has the typed table hierarchy that appears in Figure 2-2, which shows a supertable named **employee** that has a subtable named **sales_rep**. The **sales_rep** table, in turn, has a subtable named **us_sales_rep.**

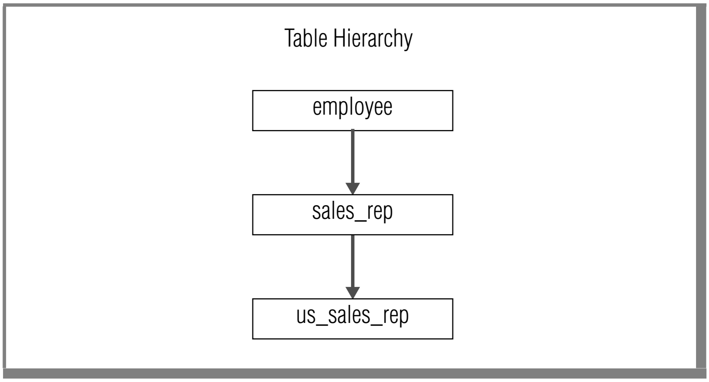


Figure 2-2
Example of Typed Table Hierarchy

When the following statement executes, the database server generates statistics on both the **sales_rep** and **us_sales_rep** tables:

```
UPDATE STATISTICS FOR TABLE sales_rep
```

In contrast, the following example generates statistical data for each column in table **sales_rep** but does not act on tables **employee** or **us_sales_rep**:

```
UPDATE STATISTICS FOR TABLE ONLY sales_rep
```

Because neither of the previous examples specified the level at which to update the statistical data, the database server uses the LOW mode by default.

IDS

Examining Index Pages

In Dynamic Server, when you execute the UPDATE STATISTICS statement in any mode, the database server reads through index pages to:

- Compute statistics for the query optimizer
- Locate pages that have the delete flag marked as `1`

If pages are found with the delete flag marked as `1`, the corresponding keys are removed from the B-tree cleaner list.

This operation is particularly useful if a system failure causes the B-tree cleaner list (which exists in shared memory) to be lost. To remove the B-tree items that have been marked as deleted but are not yet removed from the B-tree, run the UPDATE STATISTICS statement. For information on the B-tree cleaner list, see your *Administrator's Guide*.

Using the LOW Mode Option

Use the LOW option to generate and update some of the relevant statistical data regarding table, row, and page count statistics in the **systables** system catalog table. If you do not specify any mode, the LOW mode is the default.

IDS

In Dynamic Server, the LOW mode also generates and updates some index and column statistics for specified columns in the **syscolumns** and the **sysindexes** system catalog tables.

The LOW mode generates the least amount of information about the column. If you want the UPDATE STATISTICS statement to do minimal work, specify a column that is not part of an index. The **colmax** and **colmin** values in **syscolumns** are not updated unless there is an index on the column.

The following example updates statistics on the **customer_num** column of the **customer** table:

```
UPDATE STATISTICS LOW FOR TABLE customer (customer_num)
```

Because the LOW mode option does not update data in the **sysdistrib** system catalog table, all distributions associated with the **customer** table remain intact, even those that already exist on the **customer_num** column. ♦

Using the DROP DISTRIBUTIONS Option

Use the DROP DISTRIBUTIONS option to force the removal of distribution information from the **sysdistrib** system catalog table.

When you specify the DROP DISTRIBUTIONS option, the database server removes the existing distribution data for the column or columns that you specify. If you do not specify any columns, the database server removes all the distribution data for that table.

You must have the DBA privilege or be owner of the table to use this option.

The following example shows how to remove distributions for the **customer_num** column in the **customer** table:

```
UPDATE STATISTICS LOW
   FOR TABLE customer (customer_num) DROP DISTRIBUTIONS
```

As the example shows, you drop the distribution data at the same time you update the statistical data that the LOW mode option generates.

Using the MEDIUM Mode Option

Use the MEDIUM mode option to update the same statistics that you can perform with the LOW mode option and also generate statistics about the distribution of data values for each specified column. The database server places distribution information in the **sysdistrib** system catalog table.

If you use the MEDIUM mode option, the database server scans tables at least once and takes longer to execute on a given table than the LOW mode option.

When you use the MEDIUM mode option, the data for the distributions is obtained by sampling a percentage of data rows, using a statistical confidence level that you specify, or else a default confidence level of 95 percent.

Because the MEDIUM sample size is usually much smaller than the actual number of rows, this mode executes more quickly than the HIGH mode.

Because the distribution is obtained by sampling, the results can vary, because different samples of rows might produce different distribution results. If the results vary significantly, you can lower the resolution percent or increase the confidence level to obtain more consistent results.

If you specify no RESOLUTION clause, the default average percentage of the sample in each bin is `2.5`, dividing the range into 40 intervals. If you do not specify a value for *confidence_level*, the default level is `0.95`. This value can be roughly interpreted to mean that 95 times out of 100, the estimate is not statistically different from what would be obtained from HIGH distributions.

You must have the DBA privilege or be the owner of the table to create MEDIUM distributions. For more on the MEDIUM and HIGH mode options, see the "Resolution Clause" on page 2-784.

Using the HIGH Mode Option

Use the HIGH mode option to update the same statistics that you can perform with the LOW mode option and also generate statistics about the distribution of data values for each specified column. The database server places distribution information in the **sysdistrib** system catalog table.

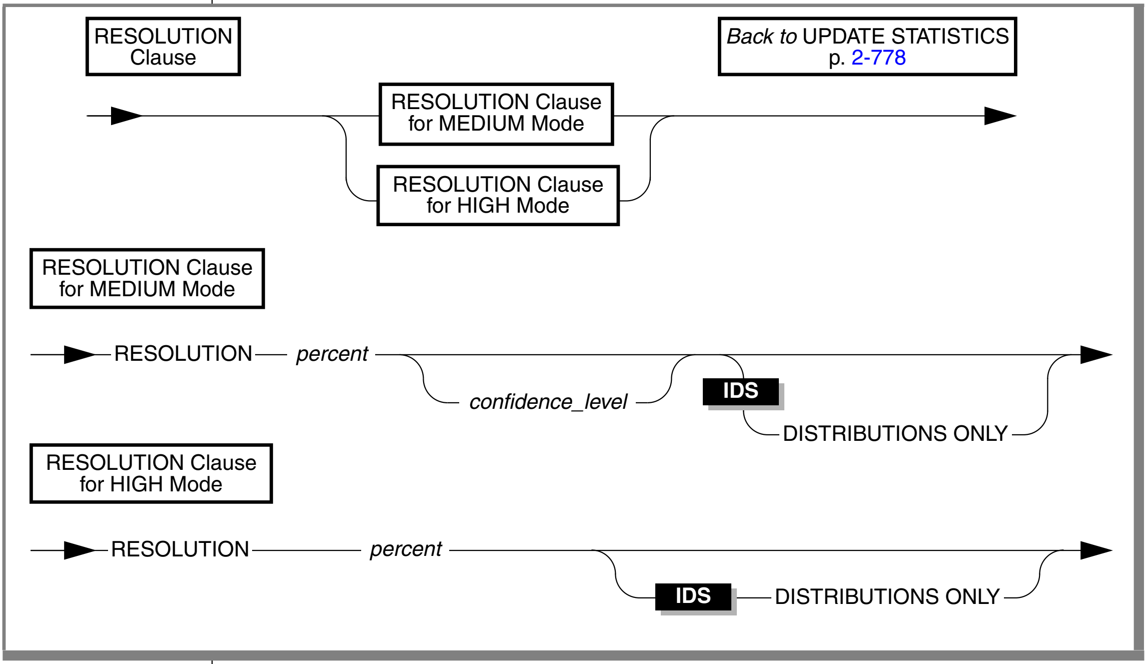
If you do not specify a RESOLUTION clause, the default percentage of data distributed to every bin is `0.5`, partitioning the range of values for each column into 200 intervals.

The constructed distribution is exact. Because more information is gathered, this mode executes more slowly than LOW or MEDIUM modes. If you use the HIGH mode option of UPDATE STATISTICS, the database server can take considerable time to gather the information across the database, particularly a database with large tables. The HIGH keyword might scan each table several times (for each column). To minimize processing time, specify a table name and column names within that table.

You must have the DBA privilege or be the owner of the table to create HIGH distributions. For more information on the MEDIUM and HIGH mode options, see the "Resolution Clause" on page 2-784.

Resolution Clause

Use the Resolution clause to adjust the size of the distribution bin, designate whether or not to avoid calculating data on indexes, and with the MEDIUM mode, to adjust the confidence level.


RESOLUTION Clause
Back to UPDATE STATISTICS p. 2-778
RESOLUTION Clause for MEDIUM Mode
RESOLUTION Clause for HIGH Mode
RESOLUTION Clause for MEDIUM Mode
RESOLUTION
percent
confidence_level
IDS
DISTRIBUTIONS ONLY
RESOLUTION Clause for HIGH Mode
RESOLUTION
percent
IDS
DISTRIBUTIONS ONLY

Element	Purpose	Restrictions	Syntax
confidence_level	Estimated fraction of the time that sampling in MEDIUM mode should produce the same results as the exact HIGH mode. Default level is `0.95`.	Must be within the range from `0.80` (minimum) to `0.99` (maximum).	Literal Number, p. 4-216
percent	Percentage of sample in each bin of distribution Default is `2.5` for MEDIUM and `0.5` for HIGH.	Minimum resolution is 1/*nrows*, for *nrows* the number of rows in the table.	Literal Number, p. 4-216

A *distribution* is a mapping of the data in a column into a set of column values, ordered by magnitude or by collation. The range of these sample values is partitioned into disjunct intervals, called *bins*, each containing an approximately equal portion of the sample of column values. For example, if one bin holds 2 percent of the data, 50 such intervals hold the entire sample.

Some statistical texts call these bins *equivalence categories*. Each contains a subset of the range of the data values that are sampled from the column.

The optimizer estimates the effect of a WHERE clause by examining, for each column included in the WHERE clause, the proportionate occurrence of data values contained in the column.

You cannot create distributions for BYTE or TEXT columns. If you include a BYTE or TEXT column in an UPDATE STATISTICS statement that specifies medium or high distributions, no distributions are created for those columns. Distributions are constructed for other columns in the list, however, and the statement does not return an error.

Columns of the VARCHAR data type do not use overflow bins, even when multiple bins are being used for duplicate values.

The amount of space that the **DBUPSPACE** environment variable specifies determines the number of times the database server scans the designated table to construct a distribution.

IDS

Using the DISTRIBUTIONS ONLY Option to Suppress Index Information

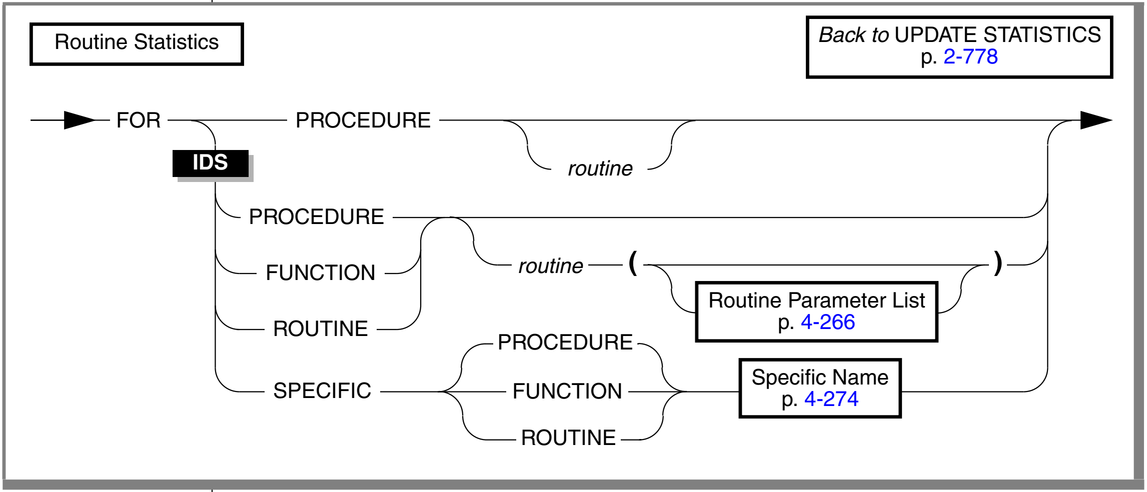
In Dynamic Server, when you specify the DISTRIBUTIONS ONLY option, you do not update index information. This option does not affect existing index information.

Use this option to avoid the examination of index information that can consume considerable processing time.

This option does not affect the recalculation of information on tables, such as the number of pages used, the number of rows, and fragment information. UPDATE STATISTICS needs this information to construct accurate column distributions and requires little time and system resources to collect it.

Routine Statistics

Use the Routine Statistics segment of the UPDATE STATISTICS statement to update the optimized execution plans for SPL routines in the **sysprocplan** system catalog table.



Element	Purpose	Restrictions	Syntax
routine	Name declared for a SPL routine in a CREATE FUNCTION or CREATE PROCEDURE statement	Must reside in current database. In an ANSI-compliant database, qualify *routine* with *owner* if you are not *owner*.	Database Object Name, p. 4-46

The following table explains the keywords of the Routine Statistics segment.

Keyword	Which Execution Plan is Reoptimized
SPECIFIC	The plan for the SPL routine called *specific name*
FUNCTION	The plan for any SPL function with the specified name (and with parameter types that match *routine parameter list*, if supplied)
PROCEDURE	The plan for any SPL procedure with the specified name (and parameter types that match *routine parameter list*, if supplied)
ROUTINE	The plan for SPL functions and procedures with the specified name (and parameter types that match *routine parameter list*, if supplied)

If you omit *routine*, execution plans are reoptimized for all SPL routines in the current database.

The **sysprocplan** system catalog table stores execution plans for SPL routines. Two actions update the **sysprocplan** system catalog table:

- Execution of an SPL routine that uses a modified table
- The UPDATE STATISTICS statement

If you change a table that an SPL routine references, you can run UPDATE STATISTICS to reoptimize on demand, rather than waiting until the next time an SPL routine that uses the table executes.

IDS

Updating Statistics for Columns of User-Defined Types

To collect statistics for a column of a user-defined data type, you must specify either MEDIUM or HIGH mode. When you execute UPDATE STATISTICS, the database server does not collect values for the **colmin** and **colmax** columns of the **syscolumns** table for columns that hold user-defined data types.

To drop statistics for a column that holds one of these data types, you must execute UPDATE STATISTICS in the LOW mode with the DROP DISTRIBUTIONS option. When you use this option, the database server removes the row in the **sysdistrib** system catalog table that corresponds to the **tableid** and **column**. In addition, the database server removes any large objects that might have been created for storing the statistics information.

Requirements

UPDATE STATISTICS collects statistics for opaque data types only if you have defined user-defined routines for **statcollect( )**, **statprint( )**, and the selectivity functions. You must have usage permissions on these routines.

In some cases, UPDATE STATISTICS also requires an sbspace as specified by the SYSSBSPACENAME onconfig parameter. For information about how to provide statistical data for a column, refer to the *IBM Informix DataBlade API Programmer's Guide*. For information about SYSSBSPACENAME, refer to your *Administrator's Reference*.

IDS

Updating Statistics When You Upgrade the Database Server

When you upgrade a database to use with a newer database server, you can use the UPDATE STATISTICS statement to convert the indexes to the form that the newer database server uses. You can choose to convert the indexes one table at a time or for the entire database at one time. Follow the conversion guidelines that are outlined in the *IBM Informix Migration Guide*.

When you use the UPDATE STATISTICS statement to convert the indexes to use with a newer database server, the indexes are implicitly dropped and re-created. The only time that an UPDATE STATISTICS statement causes table indexes to be implicitly dropped and re-created is when you upgrade a database for use with a newer database server.

Performance

The more specific you make the list of objects that UPDATE STATISTICS examines, the faster it completes execution. Limiting the number of columns distributed speeds the update. Similarly, precision affects the speed of the update. If all other keywords are the same, LOW works fastest, but HIGH examines the most data.

Related Information

Related statements: SET EXPLAIN and SET OPTIMIZATION

For a discussion of the performance implications of UPDATE STATISTICS, see your *Performance Guide*.

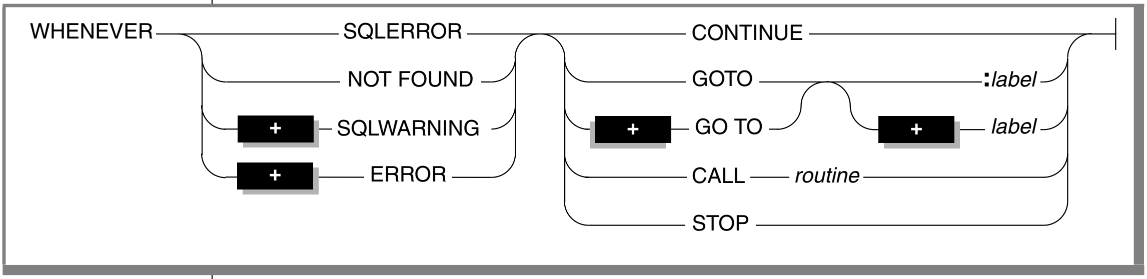
For a discussion of how to use the **dbschema** utility to view distributions created with UPDATE STATISTICS, see the *IBM Informix Migration Guide*.

E/C

WHENEVER

Use the WHENEVER statement to trap exceptions that occur during the execution of SQL statements. Use this statement with ESQL/C.

Syntax



Element	Purpose	Restrictions	Syntax
label	Statement label to which program control transfers when an exception occurs	Must exist in the same source-code module.	See language-specific rules for labels.
routine	Name of a user-defined routine (UDR) to be invoked when an exception occurs	No arguments; UDR must exist at compile time.	Database Object Name, p. 4-46

Usage

The WHENEVER statement is equivalent to placing an exception-checking routine after every SQL statement. The following table summarizes the types of exceptions for which you can check with the WHENEVER statement.

Type of Exception	WHENEVER Keyword	For More Information
Errors	SQLERROR *or* ERROR	page 2-791
Warnings	SQLWARNING	page 2-792
Not Found *or* End of Data	NOT FOUND	page 2-792

Programs that do not use the WHENEVER statement do not automatically abort when an exception occurs. Such programs must explicitly check for exceptions and take whatever corrective action their logic specifies. If you do not check for exceptions, the program simply continues running. If errors occur, however, the program might not perform its intended purpose.

The first keyword that follows WHENEVER specifies some type of exceptional condition; the last part of the statement specifies some action to take when the exception is encountered (or no action, if CONTINUE is specified). The following table summarizes possible actions that WHENEVER can specify.

Type of Action	WHENEVER Keyword	For More Information
Continue program execution	CONTINUE	page 2-793
Stop program execution	STOP	page 2-793
Transfer control to a specified label	GOTO GO TO	page 2-793
Transfer control to a UDR	CALL	page 2-794

The Scope of WHENEVER

Whenever is a preprocessor directive, rather than an executable statement. The ESQL/C preprocessor, not the database server, handles the interpretation of the WHENEVER statement. When the preprocessor encounters a WHENEVER statement in an ESQL/C source file, it inserts appropriate code into the preprocessed code after each SQL statement, based on the exception and the action that WHENEVER specifies. The scope of the WHENEVER statement begins where the statement appears in the source module and remains in effect until the preprocessor encounters one or the other of the following things while sequentially processing the source module:

- The next WHENEVER statement with the same condition (SQLERROR, SQLWARNING, or NOT FOUND) in the same source module
- The end of the source module

The following ESQL/C example program has three WHENEVER statements, two of which are WHENEVER SQLERROR statements. Line 4 uses STOP with SQLERROR to override the default CONTINUE action for errors.

Line 8 specifies the CONTINUE keyword to return the handling of errors to the default behavior. For all SQL statements between lines 4 and 8, the preprocessor inserts code that checks for errors and halts program execution if an error occurs. Therefore, any errors that the INSERT statement on line 6 generates cause the program to stop.

After line 8, the preprocessor does not insert code to check for errors after SQL statements. Therefore, any errors that the INSERT statement (line 10), the SELECT statement (line 11), and DISCONNECT statement (line 12) generate are ignored. The SELECT statement, however, does not stop program execution if it does not locate any rows; the WHENEVER statement on line 7 tells the program to continue if such an exception occurs.

```
1  main()
2  {
3  EXEC SQL connect to 'test';
4  EXEC SQL WHENEVER SQLERROR STOP;
5  printf("\n\nGoing to try first insert\n\n");
6  EXEC SQL insert into test_color values ('green');
7  EXEC SQL WHENEVER NOT FOUND CONTINUE;
8  EXEC SQL WHENEVER SQLERROR CONTINUE;
9  printf("\n\nGoing to try second insert\n\n");
10 EXEC SQL insert into test_color values ('blue');
11 EXEC SQL select paint_type from paint where color='red';
12 EXEC SQL disconnect all;
13 printf("\n\nProgram over\n\n");
14 }
```

SQLERROR Keyword

If you use the SQLERROR keyword, any SQL statement that encounters an error is handled as the WHENEVER SQLERROR statement directs. If an error occurs, the **sqlcode** variable (**sqlca.sqlcode**, **SQLCODE**) is set to a value less than zero (0) and the SQLSTATE variable is set to a class code with a value greater than 02.

The next example terminates program execution if an SQL error is detected:

```
WHENEVER SQLERROR STOP
```

If you do not use any WHENEVER SQLERROR statements in a program, the default for WHENEVER SQLERROR is CONTINUE.

ERROR Keyword

Within the WHENEVER statement (and only in this context), the keyword ERROR is a synonym for the SQLERROR keyword.

SQLWARNING Keyword

If you use the SQLWARNING keyword, any SQL statement that generates a warning is handled as the WHENEVER SQLWARNING statement directs. If a warning occurs, the first field of the warning structure in SQLCA (**sqlca.sqlwarn.sqlwarn0**) is set to `W`, and the SQLSTATE variable is set to a class code of `01`.

In addition to setting the first field of the warning structure, a warning also sets an additional field to `W`. The field that is set indicates the type of warning that occurred.

The next statement causes execution to stop if a warning condition exists:

```
WHENEVER SQLWARNING STOP
```

If you do not use any WHENEVER SQLWARNING statements in a program, the default action for WHENEVER SQLWARNING is CONTINUE.

NOT FOUND Keywords

If you use the NOT FOUND keywords, exception handling for SELECT and FETCH statements (including implicit SELECT and FETCH statements in FOREACH and UNLOAD statements) is treated differently from other SQL statements. The NOT FOUND keyword checks for the following cases:

- The **End of Data** condition: a FETCH statement that attempts to get a row beyond the first or last row in the active set
- The **Not Found** condition: a SELECT statement that returns no rows

In each case, the **sqlcode** variable is set to `100`, and the SQLSTATE variable has a class code of `02`. For the name of the **sqlcode** variable in each IBM Informix product, see the table in "SQLERROR Keyword" on page 2-791.

The following statement calls the **no_rows()** function each time the NOT FOUND condition exists:

```
WHENEVER NOT FOUND CALL no_rows
```

If you do not use any WHENEVER NOT FOUND statements in a program, the default for WHENEVER NOT FOUND is CONTINUE.

CONTINUE Keyword

Use the CONTINUE keyword to instruct the program to ignore the exception and to continue execution at the next statement after the SQL statement. The default action for all exceptions is CONTINUE. You can use this keyword to turn off a previously specified action for an exceptional condition.

STOP Keyword

Use the STOP keyword to instruct the program to stop execution when the specified exception occurs. The following statement halts execution of an ESQL/C program each time that an SQL statement generates a warning:

```
EXEC SQL WHENEVER SQLWARNING STOP;
```

GOTO Keyword

Use the GOTO clause to transfer control to the statement that the label identifies when a specified exception occurs. The GOTO keyword is ANSI-compliant syntax for this feature of embedded SQL languages like ESQL/C. (The GO TO keywords as a synonym for GOTO are an Informix extension.)

The following example shows a WHENEVER statement in ESQL/C code that transfers control to the label **missing** each time that the NOT FOUND condition occurs:

```
query_data()
   ...
   EXEC SQL WHENEVER NOT FOUND GO TO missing;
   ...
   EXEC SQL fetch lname into :lname;
   ...
   missing:
      printf("No Customers Found\n");
```

Within the scope of the WHENEVER GOTO statement, you must define the labeled statement in *each* routine that contains SQL statements. If your program contains more than one user-defined function, you might need to include the labeled statement and its code in *each* function.

If the preprocessor encounters an SQL statement within the scope of a WHENEVER... GOTO statement, but within a routine that does not have the specified label, the preprocessor tries to insert the code associated with the labeled statement but generates an error when it cannot find the label.

To correct this error, either put a labeled statement with the same label name in each UDR, or issue another WHENEVER statement to reset the error condition, or use the CALL clause to call a separate function.

CALL Clause

Use the CALL clause to transfer program control to the specified UDR when the specified type of exception occurs. Do not include parentheses after the UDR name. The following WHENEVER statement causes the program to call the **error_recovery()** function if the program detects an error:

```
EXEC SQL WHENEVER SQLERROR CALL error_recovery;
```

When the UDR returns, execution resumes at the next statement after the line that is causing the error. If you want to halt execution when an error occurs, include statements that terminate the program as part of the specified UDR.

Observe the following restrictions on the specified routine:

- The UDR cannot accept arguments nor can it return values. If it needs external information, use global variables or the WHENEVER... GOTO option to transfer program control to a label that calls the UDR.
- You cannot specify the name of an SPL routine in the CALL clause. To call an SPL routine, use the CALL clause to invoke a UDR that contains the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
- Make sure that all functions within the scope of WHENEVER... CALL statements can find a declaration of the specified function.

Related Information

Related statements: EXECUTE FUNCTION, EXECUTE PROCEDURE, and FETCH

For discussions on exception handling and error checking, see the *IBM Informix ESQL/C Programmer's Manual*.

Chapter

SPL Statements

3

In This Chapter

You can use Stored Procedure Language (SPL) statements to write SPL routines (formerly referred to as *stored procedures*), and you can store these routines in the database. SPL routines are effective tools for controlling SQL activity. This chapter contains descriptions of the SPL statements. The description of each statement includes the following information:

- A brief introduction that explains the purpose of the statement
- A syntax diagram that shows how to enter the statement correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

If a statement is composed of multiple clauses, the statement description provides the same set of information for each clause.

For task-oriented information about using SPL routines, see the *IBM Informix Guide to SQL: Tutorial*.

XPS

In Extended Parallel Server, to create an SPL function you must use the CREATE PROCEDURE statement or the CREATE PROCEDURE FROM statement. Extended Parallel Server does not support the CREATE FUNCTION nor the CREATE FUNCTION FROM statement. ♦

IDS

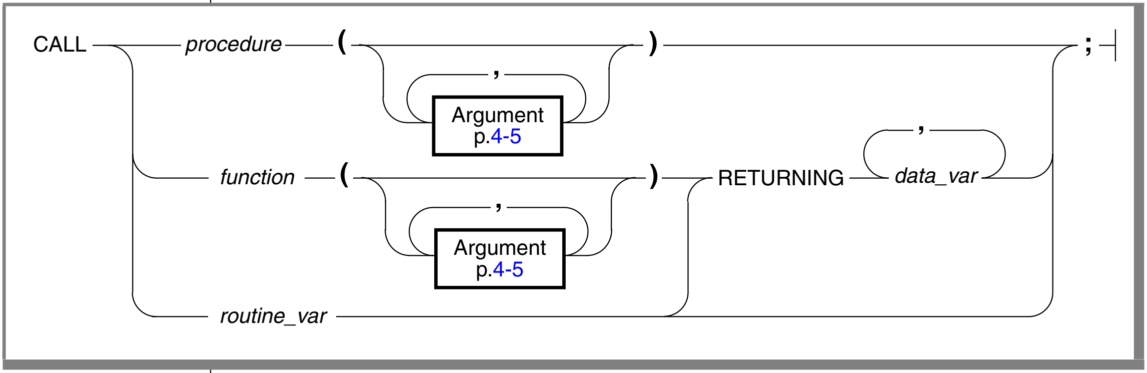
In Dynamic Server, for backward compatibility, you can create an SPL function with the CREATE PROCEDURE or CREATE PROCEDURE FROM statement. For external functions, you must use the CREATE FUNCTION or CREATE FUNCTION FROM statement. It is recommended that you use the CREATE FUNCTION or CREATE FUNCTION FROM statement when you create new user-defined functions. ♦

The SPL language does not support dynamic SQL. You cannot include any of the SQL statements that Chapter 1 classifies as “Dynamic Management Statements” within an SPL routine.

CALL

Use the CALL statement to execute a user-defined routine (UDR) from within an SPL routine.

Syntax



Element	Purpose	Restrictions	Syntax
data_var	Variable to receive the values *function* returns	The data type of *data_var* must be appropriate for the returned value.	Identifier, p. 4-189
function, procedure	User-defined function or procedure	The function or procedure must exist.	Database Object Name, p. 4-46
routine_var	Variable that contains the name of a UDR	Must be a character data type that contains the non-NULL name of an existing UDR.	Identifier, p. 4-189

Usage

The CALL statement invokes a UDR. The CALL statement is identical in behavior to the EXECUTE PROCEDURE and EXECUTE FUNCTION statements, but you can only use CALL from within an SPL routine.

You can use CALL in an ESQL/C program or with DB-Access, but only if the statement is in an SPL routine that the program or DB-Access executed.

If you CALL a user-defined *function*, you must specify a RETURNING clause.

Specifying Arguments

If a CALL statement contains more arguments than the UDR expects, you receive an error.

If CALL specifies fewer arguments than the UDR expects, the arguments are said to be missing. The database server initializes missing arguments to their corresponding default values. (See "CREATE PROCEDURE" on page 2-182 and "CREATE FUNCTION" on page 2-133.) This initialization occurs before the first executable statement in the body of the UDR. If missing arguments do not have default values, they are initialized to the value of UNDEFINED. An attempt to use any variable of UNDEFINED value results in an error.

In each UDR call, you have the option of specifying parameter names for the arguments you pass to the UDR. Each of the following examples are valid for a UDR that expects character arguments named `t`, `n`, and `d`, in that order:

```
CALL add_col (t='customer', n = 'newint', d ='integer');
CALL  add_col('customer','newint','integer');
```

The syntax is described in more detail in "Arguments" on page 4-5.

Receiving Input from the Called UDR

The RETURNING clause specifies the data variable that receives values that a a called function returns.

The following example shows two UDR calls:

```
CREATE PROCEDURE not_much()
   DEFINE i, j, k INT;
   CALL no_args (10,20);
   CALL yes_args (5) RETURNING i, j, k;
END PROCEDURE
```

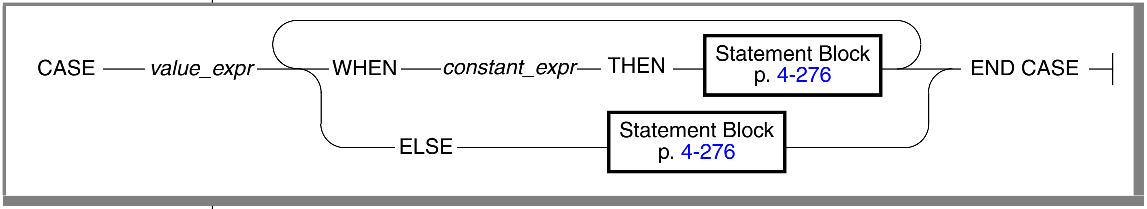
The first routine call (**no_args**) expects no returned values. The second routine call is to a function (**yes_args**), which expects three returned values. The **not_much()** procedure declares three integer variables (**i**, **j**, and **k**) to receive the returned values from **yes_args**.

XPS

CASE

Use the CASE statement when you need to take one of many branches depending on the value of an SPL variable or a simple expression. The CASE statement is a fast alternative to the IF statement.

Syntax



Element	Purpose	Restrictions	Syntax
constant_expr	Expression that specifies a literal value	Can be a literal number, quoted string, literal datetime, or literal interval. The data type must be compatible with the data type of *value_expr*.	Constant Expression, p. 4-95
value_expr	Expression that returns a value	An SPL variable or any other type of expression that returns a value.	Expression, p. 4-67

Usage

You can use the CASE statement to create a set of conditional branches within an SPL routine. Both the WHEN and ELSE clauses are optional, but you must supply one or the other. If you do not specify either a WHEN clause or an ELSE clause, you receive a syntax error.

How the Database Server Executes a CASE Statement

The database server executes the CASE statement in the following way:

- The database server evaluates the *value_expr* parameter.
- If the resulting value matches a literal value specified in the *constant_expr* parameter of a WHEN clause, the database server executes the statement block that follows the THEN keyword in that WHEN clause.

- If the value resulting from the evaluation of the *value_expr* parameter matches the *constant_expr* parameter in more than one WHEN clause, the database server executes the statement block that follows the THEN keyword in the first matching WHEN clause in the CASE statement.
- After the database server executes the statement block that follows the THEN keyword, it executes the statement that follows the CASE statement in the SPL routine.
- If the value of the *value_expr* parameter does not match the literal value specified in the *constant_expr* parameter of any WHEN clause, and if the CASE statement includes an ELSE clause, the database server executes the statement block that follows the ELSE keyword.
- If the value of the *value_expr* parameter does not match the literal value specified in the *constant_expr* parameter of any WHEN clause, and if the CASE statement does not include an ELSE clause, the database server executes the statement that follows the CASE statement in the SPL routine.
- If the CASE statement includes an ELSE clause but not a WHEN clause, the database server executes the statement block that follows the ELSE keyword.

Computation of the Value Expression in CASE

The database server computes the value of the *value_expr* parameter only one time. It computes this value at the start of execution of the CASE statement. If the value expression specified in the *value_expr* parameter contains SPL variables and the values of these variables change subsequently in one of the statement blocks within the CASE statement, the database server does not recompute the value of the *value_expr* parameter. So a change in the value of any variables contained in the *value_expr* parameter has no effect on the branch taken by the CASE statement.

Valid Statements in the Statement Block

The statement block that follows the THEN or ELSE keywords can include any SQL statement or SPL statement that is allowed in the statement block of an SPL routine. For more information, see "Statement Block" on page 4-276.

Example of CASE Statement

In the following example, the CASE statement initializes one of a set of SPL variables (named **j**, **k**, **l**, and **m**) to the value of an SPL variable named **x**, depending on the value of another SPL variable named **i**:

```
CASE i
   WHEN 1 THEN
      LET j = x;
   WHEN 2 THEN
      LET k = x;
   WHEN 3 THEN
      LET l = x;
   WHEN 4 THEN
      LET m = x;
   ELSE
      RAISE EXCEPTION 100; --illegal value
END CASE
```

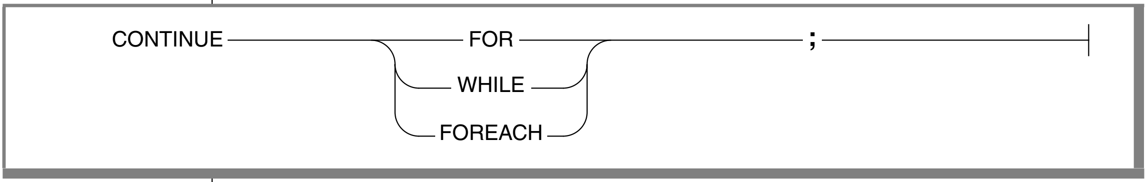
Related Information

Related statement: IF

CONTINUE

Use the CONTINUE statement to start the next iteration of the innermost FOR, WHILE, or FOREACH loop.

Syntax



Usage

When control of execution passes to a CONTINUE statement, the SPL routine skips the rest of the statements in the innermost loop of the indicated type. Execution continues at the top of the loop with the next iteration.

In the following example, the **loop_skip** function inserts values 3 through 15 into the table **testtable**. The function also returns values 3 through 9 and 13 through 15 in the process. The function does not return the value 11 because it encounters the CONTINUE FOR statement. The CONTINUE FOR statement causes the function to skip the RETURN WITH RESUME statement.

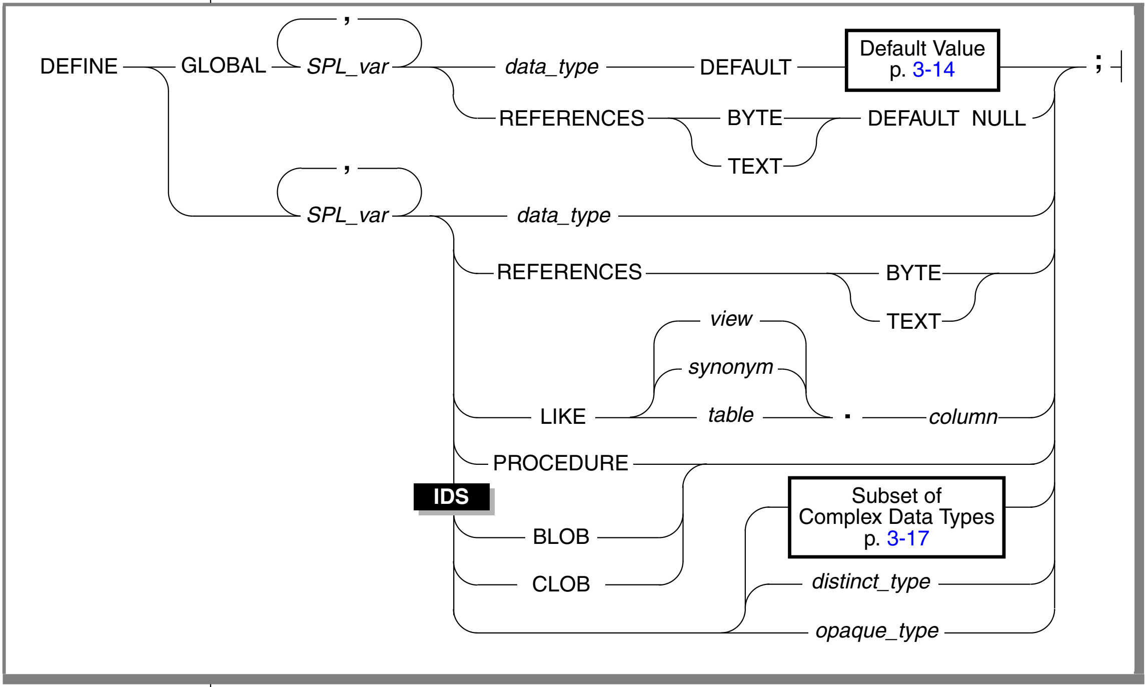
```
CREATE FUNCTION loop_skip()
   RETURNING INT;
   DEFINE i INT;
   ...
   FOR i IN (3 TO 15 STEP 2)
      INSERT INTO testtable values(i, null, null);
      IF i = 11
         CONTINUE FOR;
      END IF;
      RETURN i WITH RESUME;
   END FOR;

END FUNCTION;
```

Just as with EXIT (page 3-22), the FOR, WHILE, or FOREACH keyword must immediately follow CONTINUE to specify the type of loop. The CONTINUE statement generates errors if it cannot find the specified loop.

DEFINE

Use the DEFINE statement to declare local variables that an SPL routine uses, or to declare global variables that can be shared by several SPL routines.

Syntax



Element	Purpose	Restrictions	Syntax
column	Column name	Must already exist in the *table* or *view.*	Data type, p. 4-49
data_type	Type of *SPL_var*	See "Declaring Global Variables" on page 3-12.	Data type, p. 4-49
distinct_type	A distinct type	Must already be defined in the database.	Identifier, p. 4-189
opaque_type	An opaque type	Must already be defined in the database.	Identifier, p. 4-189
SPL_var	New SPL variable	Must be unique within statement block.	Identifier, p. 4-189
synonym, *table*, *view*	Name of a table, view, or synonym	Synonym and the table or view to which it points must already exist when the statement is issued	Database Object Name, p. 4-46

Usage

The DEFINE statement is not an executable statement. The DEFINE statement must appear after the routine header and before any other statements. If you define a local variable (by using DEFINE without the GLOBAL keyword), its scope of reference is the statement block in which it is defined. You can use the variable within the statement block. Another variable outside the statement block with a different definition can have the same name.

A variable with the GLOBAL keyword is global in scope and is available outside the statement block and to other SPL routines.Global variables can be any built-in data type except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB. Local variables can be any built-in data type except SERIAL, SERIAL8, TEXT, or BYTE. If *column* is of the SERIAL or SERIAL8 data type, declare an INT or INT8 variable (respectively) to store its value.

Referencing TEXT and BYTE Variables

The REFERENCES keyword lets you use BYTE and TEXT variables. These do not contain the actual data but are pointers to the data. The REFERENCES keyword indicates that the SPL variable is just a pointer. You can use BYTE and TEXT variables exactly as you would use any other variable in SPL.

Redeclaration or Redefinition

If you define the same variable twice in the same statement block, you receive an error. You can redefine a variable within a nested block, in which case it temporarily hides the outer declaration. This example produces an error:

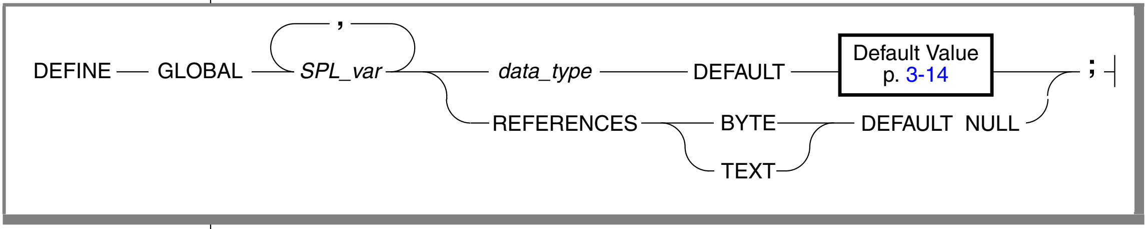
```
CREATE PROCEDURE example1()
   DEFINE n INT; DEFINE j INT;
   DEFINE n CHAR (1); -- redefinition produces an error
```

Redeclaration is valid in the following example. Within the nested statement block, **n** is a character variable. Outside the block, **n** is an integer variable.

```
CREATE PROCEDURE example2()
   DEFINE n INT; DEFINE j INT;
   ...
   BEGIN
   DEFINE n CHAR (1); -- character n masks global integer variable
   ...
END
```

Declaring Global Variables

Use the following syntax for declaring global variables:



Element	Purpose	Restrictions	Syntax
data_type	Type of *SPL_var*	See "Declaring Global Variables" on page 3-12.	Data type, p. 4-49
SPL_var	New SPL variable	Must be unique within statement block.	Identifier, p. 4-189

The GLOBAL keyword indicates that the variables that follow have a scope of reference that includes all SPL routines that run in a given DB-Access or SQL API session. The data types of these variables must match the data types of variables in the *global environment*. The global environment is the memory that is used by all the SPL routines that run in a given DB-Access or SQL API session. The values of global variables are stored in memory.

SPL routines that are running in the current session share global variables. Because the database server does not save global variables in the database, the global variables do not remain when the current session closes.

The first declaration of a global variable establishes the variable in the global environment; subsequent global declarations simply bind the variable to the global environment and establish the value of the variable at that point.

The following example shows two SPL procedures, **proc1** and **proc2**; each has defined the global variable **gl_out**:

- SPL procedure **proc1**

  ```
  CREATE PROCEDURE proc1()
     ...
     DEFINE GLOBAL gl_out INT DEFAULT 13;
     ...
     LET gl_out = gl_out + 1;
  END PROCEDURE;
  ```

- SPL procedure **proc2**

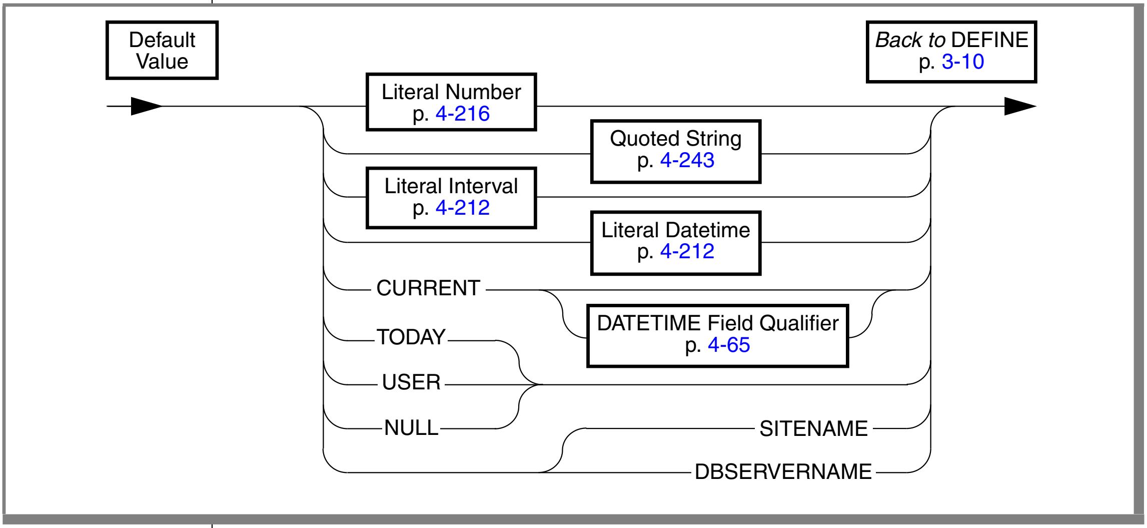
  ```
  CREATE PROCEDURE proc2()
     ...
     DEFINE GLOBAL gl_out INT DEFAULT 23;
     DEFINE tmp INT;
     ...
     LET tmp = gl_out
  END PROCEDURE;
  ```

If **proc1** is called first, **gl_out** is set to `13` and then incremented to `14`. If **proc2** is then called, it sees that **gl_out** is already defined, so the default value of `23` is not applied. Then, **proc2** assigns the existing value of `14` to **tmp**. If **proc2** had been called first, **gl_out** would have been set to `23`, and `23` would have been assigned to **tmp**. Later calls to **proc1** would not apply the default of `13`.

Databases do not share global variables. The database server and any application development tools do not share global variables.

Default Value

Global variables can have literal, NULL, or system constant default values.



If you specify a default value, the global variable is initialized with the specified value.

CURRENT

CURRENT is a valid default only for a DATETIME variable. If the YEAR TO FRACTION is its declared precision, no qualifier is needed. Otherwise, you must specify the same DATETIME qualifier when CURRENT is the default, as in the following example of a DATETIME variable:

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
      DEFAULT CURRENT YEAR TO MONTH;
```

USER

If you use the value that USER returns as the default, the variable must be defined as a CHAR, VARCHAR, NCHAR, or NVARCHAR data type. It is recommended that the length of the variable be at least 32 bytes. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the variable is too small to store the default value.

TODAY

If you use TODAY as the default, the variable must be a DATE value. (See "Constant Expressions" on page 4-95 for descriptions of TODAY and of the other system constants that can appear in the Default Value clause.)

BYTE and TEXT

The only default value valid for a BYTE or TEXT variable is NULL. The following example defines a TEXT global variable that is called **l_blob**:

```
CREATE PROCEDURE use_text()
   DEFINE i INT;
   DEFINE GLOBAL l_blob REFERENCES TEXT DEFAULT NULL;
   ...
END PROCEDURE
```

Here the REFERENCES keyword is required, because the DEFINE statement cannot declare a BYTE or TEXT data type directly; the **l_blob** variable is a pointer to a TEXT value that is stored in the global environment.

SITENAME or DBSERVERNAME

If you use the SITENAME or DBSERVERNAME keyword as the default, the variable must be a CHAR, VARCHAR, NCHAR, NVARCHAR, or LVARCHAR data type. Its default value is the name of the database server at runtime. It is recommended that the size of the variable be at least 128 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the variable is too small to store the default value.

The following example uses the SITENAME keyword to specify a default value. This example also initializes a global BYTE variable to NULL:

```
CREATE PROCEDURE gl_def()
   DEFINE GLOBAL gl_site CHAR(200) DEFAULT SITENAME;
   DEFINE GLOBAL gl_byte REFERENCES BYTE DEFAULT NULL;
   ...
END PROCEDURE
```

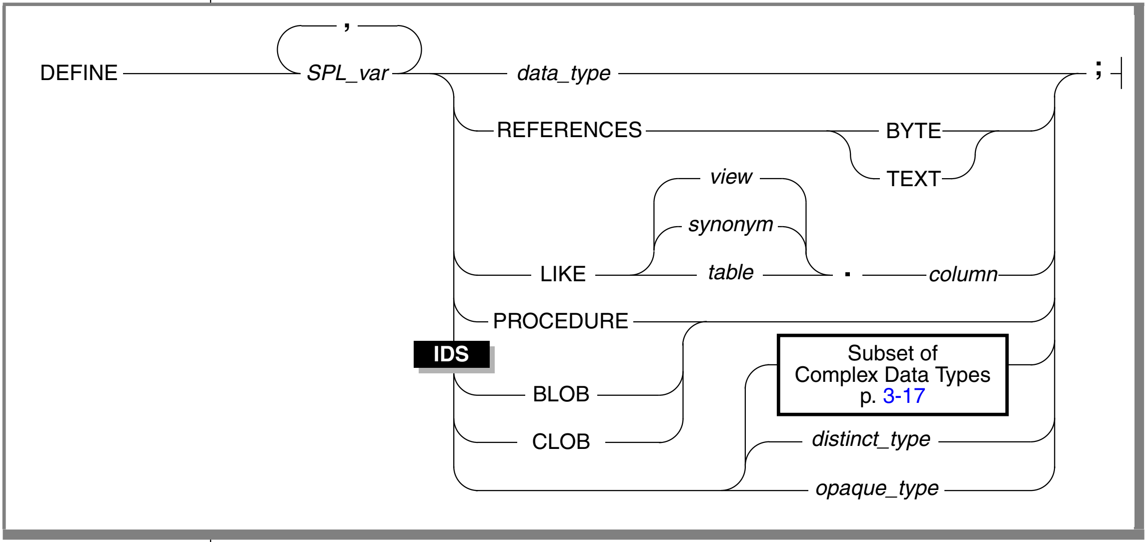
Declaring Local Variables

A *local variable* has as its scope of reference the routine in which it is declared. If you omit the GLOBAL keyword, any variables declared in the DEFINE statement are local variables, and are not visible in other SPL routines.

For this reason, different SPL routines that declare local variables of the same name can run without conflict in the same DB-Access or SQL API session.

If a local variable and a global variable have the same name, the global variable is not visible within the SPL routine where the local variable is declared. (In all other SPL routines, only the global variable is in scope.)

The following DEFINE statement syntax is for declaring local variables.



Element	Purpose	Restrictions	Syntax
column	Column name	Must already exist in the *table* or *view*	Identifier, p. 4-189
data_type	Type of *SPL_var*	Cannot be SERIAL, SERIAL8, TEXT, nor BYTE	Data type, p. 4-49
distinct_type	A distinct type	Must already be defined in the database	Identifier, p. 4-189
opaque_type	An opaque type	Must already be defined in the database.	Identifier, p. 4-189

(1 of 2)

Element	Purpose	Restrictions	Syntax
SPL_var	New SPL variable	Must be unique within statement block	Identifier, p. 4-189
synonym, table, view	Name of a table, view, or synonym	Synonym and the table or view to which it points must already exist when the statement is issued	Database Object Name, p. 4-46

(2 of 2)

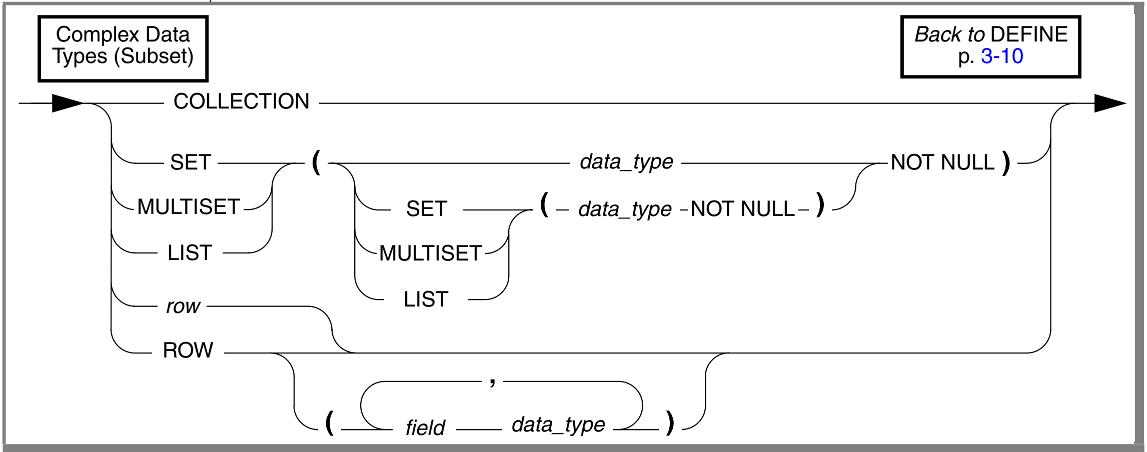
Local variables do not support default values. The following example shows some typical definitions of local variables:

```
CREATE PROCEDURE def_ex()
   DEFINE i INT;
   DEFINE word CHAR(15);
   DEFINE b_day DATE;
   DEFINE c_name LIKE customer.fname;
   DEFINE b_text REFERENCES TEXT;
END PROCEDURE
```

IDS

Subset of Complex Data Types

You can use the following syntax to declare an SPL variable as a typed or generic collection, or as a named, unnamed, or generic row data type.



Element	Purpose	Restrictions	Syntax
data_type	Type of elements of a collection or of fields of an unnamed row	Must match the data type of the values that the variable will store. Cannot be SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.	Identifier, p. 4-189
field	Field of unnamed row	Must exist in the database.	Identifier, p. 4-189
row	Named row data type	Must exist in the database.	Identifier, p. 4-189

IDS

Declaring Collection Variables

A local variable of type COLLECTION, SET, MULTISET, or LIST can hold a collection of values fetched from the database. You cannot define a collection variable as global (with the GLOBAL keyword) or with a default value.

A variable declared with the keyword COLLECTION is an untyped (or generic) collection variable that can hold a collection of any data type.

A variable defined with the type SET, MULTISET, or LIST is a typed collection variable. A typed collection variable can hold only a collection of its specified data type.

You must use the NOT NULL keywords when you define the elements of a typed collection variable, as in the following examples:

```
DEFINE a SET ( INT NOT NULL );

DEFINE b MULTISET ( ROW ( b1 INT,
                   b2 CHAR(50)
                 ) NOT NULL );

DEFINE c LIST( SET( INTEGER NOT NULL ) NOT NULL );
```

With variable **c**, both the INTEGER values in the SET and the SET values in the LIST are defined as NOT NULL.

You can define collection variables with nested complex types to hold matching nested complex type data. Any type or depth of nesting is allowed. You can nest row types within collection types, collection types within row types, collection types within collection types, row types within collection and row types, and so on.

If you declare a variable as COLLECTION type, the variable acquires varying data type declarations if it is reassigned within the same statement block, as in the following example:

```
DEFINE a COLLECTION;
LET a = setB;
...
LET a = listC;
```

In this example, **varA** is a generic collection variable that changes its data type to the data type of the currently assigned collection. The first LET statement makes **varA** a SET variable. The second LET statement makes **varA** a LIST variable.

IDS

Declaring Row Variables

Row variables hold data from named or unnamed row types. You can define a generic row variable, a named row variable, or an unnamed row variable.

A generic row variable, defined with the ROW keyword, can hold data from any row type. A named row variable holds data from the named row type specified in the variable definition.

The following statements show examples of generic row variables and named row variables:

```
DEFINE d ROW;              -- generic row variable

DEFINE rectv rectangle_t;  -- named row variable
```

A named row variable holds named row types of the same type in the variable definition.

To define a variable that will hold data stored in an unnamed row type, use the ROW keyword followed by the fields of the row type, as in:

```
DEFINE area ROW ( x int, y char(10) );
```

Unnamed row types are type-checked only by structural equivalence. Two unnamed row types are considered equivalent if they have the same number of fields, and if the fields have the same type definitions. Therefore, you could fetch either of the following row types into the variable **area** defined above:

```
ROW ( a int, b char(10) )
ROW ( area int, name char(10) )
```

Row variables can have fields, just as row types have fields. To assign a value to a field of a row variable, use the SQL dot notation *variableName.fieldName*, followed by an expression, as in the following example:

```
CREATE ROW TYPE rectangle_t (start point_t, length real,
   width real);

DEFINE r rectangle_t;
      -- Define a variable of a named row type
LET r.length = 45.5;
      -- Assign a value to a field of the variable
```

When you assign a value to a row variable, you can use any valid expression.

IDS

Declaring Opaque-Type Variables

Opaque-type variables hold data retrieved from opaque types, which you create with the CREATE OPAQUE TYPE statement. An opaque-type variable can only hold data of the opaque type on which it is defined. The following example defines a variable of the opaque type **point**, which holds the **x** and **y** coordinates of a two-dimensional point:

```
DEFINE b point;
```

Declaring Variables LIKE Columns

If you use the LIKE clause, the database server assigns the variable the same data type as a column in a table, synonym, or view.

The data types of variables that are defined as database columns are resolved at runtime; therefore, *column* and *table* do not need to exist at compile time.

Declaring a Variable LIKE a SERIAL Column

You can use the LIKE keyword to declare that a variable is like a SERIAL column. For example, if the column **serialcol** in the **mytab** table has the SERIAL data type, you can create the following SPL function:

```
CREATE FUNCTION func1()
DEFINE local_var LIKE mytab.serialcol;
RETURN;
END FUNCTION;
```

The variable **local_var** is treated as an INTEGER variable.

Declaring Variables as the PROCEDURE Type

The PROCEDURE keyword indicates that in the current scope, the variable is a call to a UDR.

IDS

The DEFINE statement does not have a FUNCTION keyword. Use the PROCEDURE keyword, whether you are calling a user-defined procedure or a user-defined function. ♦

Defining a variable of PROCEDURE type indicates that in the current statement scope, the variable is not a call to a built-in function. For example, the following statement defines **length** as an SPL routine, not as the built-in LENGTH function:

```
DEFINE length PROCEDURE;
...
LET x = length (a,b,c)
```

This definition disables the built-in LENGTH function within the scope of the statement block. You would use such a definition if you had already created a user-defined routine with the name **length**.

If you create an SPL routine with the same name as an aggregate function (SUM, MAX, MIN, AVG, COUNT) or with the name **extend**, you must qualify the routine name with the owner name.

Declaring Variables for BYTE and TEXT Data

The keyword REFERENCES indicates that the variable does not contain a BYTE or TEXT value but is a pointer to the BYTE or TEXT value. Use the variable as though it holds the data.

The following example defines a local BYTE variable:

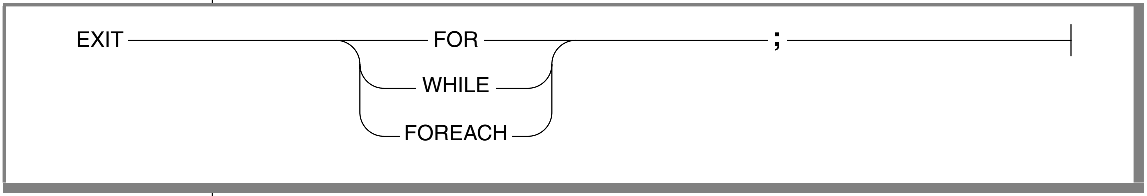
```
CREATE PROCEDURE use_byte()
   DEFINE i INT;
   DEFINE l_byte REFERENCES BYTE;
END PROCEDURE --use_byte
```

If you pass a variable of BYTE or TEXT data type to an SPL routine, the data is passed to the database server and stored in the root dbspace or dbspaces that the **DBSPACETEMP** environment variable specifies, if it is set. You do not need to know the location or name of the file that holds the data. BYTE or TEXT manipulation requires only the name of the BYTE or TEXT variable as it is defined in the routine.

EXIT

The EXIT statement terminates execution of a FOR, WHILE, or FOREACH loop.

Syntax



Usage

The EXIT statement marks the end of a FOR, WHILE, or FOREACH statement, causing the innermost loop of the specified type (FOR, WHILE, or FOREACH) to terminate. Execution resumes at the first statement outside the loop.

The FOR, WHILE, or FOREACH keyword must immediately follow EXIT. If the database server cannot find the specified loop, the EXIT statement fails. If EXIT is used outside any FOR, WHILE, or FOREACH loop, it generates errors.

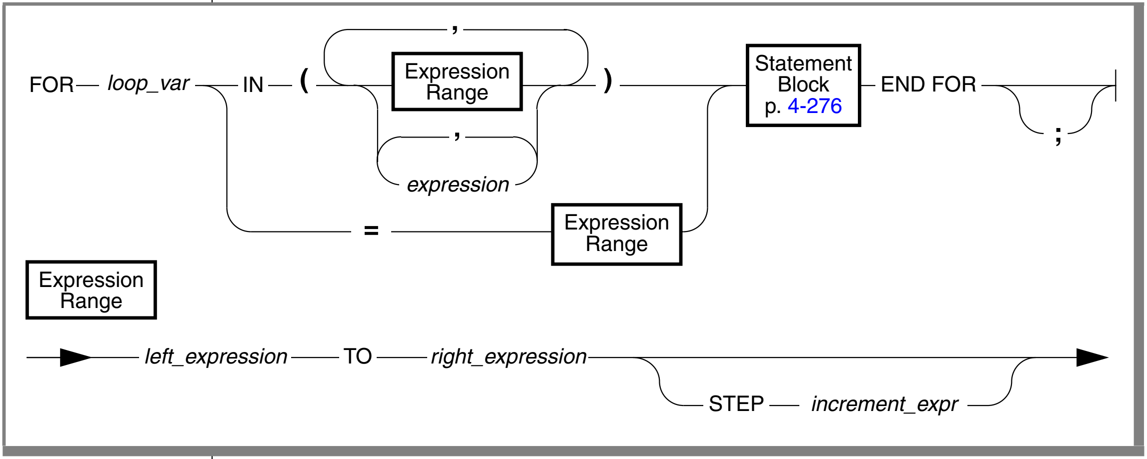
The following example uses an EXIT FOR statement. In the FOR loop, when `j` becomes `6`, the IF condition `i = 5` in the WHILE loop is true. The FOR loop stops executing, and the SPL procedure continues at the next statement outside the FOR loop (in this case, the END PROCEDURE statement). In this example, the procedure ends when `j` equals `6`:

```
CREATE PROCEDURE ex_cont_ex()
   DEFINE i,s,j, INT;
   FOR j = 1 TO 20
      IF j > 10 THEN
         CONTINUE FOR;
      END IF
      LET i,s = j,0;
      WHILE i > 0
         LET i = i -1;
         IF i = 5 THEN
            EXIT FOR;
         END IF
      END WHILE
   END FOR
END PROCEDURE
```

FOR

Use the FOR statement to initiate a controlled (definite) loop when you want to guarantee termination of the loop. The FOR statement uses expressions or range operators to specify a finite number of iterations for a loop.

Syntax



Element	Purpose	Restrictions	Syntax
expression	Value to compare with *loop_var*	Must match *loop_var* data type.	Expression, p. 4-67
increment_expr	Positive or negative value by which *loop_var* is incremented	Must return an integer. Cannot return `0`.	Expression, p. 4-67
left_expression	Starting expression of a range	Value must match SMALLINT or INT data type of *loop_var*.	Expression, p. 4-67
loop_var	Variable that determines how many times the loop executes	Must be defined and in scope within this statement block.	Identifier, p. 4-189
right_expression	Ending expression in the range	Same as for *left_expression*.	Expression, p. 4-67

Usage

The database server evaluates all expressions before the FOR statement executes. If one or more of the expressions are variables whose values change during the loop, the change has no effect on the iterations of the loop.

An error is issued, however, if an assignment within the body of the FOR statement attempts to modify the value of *loop_var*.

You can use the output from a SELECT statement as the *expression*.

The FOR loop terminates when *loop_var* is equal to the values of each element in the expression list or range in succession, or when it encounters an EXIT FOR statement.

The size of *right_expression* relative to *left_expression* determines if the range is stepped through by positive or negative increments.

Using the TO Keyword to Define a Range

The TO keyword implies a range operator. The range is defined by *left_expression* and *right_expression*, and the STEP *increment_expr* option implicitly sets the number of increments. If you use the TO keyword, *loop_var* must be an INT or SMALLINT data type.

The following example shows two equivalent FOR statements. Each uses the TO keyword to define a range. The first statement uses the IN keyword, and the second statement uses an equal sign (=). Each statement causes the loop to execute five times.

```
FOR index_var IN (12 TO 21 STEP 2)
   -- statement block
END FOR

FOR index_var = 12 TO 21 STEP 2
   -- statement block
END FOR
```

If you omit the STEP option, the database server gives *increment_expr* the value of `-1` if *right_expression* is less than *left_expression*, or `+1` if *right_expression* is more than *left_expression*. If *increment_expr* is specified, it must be negative if *right_expression* is less than *left_expression*, or positive if *right expression* is more than *left_expression*.

The two statements in the following example are equivalent. In the first statement, the STEP increment is explicit. In the second statement, the STEP increment is implicitly `1`.

```
FOR index IN (12 TO 21 STEP 1)
   -- statement block
END FOR

FOR index = 12 TO 21
   -- statement block
END FOR
```

The database server initializes the value of *loop_var* to the value of *left_expression*. In subsequent iterations, the server adds *increment_expr* to the value of *loop_var* and checks *increment_expr* to determine whether the value of *loop_var* is still between *left_expression* and *right_expression*. If so, the next iteration occurs. Otherwise, an exit from the loop takes place. Or, if you specify another range, the variable takes on the value of the first element in the next range.

Specifying Two or More Ranges in a Single FOR Statement

The following example shows a statement that traverses a loop forward and backward and uses different increment values for each direction:

```
FOR index_var IN (15 to 21 STEP 2, 21 to 15 STEP -3)
   -- statement body
END FOR
```

Using an Expression List as the Range

The database server initializes the value of *loop_var* to the value of the first expression specified. In subsequent iterations, *loop_var* takes on the value of the next expression. When the database server has evaluated the last expression in the list and used it, the loop stops.

The expressions in the IN list do not have to be numeric values, as long as you do not use range operators in the IN list. The following example uses a character expression list:

```
FOR c IN ('hello', (SELECT name FROM t), 'world', v1, v2)
   INSERT INTO t VALUES (c);
   END FOR
```

The following FOR statement shows the use of a numeric expression list:

```
FOR index IN (15,16,17,18,19,20,21)
   -- statement block
END FOR
```

Mixing Range and Expression Lists in the Same FOR Statement

If *loop_var* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture that uses an integer variable. Values in the expression list include the value that is returned from a SELECT statement, a sum of an integer variable and a constant, the values that are returned from an SPL function named **p_get_int**, and integer constants.

```
CREATE PROCEDURE for_ex ()
   DEFINE i, j INT;
   LET j = 10;
   FOR i IN (1 TO 20, (SELECT c1 FROM tab WHERE id = 1),
   j+20 to j-20, p_get_int(99),98,90 to 80 step -2)
      INSERT INTO tab VALUES (i);
   END FOR
END PROCEDURE
```

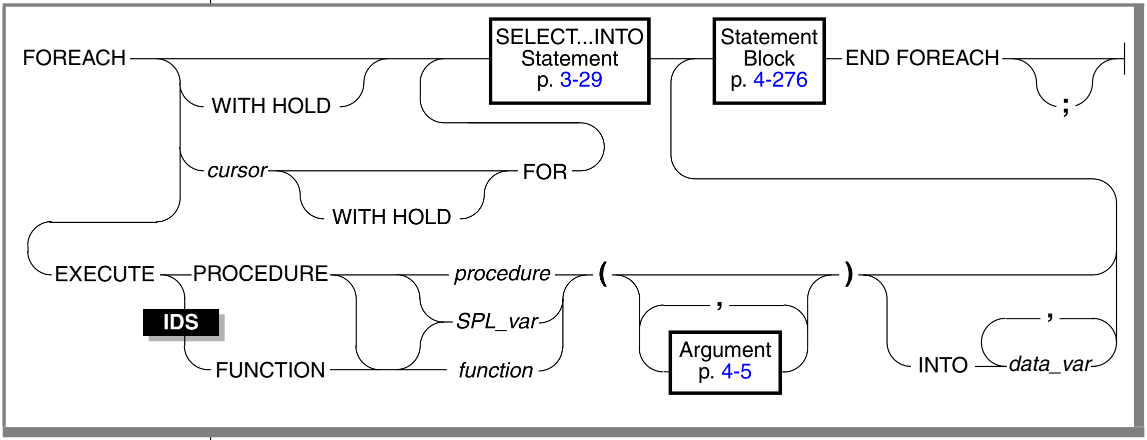
Related Information

Related statements: FOREACH, WHILE

FOREACH

Use a FOREACH loop to select and manipulate more than one row.

Syntax



Element	Purpose	Restrictions	Syntax
cursor	Identifier that you supply as a name for this FOREACH loop	Each cursor name within a routine must be unique.	Identifier, p. 4-189
data_var	SPL variable in the calling routine that receives the returned values	Data type of *data_var* must be appropriate for returned value.	Identifier, p. 4-189
function, procedure	SPL function or procedure to execute	Function or procedure must exist.	Database Object Name, p. 4-46
SPL_var	SPL variable that contains the name of a routine to execute	Must be a CHAR, VARCHAR, NCHAR, or NVARCHAR type.	Identifier, p. 4-189

Usage

A FOREACH loop is the procedural equivalent of using a cursor. To execute a FOREACH statement, the database server takes these actions:

1. It declares and implicitly opens a cursor.
2. It obtains the first row from the query contained within the FOREACH loop, or else the first set of values from the called routine.

3. It assigns to each variable in the variable list the value of the corresponding value from the active set that the SELECT statement or the called routine creates.
4. It executes the statement block.
5. It fetches the next row from the SELECT statement or called routine on each iteration, and it repeats steps 3 and 4.
6. It terminates the loop when it finds no more rows that satisfy the SELECT statement or called routine. It closes the implicit cursor when the loop terminates.

Because the statement block can contain additional FOREACH statements, cursors can be nested. No limit exists on the number of nested cursors.

An SPL routine that returns more than one row, collection element, or set of values is called a cursor function. An SPL routine that returns only one row or value is a noncursor function.

This SPL procedure illustrates FOREACH statements with a SELECT...INTO clause, with an explicitly named cursor, and with a procedure call:

```
CREATE PROCEDURE foreach_ex()
   DEFINE i, j INT;
   FOREACH SELECT c1 INTO i FROM tab ORDER BY 1
      INSERT INTO tab2 VALUES (i);
   END FOREACH
   FOREACH cur1 FOR SELECT c2, c3 INTO i, j FROM tab
      IF j > 100 THEN
         DELETE FROM tab WHERE CURRENT OF cur1;
         CONTINUE FOREACH;
      END IF
      UPDATE tab SET c2 = c2 + 10 WHERE CURRENT OF cur1;
   END FOREACH
   FOREACH EXECUTE PROCEDURE bar(10,20) INTO i
      INSERT INTO tab2 VALUES (i);
   END FOREACH
END PROCEDURE; -- foreach_ex
```

A select cursor is closed when any of the following situations occur:

- The cursor returns no further rows.
- The cursor is a select cursor without a HOLD specification, and a transaction completes using COMMIT or ROLLBACK statements.
- An EXIT statement executes, which transfers control out of the FOREACH statement.

- An exception occurs that is not trapped inside the body of the FOREACH statement. (See "ON EXCEPTION" on page 3-39.)
- A cursor in the calling routine that is executing this cursor routine (within a FOREACH loop) closes for any reason.

Using a SELECT...INTO Statement

As indicated in the diagram for "FOREACH" on page 3-27, not all clauses and options of the SELECT statement are available for you to use in a FOREACH statement. The SELECT statement in the FOREACH statement must include the INTO clause. It can also include UNION and ORDER BY clauses, but it cannot use the INTO TEMP clause. For a complete description of SELECT syntax and usage, see "SELECT" on page 2-581. The data type and count of each variable in the variable list must match each value that the SELECT...INTO statement returns.

Using Hold Cursors

The WITH HOLD keywords specify that the cursor should remain open when a transaction closes (by being committed or rolled back).

Updating or Deleting Rows Identified by Cursor Name

Specify a *cursor* name in the FOREACH statement if you intend to use the WHERE CURRENT OF *cursor* clause in UPDATE or DELETE statements that operate on the current row of *cursor* within the FOREACH loop. Although you cannot include the FOR UPDATE keywords in the SELECT ... INTO segment of the FOREACH statement, the cursor behaves like a FOR UPDATE cursor.

For a discussion of locking, see the section on "Locking with an Update Cursor" on page 2-330. For a discussion of isolation levels, see the description of "SET ISOLATION" on page 2-691.

IDS

Using Collection Variables

The FOREACH statement allows you to declare a cursor for an SPL collection variable. Such a cursor is called a *collection cursor*. Use a collection variable to access the elements of a collection (SET, MULTISET, LIST) column. Use a cursor when you want to access one or more elements in a collection variable.

Restrictions

When you use a collection cursor to fetch individual elements from a collection variable the FOREACH statement has the following restrictions:

- It cannot contain the WITH HOLD keywords.
- It must contain a restricted SELECT statement in the FOREACH loop.

In addition, the SELECT statement that you associate with the collection cursor has the following restrictions:

- Its general structure is SELECT… INTO… FROM TABLE. The statement selects one element at a time from a collection variable named after the TABLE keyword into another variable called an *element variable*.
- It cannot contain an expression in the select list.
- It cannot include the following clauses or options: WHERE, GROUP BY, ORDER BY, HAVING, INTO TEMP, and WITH REOPTIMIZATION.
- The data type of the element variable must be the same as the element type of the collection.
- The data type of the element variable can be any opaque, distinct, or collection data type, or any built-in data type except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.
- If the collection contains opaque, distinct, built-in, or collection types, the select list must be an asterisk (*).
- If the collection contains row types, the select list can be a list of one or more field names.

Examples

The following excerpt from an SPL routine shows how to fill a collection variable and then how to use a cursor to access individual elements:

```
DEFINE a SMALLINT;
DEFINE b SET(SMALLINT NOT NULL);
SELECT numbers INTO b FROM table1 WHERE id = 207;
FOREACH cursor1 FOR
   SELECT * INTO a FROM TABLE(b);
...
END FOREACH;
```

In this example, the SELECT statement selects one element at a time from the collection variable **b** into the element variable **a**. The select list is an asterisk, because the collection variable **b** contains a collection of built-in types. The variable **b** is used with the TABLE keyword as a Collection-Derived Table. For more information, see "Collection-Derived Table" on page 4-7.

The next example also shows how to fill a collection variable and then how to use a cursor to access individual elements. This example, however, uses a list of row-type fields in its select list.

```
DEFINE employees employee_t;
DEFINE n VARCHAR(30);
DEFINE s INTEGER;

SELECT emp_list into employees FROM dept_table
   WHERE dept_no = 1057;
FOREACH cursor1 FOR
   SELECT name,salary
      INTO n,s FROM TABLE( employees ) AS e;
...
END FOREACH;
```

Here the collection variable **employees** contains a collection of row types. Each row type contains the fields **name** and **salary**. The collection query selects one name and salary combination at a time, placing **name** into **n** and **salary** into **s**. The AS keyword declares **e** as an alias for the collection-derived table **employees**. The alias exists as long as the SELECT statement executes.

Modifying Elements in a Collection Variable

To update an element of a collection within an SPL routine, you must first declare a cursor with the FOREACH statement.

Then, within the FOREACH loop, select elements one at a time from the collection variable, using the collection variable as a collection-derived table in a SELECT query.

When the cursor is positioned on the element to be updated, you can use the WHERE CURRENT OF clause, as follows:

- The UPDATE statement with the WHERE CURRENT OF clause updates the value in the current element of the collection variable.
- The DELETE statement with the WHERE CURRENT OF clause deletes the current element from the collection variable.

Calling a UDR in the FOREACH Loop

In general, use these guidelines for calling another UDR from an SPL routine:

- To call a user-defined procedure, use EXECUTE PROCEDURE *procedure name*.
- To call a user-defined function, use EXECUTE FUNCTION *function name* (or EXECUTE PROCEDURE *function name* if the user-defined function was created with the CREATE PROCEDURE statement).

XPS

In Extended Parallel Server, you must use EXECUTE PROCEDURE. Extended Parallel Server does not support the EXECUTE FUNCTION statement. ♦

IDS

In Dynamic Server, if you use EXECUTE PROCEDURE, the database server looks first for a user-defined procedure of the name you specify. If it finds the procedure, the database server executes it. If it does not find the procedure, it looks for a user-defined function of the same name to execute. If the database server finds neither a function nor a procedure, it issues an error message.If you use EXECUTE FUNCTION, the database server looks for a user-defined function of the name you specify. If it does not find a function of that name, the database server issues an error message. ♦

A called SPL function can return zero (0) or more values or rows.

The data type and count of each variable in the variable list must match each value that the function returns.

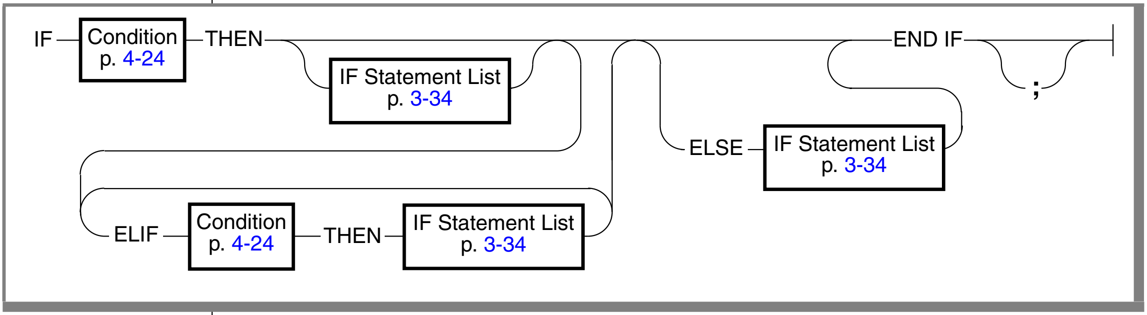
Related Information

Related statements: FOR, WHILE

IF

Use an IF statement to create a branch within an SPL routine.

Syntax


IF
Condition p. 4-24
THEN
IF Statement List p. 3-34
END IF
;
ELSE
IF Statement List p. 3-34
ELIF
Condition p. 4-24
THEN
IF Statement List p. 3-34

Usage

The condition that the IF clause states is evaluated. If the result is true, the statements that follow the THEN keyword execute. If the result is false, and an ELIF clause exists, the statements that follow the ELIF clause execute. If no ELIF clause exists, or if the condition in the ELIF clause is not true, the statements that follow the ELIF keyword execute.

ELIF Clause

Use the ELIF clause to specify one or more additional conditions to evaluate. If the IF condition is false, the ELIF condition is evaluated. If the ELIF condition is true, the statements that follow the ELIF clause execute.

ELSE Clause

The ELSE clause executes if no true previous condition exists in the IF clause or any of the ELIF clauses.

In the following example, the SPL function uses an IF statement with both an ELIF clause and an ELSE clause. The IF statement compares two strings.

The function displays a `1` to indicate that the first string comes before the second string alphabetically, or a `-1` if the first string comes after the second string alphabetically. If the strings are the same, a zero (0) is returned.

```
CREATE FUNCTION str_compare (str1 CHAR(20), str2 CHAR(20))
   RETURNING INT;
   DEFINE result INT;
   IF str1 > str2 THEN
      LET result =1;
   ELIF str2 > str1 THEN
      LET result = -1;
   ELSE
      LET result = 0;
   END IF
   RETURN result;
END FUNCTION -- str_compare
```

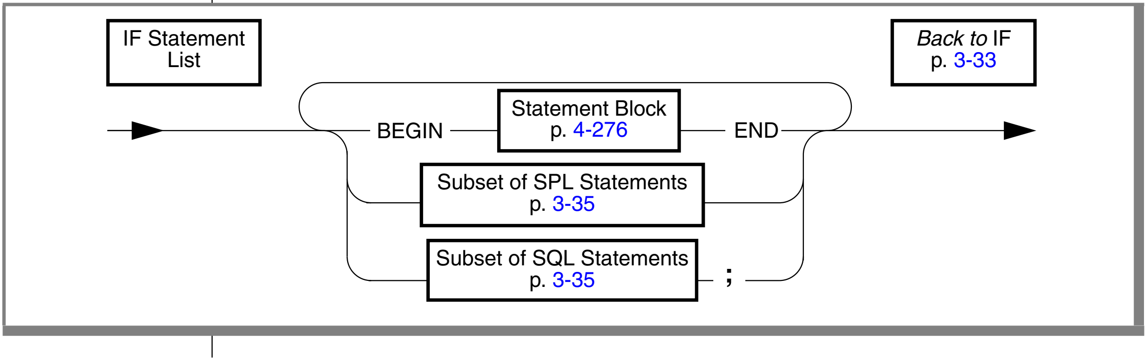
Conditions in an IF Statement

Conditions in an IF statement are evaluated in the same way as conditions in a WHILE statement. If any expression in the condition evaluates to NULL, the condition returns false. Consider the following points:

1. If the expression *x* evaluates to NULL, then *x* is not true by definition. Furthermore, `NOT(x)` is also *not* true.
2. IS NULL is the sole operator that can yield true for *x*. That is, *x* IS NULL is true, and *x* IS NOT NULL is not true.

If an expression in the condition has an UNKNOWN value (due to the use of an uninitialized variable), the statement terminates and raises an exception.

IF Statement List



Subset of SPL Statements Allowed in the IF Statement List

You can use any of the following SPL statements in the IF statement list.

- CALL
- CONTINUE
- EXIT
- FOR
- FOREACH
- IF
- LET
- RAISE EXCEPTION
- RETURN
- SYSTEM
- TRACE
- WHILE

The "Subset of SPL Statements" syntax diagram for the "IF Statement List" refers to the SPL statements that are listed in the preceding table.

SQL Statements Not Allowed in an IF Statement

The "Subset of SQL Statements" element in the syntax diagram for the "IF Statement List" refers to all SQL statements, except for the following SQL statements, which are not valid in the IF statement list.

- ALLOCATE DESCRIPTOR
- CLOSE DATABASE
- CONNECT
- CREATE DATABASE
- CREATE PROCEDURE
- DATABASE
- DEALLOCATE DESCRIPTOR
- DECLARE
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- FLUSH
- LOAD
- OPEN
- OUTPUT
- PREPARE
- PUT
- SET CONNECTIPON
- SET DESCRIPTOR
- UNLOAD
- WHENEVER
- FREE
- GET DESCRIPTOR
- GET DIAGNOSTICS
- INFO

Many of these statements are prohibited by the more general rule that the dynamic management statements of SQL are not valid within an SPL routine.

You can use a SELECT statement only if you use the INTO TEMP clause to store the result set of the SELECT statement in a temporary table.

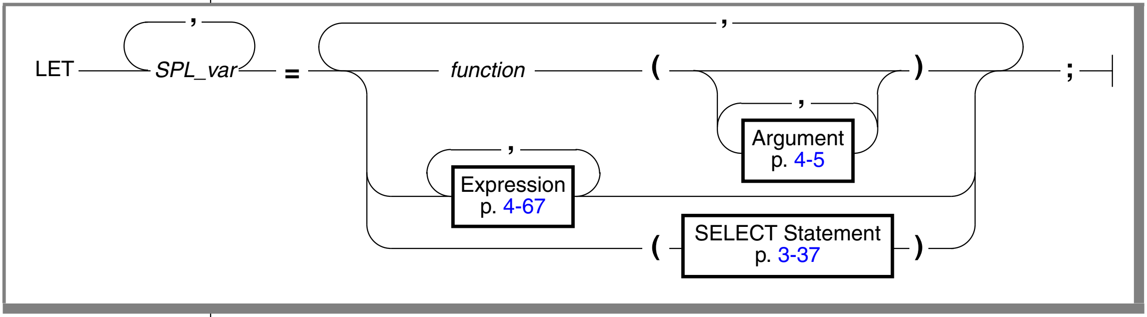
Related Information

Related statements: CASE, WHILE

LET

Use the LET statement to assign values to variables or to call a user-defined SPL routine and assign the returned value or values to SPL variables.

Syntax



Element	Purpose	Restrictions	Syntax
function	SPL function to be invoked	Must exist in the database.	Database Object Name, p. 4-46
SPL_var	SPL variable to receive a value that the *function*, expression, or query returns	Must be defined and in scope within the statement block.	Identifier, p. 4-189

Usage

The LET statement can assign a value returned by an expression, function, or query to an SPL variable. At runtime, the value to be assigned is calculated first. The resulting value is cast to the data type of *SPL_var*, if possible, and the assignment occurs. If conversion is not possible, an error occurs, and the value of the variable is undefined. (A LET operation that assigns a single value to a single SPL variable is called a *simple assignment*.)

A *compound assignment* assigns multiple expressions to multiple SPL variables.The data types of expressions in the expression list do not need to match the data types of the corresponding variables in the variable list, because the database server automatically converts the data types. (For a detailed discussion of casting, see the *IBM Informix Guide to SQL: Reference*.)

In multiple-assignment operations, the number of variables to the left of the " = " symbol must match the number of values returned by the functions, expressions, and queries listed on the right of the " = " symbol. The following example shows several LET statements that assign values to SPL variables:

```
LET a   = c + d ;
LET a,b = c,d ;
LET expire_dt = end_dt + 7 UNITS DAY;
LET name = 'Brunhilda';
LET sname = DBSERVERNAME;
LET this_day = TODAY;
```

You cannot use multiple values to the right of the " = " symbol to operate on other values. For example, the following statement is not valid:

```
LET a,b = (c,d) + (10,15); -- INVALID EXPRESSION
```

Using a SELECT Statement in a LET Statement

The examples in this section use a SELECT statement in a LET statement. You can use a SELECT statement to assign values to one or more variables on the left side of the equals (=) operator, as the following example shows:

```
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
LET a,b,c = (SELECT c1,c2 FROM t WHERE id = 1), 15;
```

You cannot use a SELECT statement to make multiple values operate on other values. The following example is invalid:

```
LET a,b = (SELECT c1,c2 FROM t) + (10,15); -- ILLEGAL CODE
```

Because a LET statement is equivalent to a SELECT...INTO statement, the two statements in the following example have the same results: `a=c` and `b=d`:

```
CREATE PROCEDURE proof()
   DEFINE a, b, c, d INT;
   LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
   SELECT c1, c2 INTO c, d FROM t WHERE id = 1
END PROCEDURE
```

If the SELECT statement returns more than one row, you must enclose the SELECT statement in a FOREACH loop.

For a complete description of SELECT syntax and usage, see "SELECT" on page 2-581.

Calling a Function in a LET Statement

You can call a user-defined function in a LET statement and assign the return values to an SPL variable. The SPL variable receives the returned values from the called function.

An SPL function can return multiple values (that is, values from multiple columns in the same row) into a list of variable names. In other words, the function can have multiple values in its RETURN statement and the LET statement can have multiple variables to receive the returned values.

When you call the function, you must specify all the necessary arguments to the function unless the arguments of the function have default values. If you name one of the parameters in the called function, with syntax such as **name = 'smith'**, you must name all of the parameters.

An SPL function that selects and returns more than one row must be enclosed in a FOREACH loop.

The following two examples show valid LET statements:

```
LET a, b, c = func1(name = 'grok', age = 17);
LET a, b, c = 7, func2('orange', 'green');
```

The following LET statement is not valid because it tries to add the output of two functions and then assign the sum to two variables, *a* and *b*. You can easily split this LET statement into two valid LET statements.

```
LET a, b = func1() + func2(); -- ILLEGAL CODE
```

A function called in a LET statement can have an argument of COLLECTION, SET, MULTISET, or LIST. You can assign the value that the function returns to a variable, for example:

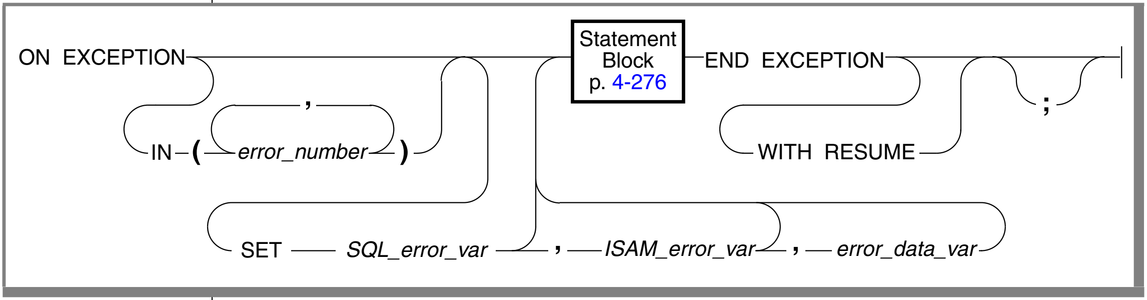
```
LET d = function1(collection1);
LET a = function2(set1);
```

In the first statement, the SPL function **function1** accepts **collection1** (that is, any collection data type) as an argument and returns its value to the variable **d**. In the second statement, the SPL function **function2** accepts **set1** as an argument and returns a value to the variable **a**.

ON EXCEPTION

Use the ON EXCEPTION statement to specify the actions to be taken for a specified error or a list of errors during execution of a statement block.

Syntax



Element	Purpose	Restrictions	Syntax
error_data_var	SPL variable to receive a string returned by an SQL error or by a user-defined exception	Must be a character type to receive the error information. Must be valid in the current statement block.	Identifier, p. 4-189
error_number	SQL error number or an error number created by a RAISE EXCEPTION statement that is to be trapped	Must be of integer type. Must be valid in the current statement block.	Literal Number, p. 4-216
ISAM_error_var	SPL variable that receives the ISAM error number of the exception raised	Must be of integer type. Must be valid in the current statement block.	Identifier, p. 4-189
SQL_error_var	SPL variable that receives the SQL error number of the exception raised	Must be a character type. Must be valid in the current statement block.	Identifier, p. 4-189

Usage

The ON EXCEPTION statement, together with the RAISE EXCEPTION statement, provides an error-trapping and error-recovery mechanism for SPL. ON EXCEPTION specifies a list of errors that you want to trap as the SPL routine executes, and specifies the action to take when the error occurs within the statement block. If the IN clause is omitted, then all errors are trapped.

A statement block can include more than one ON EXCEPTION statement. The exceptions that are trapped can be either system-defined or user-defined.

The scope of an ON EXCEPTION statement is the statement block that follows the ON EXCEPTION statement and all the statement blocks that are nested within that following statement block.

When an exception is trapped, the error status is cleared.

If you specify a variable to receive an ISAM error, but no accompanying ISAM error exists, a zero (0) is assigned to the variable. If you specify a variable to receive the error text, but none exists, the variable stores an empty string.

ON EXCEPTION has no effect within a UDR that is called by a trigger.

Placement of the ON EXCEPTION Statement

The ON EXCEPTION statement is a declarative statement, not an executable statement. For this reason, ON EXCEPTION must precede any executable statement and must follow any DEFINE statement within an SPL routine.

The following example shows the correct placement of an ON EXCEPTION statement. Function **add_salesperson( )** inserts a set of values into a table. If the table does not exist, it is created, and the values are inserted. The function also returns the total number of rows in the table after the insert occurs.

```
CREATE FUNCTION add_salesperson(last CHAR(15), first CHAR(15))
   RETURNING INT;
   DEFINE x INT;
   ON EXCEPTION IN (-206) -- If no table was found, create one
      CREATE TABLE emp_list
          (lname CHAR(15),fname CHAR(15), tele CHAR(12));
      INSERT INTO emp_list VALUES -- and insert values
          (last, first, '800-555-1234');
   END EXCEPTION WITH RESUME
   INSERT INTO emp_list VALUES (last, first, '800-555-1234')
   LET x = SELECT count(*) FROM emp_list;
   RETURN x;
END FUNCTION
```

When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement, which traps the specified error code. The ON EXCEPTION statement can have the error number in the IN clause or have no IN clause. If the database server finds no pertinent ON EXCEPTION statement, the error code passes back to the caller (the SPL routine, application, or interactive user), and execution aborts.

In the previous example, the minus (-) symbol is required in the IN clause that specifies error -206; most error codes are negative integers.

The following example uses two ON EXCEPTION statements with the same error number so that error code 691 can be trapped in two levels of nesting:

```
CREATE PROCEDURE delete_cust (cnum INT)
   ON EXCEPTION  IN (-691)    -- children exist
      BEGIN -- Begin-end so no other DELETEs get caught in here.
          ON EXCEPTION IN (-691)
              DELETE FROM another_child WHERE num = cnum;
            DELETE FROM orders WHERE customer_num = cnum;
         END EXCEPTION -- for 691
      DELETE FROM orders WHERE customer_num = cnum;
      END
DELETE FROM cust_calls WHERE customer_num = cnum;
      DELETE FROM customer WHERE customer_num = cnum;
    END EXCEPTION
       DELETE FROM customer WHERE customer_num = cnum;
END PROCEDURE
```

Using the IN Clause to Trap Specific Exceptions

An error is trapped if the SQL error code or the ISAM error code matches an exception code in the list of error numbers. The search through the list of errors begins from the left and stops with the first match. You can use a combination of an ON EXCEPTION statement without an IN clause and one or more ON EXCEPTION statements with an IN clause. When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement that traps the particular error code.

```
CREATE PROCEDURE ex_test ()
   DEFINE error_num INT;
   ...
   ON EXCEPTION SET error_num
   -- action C
   END EXCEPTION
   ON EXCEPTION IN (-300)
   -- action B
   END EXCEPTION
   ON EXCEPTION IN (-210, -211, -212) SET error_num
   -- action A
   END EXCEPTION
```

A summary of the sequence of statements in the previous example would be:

1. Test for an error.
2. If error `-210`, `-211`, or `-212` occurs, take action `A`.
3. If error `-300` occurs, take action `B`.
4. If any other error occurs, take action `C`.

Receiving Error Information in the SET Clause

If you use the SET clause, when an exception occurs, the SQL error code and (optionally) the ISAM error code are inserted into the variables that are specified in the SET clause. If you provide an *error_data_var*, any error text that the database server returns is put into the *error_data_var*. Error text includes information such as the offending table or column name.

Forcing Continuation of the Routine

The first example in "Placement of the ON EXCEPTION Statement" on page 3-40 uses the WITH RESUME keyword to indicate that after the statement block in the ON EXCEPTION statement executes, execution is to continue at the `LET x = SELECT COUNT(*) FROM emp_list` statement, which is the line following the line that raised the error. For this function, the result is that the count of salespeople names occurs even if the error occurred.

Continuing Execution After an Exception Occurs

If you omit the WITH RESUME keywords, the next statement that executes after an exception occurs depends on the placement of the ON EXCEPTION statement, as the following scenarios describe:

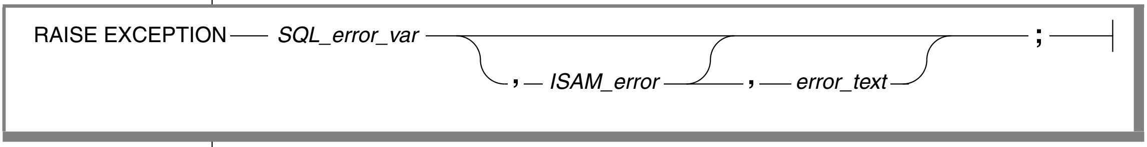
- If the ON EXCEPTION statement is inside a statement block with a BEGIN and an END keyword, execution resumes with the first statement (if any) after that BEGIN...END block. That is, it resumes after the scope of the ON EXCEPTION statement.
- If the ON EXCEPTION statement is inside a loop (FOR, WHILE, FOREACH), the rest of the loop is skipped, and execution resumes with the next iteration of the loop.
- If no statement or block, but only the SPL routine, contains the ON EXCEPTION statement, the routine executes a RETURN statement with no arguments, returning a successful status and no values.

To prevent an infinite loop, if an error occurs during execution of the statement block, then the search for another ON EXCEPTION statement to trap the error does not include the current ON EXCEPTION statement.

RAISE EXCEPTION

Use the RAISE EXCEPTION statement to simulate the generation of an error.

Syntax



Element	Purpose	Restrictions	Syntax
error_text	SPL variable or expression that contains the error message text	Must be a character data type and be valid in the statement block.	Identifier, p. 4-189; Expression, p. 4-67
ISAM_error	SPL variable or expression that represents an ISAM error number Default is `0`.	Must return a SMALLINT value. You can place a unary minus sign before the error number.	Identifier, p. 4-189; Expression, p. 4-67
SQL_error	SPL variable or expression that represents an SQL error number	Same as for *ISAM_error*	Identifier, p. 4-189; Expression, p. 4-67

Usage

Use the RAISE EXCEPTION statement to simulate an error or to generate an error with a custom message. An ON EXCEPTION statement can trap the generated error.

If you omit *ISAM_error*, the database server sets the ISAM error code to zero (`0`) when the exception is raised. If you want to specify *error_text* but not specify a value for *ISAM_error*, specify zero (`0`) as the value of *ISAM_error*.

The RAISE EXCEPTION statement can raise either system-generated exceptions or user-generated exceptions. For example, the following statement raises the error number `-208` and inserts the text `a missing file` into the variable of the system-generated error message:

```
RAISE EXCEPTION -208, 0, 'a missing file';
```

Here the minus (-) symbol is required after the EXCEPTION keyword for error -208; most error codes are negative integers.

Special Error Numbers

The special error number `-746` allows you to produce a customized message. For example, the following statement raises the error number `-746` and returns the quoted text:

```
RAISE EXCEPTION -746, 0, 'You broke the rules';
```

In the following example, a negative value for `alpha` raises exception `-746` and provides a specific message that describes the problem. The code should contain an ON EXCEPTION statement that traps for an exception of `-746`.

```
FOREACH SELECT c1 INTO alpha FROM sometable
IF alpha < 0 THEN
RAISE EXCEPTION -746, 0, 'a < 0 found' -- emergency exit
END IF
END FOREACH
```

When the SPL routine executes and the IF condition is met, the database server returns the following error:

```
-746: a < 0 found.
```

For more information about the scope and compatibility of exceptions, see "ON EXCEPTION" on page 3-39.

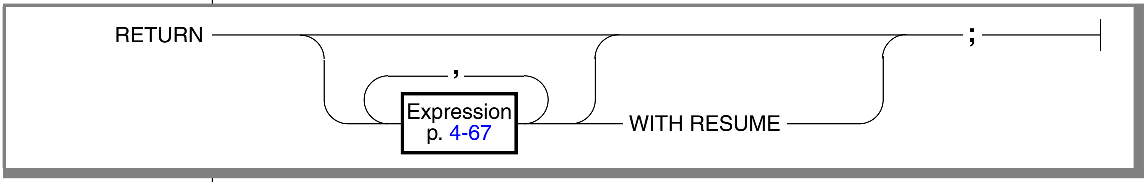
Related Information

Related statement: ON EXCEPTION

RETURN

Use the RETURN statement to specify what values (if any) the SPL function returns to the calling module.

Syntax



Usage

IDS

In Dynamic Server, for backward compatibility, you can use this statement inside a CREATE PROCEDURE statement to create an SPL function. However, only use this statement inside the CREATE FUNCTION statement to create new user-defined functions. ♦

All the RETURN statements in the SPL function must be consistent with the RETURNING clause of the CREATE FUNCTION (or CREATE PROCEDURE) statement, which defines the function.

The number and data type of values in the RETURN statement, if any, must match in number and data type the data types that are listed in the RETURNING clause of the CREATE FUNCTION (or CREATE PROCEDURE) statement. You can choose to return no values even if you specify one or more values in the RETURNING clause. If you use a RETURN statement without any expressions, but the calling UDR or program expects one or more returned values, it is equivalent to returning the expected number of NULL values to the calling program.

A RETURN statement without any expressions exits only if the SPL function is declared as not returning any values; otherwise it returns NULL values.

IDS

In an SPL program, you can use an external function as an expression in a RETURN statement provided that the external function is not an iterator function. An *iterator function* is an external function that returns one or more rows of data and therefore requires a cursor to execute. ♦

This SPL function includes two valid RETURN statements. A program that calls this function should check if no values are returned and act accordingly.

```
CREATE FUNCTION two_returns (stockno INT)
   RETURNING CHAR (15);
   DEFINE des CHAR(15);
   ON EXCEPTION (-272) -- if user doesn't have select privilege...
      RETURN;      -- return no values.
   END EXCEPTION;
   SELECT DISTINCT descript INTO des FROM stock
         WHERE stocknum = stockno;
   RETURN des;
END FUNCTION
```

WITH RESUME Keyword

If you use the WITH RESUME keyword, after the RETURN statement executes, the next invocation of the SPL function (upon the next FETCH or FOREACH statement) starts from the statement that follows the RETURN statement. Any function that executes a RETURN WITH RESUME statement must be invoked within a FOREACH loop, or else in the FROM clause of a query.

E/C

If an SPL routine executes a RETURN WITH RESUME statement, a FETCH statement in an ESQL/C application can call the SPL routine. ♦

The following example shows a cursor function that another UDR can call. After the RETURN WITH RESUME statement returns each value to the calling UDR or program, the next line of **series** executes the next time **series** is called. If the variable **backwards** equals zero (0), no value is returned to the calling UDR or program, and execution of **series** stops.

```
CREATE FUNCTION series (limit INT, backwards INT)
   RETURNING INT;
   DEFINE i INT;
   FOR i IN (1 TO limit)
      RETURN i WITH RESUME;
   END FOR
   IF backwards = 0 THEN
      RETURN;
   END IF
   FOR i IN (limit TO 1 STEP -1)
      RETURN i WITH RESUME;
   END FOR
END FUNCTION -- series
```

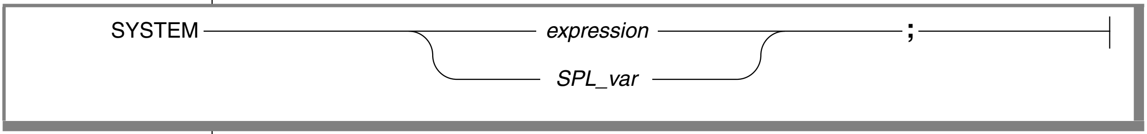
IDS

SPL iterator functions must include the RETURN WITH RESUME statement. For information about using an iterator function with a virtual table interface in the FROM clause of a query, see "Iterator Functions" on page 2-603. ♦

SYSTEM

Use the SYSTEM statement to issue an operating-system command from within an SPL routine.

Syntax



Element	Purpose	Restrictions	Syntax
expression	Evaluates to a user-executable operating-system command	You cannot specify that the command run in the background.	Operating-system dependent
SPL_var	SPL variable that contains a command	Must be of a character data type.	Identifier, p. 4-189

Usage

If the specified *expression* is not a character expression, it is converted to a character expression and passed to the operating system for execution.

The command that SYSTEM specifies cannot run in the background. The database server waits for the operating system to complete execution of the command before it continues to the next statement in the SPL routine. The SPL routine cannot use any returned value(s) from the command.

If the operating-system command fails (that is, returns a nonzero status for the command), an exception is raised that contains the returned operating-system status as the ISAM error code and an appropriate SQL error code.

A rollback does not terminate a system call, so a suspended transaction can wait indefinitely for the call to return. For instructions on recovery from a deadlock during a long transaction rollback, see the *Administrator's Guide*.

IDS

The dynamic log feature of Dynamic Server automatically adds log files until the long transaction completes or rolls back successfully. ♦

In DBA- and owner-privileged SPL routines that contain SYSTEM statements, the command runs with the permissions of the user who executes the routine.

UNIX

Examples of the SYSTEM Statement on UNIX

The SYSTEM statement in the following example of an SPL routine causes the UNIX operating system to send a mail message to the system administrator:

```
CREATE PROCEDURE sensitive_update()
   ...
   LET mailcall = 'mail headhoncho < alert';
   -- code to execute if user tries to execute a specified
   -- command, then sends email to system administrator
   SYSTEM mailcall;
   ...
END PROCEDURE; -- sensitive_update
```

You can use a double-pipe symbol (||) to concatenate expressions with a SYSTEM statement, as the following example shows:

```
CREATE PROCEDURE sensitive_update2()
   DEFINE user1 char(15);
   DEFINE user2 char(15);
   LET user1 = 'joe';
   LET user2 = 'mary';
   ...
   -- code to execute if user tries to execute a specified
   -- command, then sends email to system administrator
   SYSTEM 'mail -s violation' ||user1 || ' ' || user2
                 || '< violation_file';
   ...
END PROCEDURE; --sensitive_update2
```

Windows

Example of the SYSTEM Statement on Windows

The first SYSTEM statement in the following example of an SPL routine causes Windows to send an error message to a temporary file and to put the message in a system log that is sorted alphabetically. The second SYSTEM statement causes the operating system to delete the temporary file.

```
CREATE PROCEDURE test_proc()
   ...
   SYSTEM 'type errormess101 > %tmp%tmpfile.txt |
         sort >> %SystemRoot%systemlog.txt';
   SYSTEM 'del %tmp%tmpfile.txt';
   ...
END PROCEDURE; --test_proc
```

The expressions that follow the SYSTEM statements in this example contain variables **%tmp%** and **%SystemRoot%** that are defined by Windows.

Setting Environment Variables in SYSTEM Commands

When the operating-system command that SYSTEM specifies is executed, no guarantee exists that any environment variables that the user application sets are passed to the operating system. If you set an environment variable in a SYSTEM command, the setting is only valid during that SYSTEM command.

To avoid this potential problem, the following method is recommended to ensure that any environment variables that the user application requires are carried forward to the operating system.

To Change Environment Settings for an Operating System Command

1. Create a shell script (on UNIX systems) or a batch file (on Windows platforms) that sets up the desired environment and then executes the operating system command.
2. Use the SYSTEM command to execute the shell script or batch file.

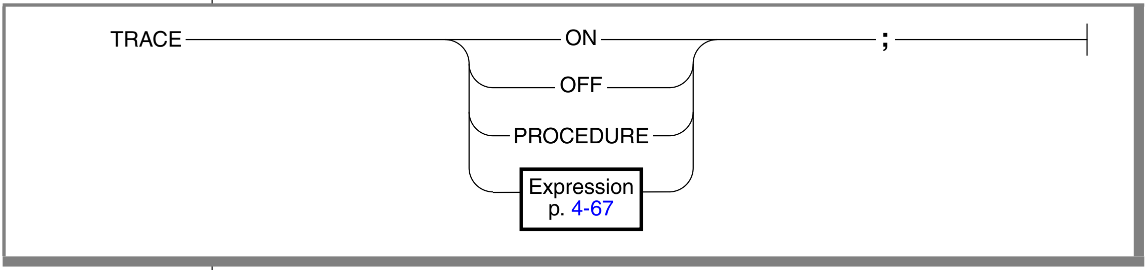
This solution has an additional advantage, in that if you subsequently need to change the environment, you can modify the shell script or the batch file without needing to recompile the SPL routine.

For information about operating system commands that set environment variables, see the *IBM Informix Guide to SQL: Reference*.

TRACE

Use the TRACE statement to control the generation of debugging output.

Syntax



Usage

The TRACE statement generates output that is sent to the file that the SET DEBUG FILE TO statement specifies. Tracing writes to the debug file the current values of the following program objects:

- SPL variables
- Routine arguments
- Return values
- SQL error codes
- ISAM error codes

The output of each executed TRACE statement appears on a separate line.

If you use the TRACE statement before you specify a DEBUG file to contain the output, an error is generated.

Any routine that the SPL routine calls inherits the trace state. That is, a called routine (on the same database server) assumes the same trace state (ON, OFF, or PROCEDURE) as the calling routine. The called routine can set its own trace state, but that state is not passed back to the calling routine.

A routine that is executed on a remote database server does not inherit the trace state.

TRACE ON

If you specify the keyword ON, all statements are traced. The values of variables (in expressions or otherwise) are printed before they are used. To turn tracing ON implies tracing both routine calls and statements in the body of the routine.

TRACE OFF

If you specify the keyword OFF, all tracing is turned off.

TRACE PROCEDURE

If you specify the keyword PROCEDURE, only the routine calls and return values, but not the body of the routine, are traced.

The TRACE statement does not have ROUTINE or FUNCTION keywords. Therefore, use the TRACE PROCEDURE keywords even if the SPL routine you want to trace is a function.

Displaying Expressions

You can use the TRACE statement with a quoted string or an expression to display values or comments in the output file. If the expression is not a literal expression, the expression is evaluated before it is written to the output file.

You can use the TRACE statement with an expression even if you used a TRACE OFF statement earlier in a routine. You must first, however, use the SET DEBUG statement to establish a trace output file.

The next example uses a TRACE statement with an expression after using a TRACE OFF statement. The example uses UNIX file naming conventions.

```
CREATE PROCEDURE tracing ()
   DEFINE i INT;
BEGIN
   ON EXCEPTION IN (1)
   END EXCEPTION; -- do nothing
   SET DEBUG FILE TO '/tmp/foo.trace';
   TRACE OFF;
   TRACE 'Forloop starts';

   FOR i IN (1 TO 1000)
      BEGIN
         TRACE 'FOREACH starts';
         FOREACH SELECT...INTO a FROM t
            IF <some condition> THEN
               RAISE EXCEPTION 1    -- emergency exit
            END IF
         END FOREACH
         -- return some value
      END
   END FOR
   -- do something
END;
END PROCEDURE
```

Example Showing Different Forms of TRACE

The following example shows several different forms of the TRACE statement. The example uses Windows file naming conventions.

```
CREATE PROCEDURE testproc()
   DEFINE i INT;

   SET DEBUG FILE TO 'C:\tmp\test.trace';
   TRACE OFF;
   TRACE 'Entering foo';

   TRACE PROCEDURE;
   LET i = test2();

   TRACE ON;
   LET i = i + 1;

   TRACE OFF;
   TRACE 'i+1 = ' || i+1;
   TRACE 'Exiting testproc';

   SET DEBUG FILE TO 'C:\tmp\test2.trace';

END PROCEDURE
```

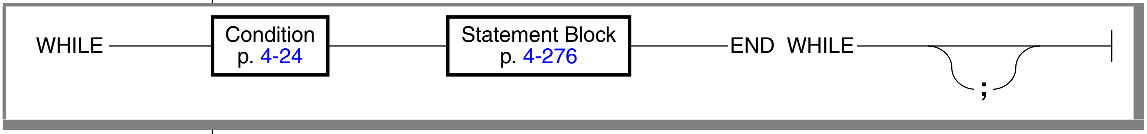
Looking at the Traced Output

To see the traced output, use a text editor or similar utility to display or read the contents of the file.

WHILE

Use the WHILE statement to establish a loop with variable end conditions.

Syntax



Usage

The *condition* is evaluated before the *statement block* first runs and before each subsequent iteration. Iterations continue as long as the *condition* remains TRUE. The loop terminates when the *condition* evaluates to not true.

If any expression within the *condition* evaluates to NULL, the *condition* becomes not true unless you are explicitly testing with the IS NULL operator.

If an expression within the *condition* has an unknown value because it references uninitialized SPL variables, an immediate error results. In this case, the loop terminates, and an exception is raised.

Example of WHILE Loops in an SPL Routine

The following example illustrates the use of WHILE loops in an SPL routine. In the SPL procedure, **simp_while**, the first WHILE loop executes a DELETE statement. The second WHILE loop executes an INSERT statement and increments the value of an SPL variable.

```
CREATE PROCEDURE simp_while()
   DEFINE i INT;
   WHILE EXISTS (SELECT fname FROM customer
       WHERE customer_num > 400)
      DELETE FROM customer WHERE id_2 = 2;
   END WHILE;
   LET i = 1;
   WHILE i < 10
      INSERT INTO tab_2 VALUES (i);
      LET i = i + 1;
   END WHILE;
END PROCEDURE
```

Chapter

Segments

4

In This Chapter

Segments are language elements, such as table names or expressions, that appear as a subdiagram reference box in the syntax diagrams of SQL and SPL statements. A segment that can occur in only one statement is described within the description of the statement. For the sake of clarity, ease of use, and comprehensive treatment, however, most segments that can occur in various SQL or SPL statements are discussed separately in this section.

Scope of Segment Descriptions

The description of each segment includes the following information:

- A brief introduction that explains the purpose of the segment
- A syntax diagram that shows how to enter the segment correctly
- A table that explains the terms in the syntax diagram for which you must substitute names, values, or other specific information
- Rules of usage, typically including examples that illustrate these rules

If a segment consists of multiple parts, the segment description provides similar information about each part. Some descriptions conclude with references to related information in this and in other manuals.

Use of Segment Descriptions

The syntax diagram within each segment description is not a stand-alone diagram. Rather, it is a subdiagram of the syntax of the SQL statements (in Chapter 2) and SPL statements (in Chapter 3) that can include the segment.

SQL or SPL syntax descriptions can refer to segment descriptions in two ways:

- A subdiagram-reference box in a statement syntax diagram can list a segment name and the page where the segment description begins.
- The **Syntax** column of the table beneath a syntax diagram can list a segment name and the page where the segment description begins.

If the syntax diagram for a statement includes a reference to a segment, turn to that segment description to see the complete syntax for the segment.

For example, if you want to write a CREATE VIEW statement that includes a *database* and *database server* qualifiers of the *view* name, first look up the syntax diagram for the CREATE VIEW statement in Chapter 4. The table beneath that diagram refers to the Database Object Name segment for the syntax of *view.* Then use the Database Object Name segment syntax to enter a valid CREATE VIEW statement that also specifies the database name and database server name for the view. For example, the following SQL statement defines a view called **name_only** in the **sales** database on the **boston** database server:

```
CREATE VIEW sales@boston:name_only AS
      SELECT customer_num, fname, lname FROM customer
```

Segments in This Section

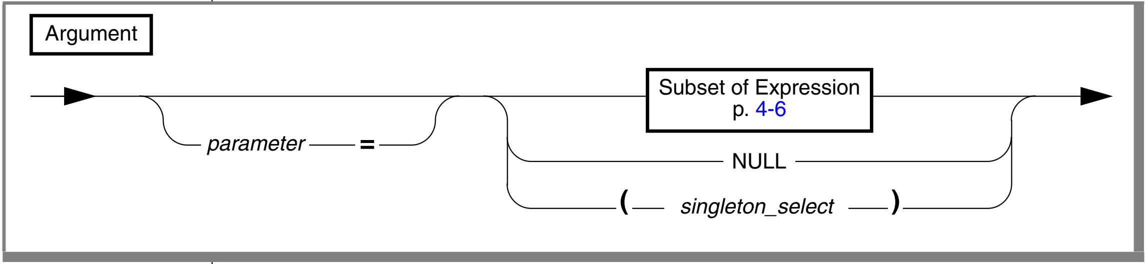
This section describes the following segments:

Argument
Collection-Derived Table
Condition
Database Name
Database Object Name
Data Type
DATETIME Field Qualifier
Expression
External Routine Reference
Identifier
INTERVAL Field Qualifier
Jar Name
Literal Collection
Literal DATETIME
Literal INTERVAL
Literal Number
Literal Row
Optimizer Directives
Owner Name
Purpose Options
Quoted String
Relational Operator
Return Clause
Routine Modifier
Routine Parameter List
Shared-Object Filename
Specific Name
Statement Block

Arguments

Use the Argument segment to pass a specific value to a routine. Use this segment wherever you see a reference to an argument in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
parameter	Name of parameter for which you supply a value	Must match a name that CREATE FUNCTION or CREATE PROCEDURE statement declared.	Expression, p. 4-67
singleton _select	Embedded query that returns a single value	Must return exactly one value of a data type and length compatible with *parameter.*	SELECT, p. 2-581

Usage

The CREATE PROCEDURE or CREATE FUNCTION statement can define a parameter list for a UDR. If the parameter list is not empty, you must enter arguments when you invoke the UDR. An *argument* is a specific value whose data type is compatible with that of the corresponding UDR parameter.

When you execute a UDR, you can enter arguments in either of two ways:

- With a parameter name (in the form *parameter name = expression)*, even if the arguments are not in the same order as the parameters
- By position, with no *parameter* name, where each *expression* is in the same order as the parameter to which the argument corresponds

You cannot mix these two ways of specifying arguments within a single invocation of a routine. If you specify a *parameter* name for one argument, for example, you must use parameter names for all the arguments.

In the following example, both statements are valid for a user-defined procedure that expects three character arguments, **t**, **d**, and **n**:

```
EXECUTE PROCEDURE add_col (t ='customer', d ='integer',
   n ='newint');

EXECUTE PROCEDURE add_col ('customer','newint','integer') ;
```

Comparing Arguments to the Parameter List

When you create or register a UDR with CREATE PROCEDURE or CREATE FUNCTION, you declare a *parameter list* with the names and data types of the parameters that the UDR expects. (Parameter names are optional for external routines written in the C or Java languages.) See "Routine Parameter List" on page 4-266 for details of declaring parameters.

User-defined routines can be *overloaded*, if different routines have the same identifier, but have different numbers of declared parameters. For more information about overloading, see "Routine Overloading and Naming UDRs with a Routine Signature" on page 4-48.

If you attempt to execute a UDR with more arguments than the UDR expects, you receive an error.

If you invoke a UDR with fewer arguments than the UDR expects, the omitted arguments are said to be *missing*. The database server initializes missing arguments to their corresponding default values. This initialization occurs before the first executable statement in the body of the UDR.

If missing arguments do not have default values, the database server initializes the arguments to the value UNDEFINED. You cannot, however, use a variable with a value of UNDEFINED within the UDR. If you do, the database server issues an error.

Subset of Expressions Valid as an Argument

The diagram for "Arguments" on page 4-5 refers to this section.

You can use any expression as an argument, except an aggregate function. If you use a subquery or function call as an argument, the subquery or function must return a single value of the appropriate data type and size. For the syntax and usage of SQL expressions, see "Expression" on page 4-67.

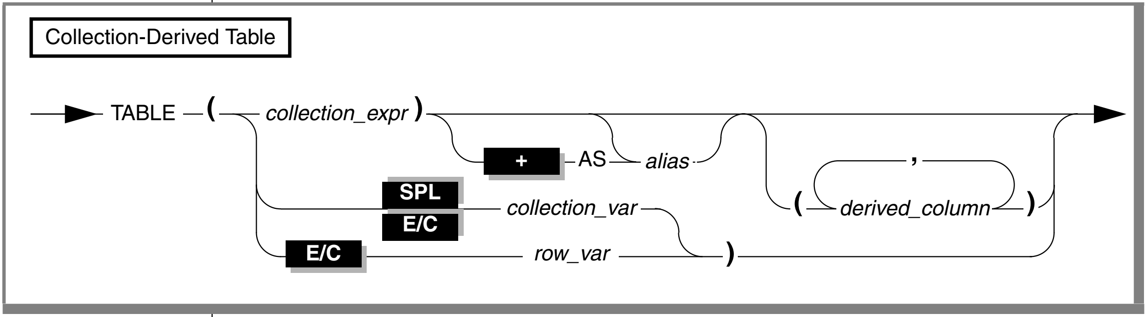
+

IDS

Collection-Derived Table

A *collection-derived table* is a virtual table in which the values in the rows of the table are equivalent to the elements of a collection. Use this segment where you see a reference to Collection-Derived Table in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
alias	Temporary name for a collection-derived table within the scope of a SELECT statement. The default is implementation dependent.	If potentially ambiguous, you must precede *alias* with the AS keyword. See "The AS Keyword" on page 2-595.	Identifier, p. 4-189
collection_expr	Any expression that evaluates to a single collection	See "Restrictions with the Collection-Expression Format" on page 4-9.	Expression, p. 4-67
collection_var, *row_var*	Name of typed or untyped collection variable, or an ESQL/C **row** variable that holds the collection-derived table	Must have been declared in an ESQL/C program or (for *collection_var*) in an SPL routine.	See the *IBM Informix ESQL/C Programmer's Manual* or "DEFINE" on page 3-10.
derived_column	Temporary name for a derived column in a table	If the underlying collection is not a ROW type, you can specify only one derived-column name.	Identifier, page 4-189

Usage

A collection-derived table can appear where a *table* is valid in the UPDATE statement, in the FROM clause of the SELECT or DELETE statement, or in the INTO clause of an INSERT statement.

Use the collection-derived-table segment to accomplish these tasks:

- Access the elements of a collection as you would the rows of a table
- Specify a collection variable to access, instead of a table name
- Specify an ESQL/C **row** variable to access, instead of a table name

The TABLE keyword converts a collection into a virtual table. You can use the collection expression format to query a collection column, or you can use the **collection** variable or **row** variable format to manipulate the data in a collection column.

Accessing a Collection Through a Virtual Table

When you use the collection expression format of the collection-derived-table segment to access the elements of a collection, you can select elements of the collection directly through a virtual table. You can use this format only in the FROM clause of a SELECT statement. The FROM clause can be in either a query or a subquery.

With this format you can use joins, aggregates, the WHERE clause, expressions, the ORDER BY clause, and other operations that are not available when you use the collection-variable format. This format reduces the need for multiple cursors and temporary tables.

Examples of possible collection expressions include column references, scalar subquery, dotted expression, functions, operators (through overloading), collection subqueries, literal collections, collection constructors, cast functions, and so on.

Restrictions with the Collection-Expression Format

When you use the collection-expression format, certain restrictions apply:

- A collection-derived table is read-only.
    - It cannot be the target of INSERT ‘, UPDATE, or DELETE statements.

      To perform insert, update, and delete operations, you must use the collection-variable format.
    - It cannot be the underlying table of an updatable cursor or view.
- You cannot use the WITH ORDINALITY keywords to introduce a new column whose value is the ordinality of a row in the list expression.
- If the collection is a LIST data type, the resulting collection-derived table does not preserve the order of the elements in the LIST.
- The underlying collection expression cannot evaluate to NULL.
- The collection expression cannot contain a reference to a collection on a remote database server.
- The collection expression cannot contain column references to tables that appear in the same FROM clause. That is, the collection-derived table must be independent of other tables in the FROM clause.

  For example, the following statement returns an error because the collection-derived table, `TABLE (parents.children)`, refers to the table **parents**, which is also referenced in the FROM clause:

  ```
  SELECT COUNT(*)
     FROM parents, TABLE(parents.children) c_table
     WHERE parents.id = 1001
  ```

  To counter this restriction, you might write a query that contains a subquery in the select list of the Projection clause:

  ```
  SELECT (SELECT COUNT(*)
        FROM TABLE(parents.children) c_table)
     FROM parents WHERE parents.id = 1001
  ```

E/C

Additional Restrictions That Apply to ESQL/C

In addition to the previously described restrictions, the following restrictions also apply when you use the collection-expression format with ESQL/C:

- You cannot use the untyped collection-host variable, COLLECTION.
- You cannot use the format `TABLE(?)`.

  The data type of the underlying collection variable must be determined statically. To counter this restriction, you can explicitly cast the variable to a typed collection data type (SET, MULTISET, or LIST) that the database server recognizes. For example,

  ```
  TABLE(CAST(? AS type))
  ```
- You cannot use the format `TABLE(:`*hostvar*`)`.

  To counter this restriction, you must explicitly cast the variable to a typed collection data type (SET, MULTISET, or LIST) that the database server recognizes. For example,

  ```
  TABLE(CAST(:hostvar AS type))
  ```

Row Type of the Resulting Collection-Derived Table

If you do not specify a derived-column name, the behavior of the database server depends on the data types of the elements in the underlying collection.

Although a collection-derived table appears to contain columns of individual data types, these columns are, in fact, the fields of a ROW data type. The data type of the ROW type as well as the column name depend on several factors.

If the data type of the elements of the underlying collection expression is *type*, the database server determines the ROW type of the collection-derived table by the following rules:

- If *type* is a ROW type, and no derived column list is specified, then the ROW type of the collection-derived table is *type*.
- If *type* is a ROW type and a derived column list is specified, then the row type of the collection-derived table is an unnamed ROW type whose column types are the same as those of *type* and whose column names are taken from the derived column list.

- If *type* is not a ROW type, the row type of the collection-derived table is an unnamed ROW type that contains one column of *type*, and whose name is specified in the derived column list. If no name is specified, the database server assigns an implementation-dependent name to the column.

The extended examples that the following table shows illustrate these rules. The table uses the following schema for its examples:

```
CREATE ROW TYPE person (name CHAR(255), id INT);
CREATE TABLE parents
   (
   name CHAR(255),
   id INT,
   children LIST (person NOT NULL)
   );
CREATE TABLE parents2
   (
   name CHAR(255),
   id INT,
   children_ids LIST (INT NOT NULL)
   );
```

Row Type	Explicit Derived-Column List	Resulting ROW Type of the Collection-Derived Table	Code Example

Yes	No	*Type*	`SELECT (SELECT c_table.name`
			`FROM TABLE(parents.children) c_table`
			`WHERE c_table.id = 1002)`
			`FROM parents WHERE parents.id = 1001`
			In this example, the ROW type of **c_table** is **parents**.

Yes	Yes	Unnamed ROW type of which the column type is *Type* and the column name is the name in the derived-column list	`SELECT (SELECT c_table.c_name`
			`FROM TABLE(parents.children)`
			`c_table(c._name, c_id)`
			`WHERE c_table.c_id = 1002)`
			`FROM parents WHERE parents.id = 1001`
			In this example, the ROW type of **c_table** is **ROW(c_name CHAR(255), c_id INT)**.

No	No	Unnamed ROW that contains one column of *Type* that is assigned an implementation-dependent name	In the following example, if you do not specify **c_id**, the database server assigns a name to the derived column. In this case, the ROW type of **c_table** is **ROW(***server_defined_name* **INT)**.

No	Yes	Unnamed ROW type that contains one column of *Type* whose name is in the derived-column list.	`SELECT(SELECT c_table.c_id FROM`
			`TABLE(parents2.child_ids) c_table (c_id)`
			`WHERE c_table.c_id = 1002)`
			`FROM parents WHERE parents.id = 1001`
			Here the ROW type of **c_table** is **ROW(c_id INT)**.

The following program fragment creates a collection-derived table using an SPL function that returns a single value:

```
CREATE TABLE wanted(person_id int);
CREATE FUNCTION
    wanted_person_count (person_set SET(person NOT NULL))
RETURNS INT;
RETURN( SELECT COUNT (*)
    FROM TABLE (person_set) c_table, wanted
    WHERE c_tabel.id = wanted.person_id);
END FUNCTION;
```

The next program fragment shows the more general case of creating a collection-derived table using an SPL function that returns multiple values:

```
-- Table of categories and child categories,
-- allowing any number of levels of subcategories
CREATE TABLE CategoryChild (
         categoryId      INTEGER,
         childCategoryId SMALLINT
);

INSERT INTO CategoryChild VALUES (1, 2);
INSERT INTO CategoryChild VALUES (1, 3);
INSERT INTO CategoryChild VALUES (1, 4);
INSERT INTO CategoryChild VALUES (2, 5);
INSERT INTO CategoryChild VALUES (2, 6);
INSERT INTO CategoryChild VALUES (5, 7);
INSERT INTO CategoryChild VALUES (7, 8);
INSERT INTO CategoryChild VALUES (7, 9);
INSERT INTO CategoryChild VALUES (4, 10);

-- "R" == ROW type
CREATE ROW TYPE categoryLevelR (
         categoryId      INTEGER,
         level   SMALLINT );

-- DROP FUNCTION categoryDescendants (
--              INTEGER, SMALLINT );
CREATE FUNCTION categoryDescendants (
         pCategoryId INTEGER,
         pLevel      SMALLINT DEFAULT 0 )
RETURNS MULTISET (categoryLevelR NOT NULL);

-- "p" == Prefix for Parameter names
-- "l" == Prefix for Local variable names
DEFINE lCategoryId LIKE CategoryChild.categoryId;
DEFINE lRetSet MULTISET (categoryLevelR NOT NULL);
DEFINE lCatRow categoryLevelR;
-- TRACE ON;
-- Must initialize collection before inserting rows
LET lRetSet
   = 'MULTISET{}' :: MULTISET (categoryLevelR NOT NULL);
```

```
FOREACH
SELECT childCategoryId INTO lCategoryId
   FROM CategoryChild WHERE categoryId = pCategoryId;
INSERT INTO TABLE (lRetSet)
   VALUES (ROW (lCategoryId, pLevel+1)::categoryLevelR);

-- INSERT INTO TABLE (lRetSet);
-- EXECUTE FUNCTION categoryDescendantsR ( lCategoryId,
-- pLevel+1 );
-- Need to iterate over results and insert into SET.
-- See the SQL Tutorial, pg. 10-52:
-- "Tip: You can only insert one value at a time
-- into a simple collection."
   FOREACH
   EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
     INTO lCatRow;
       INSERT INTO TABLE (lRetSet)
           VALUES (lCatRow);
   END FOREACH;
END FOREACH;

RETURN lRetSet;
END FUNCTION
;


-- "R" == recursive
-- DROP FUNCTION categoryDescendantsR (INTEGER, SMALLINT);
CREATE FUNCTION categoryDescendantsR (
         pCategoryId INTEGER,
         pLevel      SMALLINT DEFAULT 0
)
RETURNS categoryLevelR
;

DEFINE lCategoryId      LIKE CategoryChild.categoryId;
DEFINE lCatRow          categoryLevelR;

FOREACH
   SELECT  childCategoryId
   INTO    lCategoryId
   FROM    CategoryChild
   WHERE   categoryId = pCategoryId
   RETURN ROW (lCategoryId, pLevel+1)::categoryLevelR WITH RESUME;

   FOREACH
   EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
      INTO    lCatRow
      RETURN  lCatRow WITH RESUME;
   END FOREACH;
END FOREACH;

END FUNCTION;
```

```
-- Test the functions:
SELECT lev, col
FROM    TABLE ((
         categoryDescendants (1, 0)
         )) AS CD (col, lev)
;
```

Accessing a Collection Through a Collection Variable

When you use the collection-variable format of the collection-derived-table segment, you use a host or program variable to access and manipulate the elements of a collection. This format allows you to modify the contents of a variable as you would a table in the database and then update the actual table with the contents of the **collection** variable.

You can use the collection-variable format (the TABLE keyword preceding a collection variable) in place of the name of a table, synonym, or view in the following SQL statements:

- The FROM clause of the SELECT statement to access an element of the collection variable
- The INTO clause of the INSERT statement to add a new element to the collection variable
- The DELETE statement to remove an element from the collection variable
- The UPDATE statement to modify an existing element in the collection variable

E/C

- The DECLARE statement to declare a select or insert cursor to access multiple elements of an ESQL/C collection-host variable
- The FETCH statement to retrieve a single element from a collection-host variable that is associated with a select cursor
- The PUT statement to retrieve a single element from a collection-host variable that is associated with an insert cursor ♦

SPL

- The FOREACH statement to declare a cursor to access multiple elements of an SPL collection variable and to retrieve a single element from this collection variable ♦

IDS

Using a Collection Variable to Manipulate Collection Elements

When you use data manipulation statements (SELECT, INSERT, UPDATE, or DELETE) in conjunction with a collection variable, you can modify one or more elements in a collection.

To modify elements in a collection

1. Create a collection variable in your SPL routine or ESQL/C program.

   For information on how to declare a collection variable in ESQL/C, see the *IBM Informix ESQL/C Programmer's Manual*. For information on how to define a collection variable in SPL, see "DEFINE" on page 3-10.

E/C

2. In ESQL/C, allocate memory for the collection; see "ALLOCATE COLLECTION" on page 2-8. ♦
3. Optionally, use a SELECT statement to select a collection column into the collection variable.

   If the collection variable is an untyped COLLECTION variable, you must perform a SELECT from the collection column before you use the variable in the collection-derived-table segment. The SELECT statement allows the database server to obtain the collection type.
4. Use the appropriate data manipulation statement with the collection-derived-table segment to add, delete, or update collection elements in the collection variable.

   To insert more than one element or to update or delete a *specific* element of a collection, you must use a cursor for the collection variable.

   - For more information on how to use an update cursor with ESQL/C, see "DECLARE" on page 2-323.
   - For more information on how to use an update cursor with SPL, see "FOREACH" on page 3-27.
5. After the collection variable contains the correct elements, use an INSERT or UPDATE statement on the table or view that holds the actual collection column to save the changes that the collection variable holds.

   - With the UPDATE statement, specify the collection variable in the SET clause.
   - With the INSERT statement, specify the collection variable in the VALUES clause.

The collection variable stores the elements of the collection. It has no intrinsic connection, however, with a database column. Once the collection variable contains the correct elements, you must then save the variable into the actual collection column of the table with either an INSERT or an UPDATE statement.

Example of Deleting from a Collection in ESQL/C

Suppose that the **set_col** column of a row in the **table1** table is defined as a SET and for one row contains the values {1,8,4,5,2}. The following ESQL/C code fragment uses an update cursor and a DELETE statement with a WHERE CURRENT OF clause to delete the element whose value is 4:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(smallint not null) a_set;
   int an_int;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL allocate collection :a_set;
EXEC SQL select set_col into :a_set from table1
   where int_col = 6;
EXEC SQL declare set_curs cursor for
   select * from table(:a_set)
   for update;

EXEC SQL open set_curs;
while (i<coll_size)
{
   EXEC SQL fetch set_curs into :an_int;
   if (an_int = 4)
   {
      EXEC SQL delete from table(:a_set)
         where current of set_curs;
      break;
   }
   i++;
}
EXEC SQL update table1 set set_col = :a_set
   where int_col = 6;
EXEC SQL deallocate collection :a_set;
EXEC SQL close set_curs;
EXEC SQL free set_curs;
```

After the DELETE statement executes, this collection variable contains the elements {1,8,5,2}. The UPDATE statement at the end of this code fragment saves the modified collection into the **set_col** column. Without this UPDATE statement, element 4 of the collection column is not deleted.

For information on how to use collection-host variables in an ESQL/C program, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

Example of Deleting from a Collection

Suppose that the **set_col** column of a row in the **table1** table is defined as a SET and one row contains the values {1,8,4,5,2}. The following SPL code fragment uses a FOREACH loop and a DELETE statement with a WHERE CURRENT OF clause to delete the element whose value is 4:

```
CREATE_PROCEDURE test6()

   DEFINE a SMALLINT;
   DEFINE b SET(SMALLINT NOT NULL);
   SELECT set_col INTO b FROM table1
      WHERE id = 6;
      -- Select the set in one row from the table
      -- into a collection variable
   FOREACH cursor1 FOR
      SELECT * INTO a FROM TABLE(b);
         -- Select each element one at a time from
         -- the collection derived table b into a
      IF a = 4 THEN
         DELETE FROM TABLE(b)
            WHERE CURRENT OF cursor1;
            -- Delete the element if it has the value 4
         EXIT FOREACH;
      END IF;
   END FOREACH;

   UPDATE table1 SET set_col = b
      WHERE id = 6;
      -- Update the base table with the new collection

END PROCEDURE;
```

This SPL routine declares two SET variables, **a** and **b**, each to hold a set of SMALLINT values. The first SELECT statement copies a SET column from one row of **table1** into variable **b**. The routine then declares a cursor called **cursor1** that copies one element at a time from **b** into SET variable **a**. When the cursor is positioned on the element whose value is 4, the DELETE statement removes that element from SET variable **b**. Finally, the UPDATE statement replaces the row of **table1** with the new collection that is stored in variable **b**.

For information on how to use collection variables in an SPL routine, see the *IBM Informix Guide to SQL: Tutorial*.

Example of Updating a Collection

Suppose that the **set_col** column of a table called **table1** is defined as a SET and that it contains the values {1,8,4,5,2}. The following ESQL/C program changes the element whose value is 4 to a value of 10:

```
main
{
   EXEC SQL BEGIN DECLARE SECTION;
      int a;
      collection b;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL allocate collection :b;
   EXEC SQL select set_col into :b from table1
      where int_col = 6;

   EXEC SQL declare set_curs cursor for
      select * from table(:b) for update;
   EXEC SQL open set_curs;
   while (SQLCODE != SQLNOTFOUND)
   {
      EXEC SQL fetch set_curs into :a;
      if (a = 4)
      {
         EXEC SQL update table(:b)(x)
            set x = 10 where current of set_curs;
         break;
      }
   }
   EXEC SQL update table1 set set_col = :b
      where int_col = 6;
   EXEC SQL deallocate collection :b;
   EXEC SQL close set_curs;
   EXEC SQL free set_curs;
}
```

After you execute this ESQL/C program, the **set_col** column in **table1** contains the values {1,8,10,5,2}.

This ESQL/C program defines two **collection** variables, **a** and **b**, and selects a SET from **table1** into **b**. The WHERE clause ensures that only one row is returned. Then the program defines a collection cursor, which selects elements one at a time from **b** into **a**. When the program locates the element with the value 4, the first UPDATE statement changes that element value to 10 and exits the loop.

In the first UPDATE statement, **x** is a derived-column name used to update the current element in the collection-derived table. The second UPDATE statement updates the base table **table1** with the new collection.

For information on how to use **collection** host variables in an ESQL/C program, see the discussion of complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

Example of Inserting a Value into a Multiset Collection

Suppose the ESQL/C host variable **a_multiset** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(integer not null) a_multiset;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement adds a new MULTISET element of 142,323 to **a_multiset**:

```
EXEC SQL allocate collection :a_multiset;
EXEC SQL select multiset_col into :a_multiset from table1
   where id = 107;
EXEC SQL insert into table(:a_multiset) values (142323);
EXEC SQL update table1 set multiset_col = :a_multiset
   where id = 107;

EXEC SQL deallocate collection :a_multiset;
```

When you insert elements into a **client-collection** variable, you cannot specify a SELECT statement or an EXECUTE FUNCTION statement in the VALUES clause of the INSERT. When you insert elements into a **server-collection** variable, however, the SELECT and EXECUTE FUNCTION statements are valid in the VALUES clause. For more information on **client-** and **server-collection** variables, see the *IBM Informix ESQL/C Programmer's Manual*.

Accessing a Nested Collection

If the element of the collection is itself a complex type (**collection** or **row** type), the collection is a *nested collection*. For example, suppose the ESQL/C **collection** variable, **a_set**, is a nested collection that is defined as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(list(integer not null)) a_set;
   client collection list(integer not null) a_list;
   int an_int;
EXEC SQL END DECLARE SECTION;
```

To access the elements (or fields) of a nested collection, use a **collection** or **row** variable that matches the element type (**a_list** and **an_int** in the preceding code fragment) and a select cursor.

Accessing a Row Variable

The TABLE keyword can make an ESQL/C **row** variable a collection-derived table. That is, a row appears as a table in an SQL statement. For a **row** variable, think of the collection-derived table as a table of one row, with each field of the **row** type being a column of the row. Use the TABLE keyword in place of the name of a table, synonym, or view in these SQL statements:

- The FROM clause of the SELECT statement to access a field of the **row** variable
- The UPDATE statement to modify an existing field in the **row** variable

The DELETE and INSERT statements do not support a **row** variable in the collection-derived-table segment.

For example, suppose an ESQL/C host variable **a_row** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   row(x int, y int, length float, width float) a_row;
EXEC SQL END DECLARE SECTION;
```

The following ESQL/C code fragment adds the fields in the **a_row** variable to the **row_col** column of the **tab_row** table:

```
EXEC SQL update table(:a_row)
   set x=0, y=0, length=10, width=20;
EXEC SQL update rectangles set rect = :a_row;
```

Related Information

Related statements: DECLARE, DELETE, DESCRIBE, FETCH, INSERT, PUT, SELECT, UPDATE, DEFINE, and FOREACH

For information on how to use COLLECTION variables in an SPL routine, see the *IBM Informix Guide to SQL: Tutorial*.

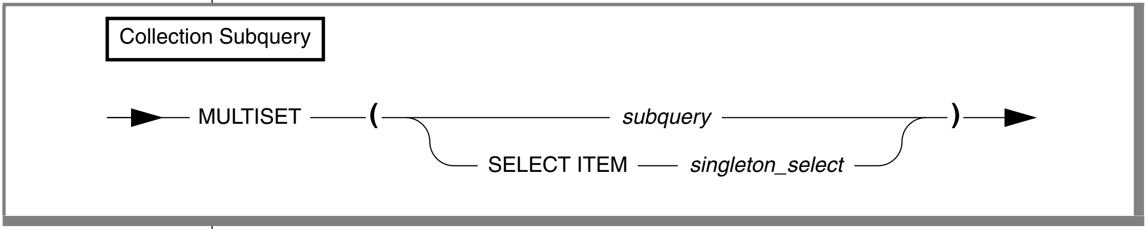
For information on how to use **collection** or **row** variables in an ESQL/C program, see the chapter on complex data types in the *IBM Informix ESQL/C Programmer's Manual*.

IDS

Collection Subquery

You can use a Collection Subquery to create a multiset collection from the results of a subquery.

Syntax



Element	Purpose	Restrictions	Syntax
singleton _select	Subquery returning exactly one row	Subquery cannot repeat the SELECT keyword, nor include the FIRST or ORDER BY clause	SELECT, p. 2-581
subquery	Embedded query	Cannot contain the FIRST or ORDER BY clause	SELECT, p. 2-581

Usage

The MULTISET and SELECT ITEM keywords have the following significance:

- MULTISET specifies a collection of elements that can contain duplicate values, but that has no specific order of elements.
- SELECT ITEM supports only one expression in the projection list. You cannot repeat the SELECT keyword in the singleton subquery.

You can use a collection subquery in the following contexts:

- The SELECT and WHERE clauses of the SELECT statement
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement
- Wherever you can use a collection expression (that is, any expression that evaluates to a single collection)
- As an argument passed to a user-defined routine

The following restrictions apply to a collection subquery:

- The Projection clause cannot contain duplicate column (field) names.
- It cannot contain aliases for table names. (But it can use aliases for column (field) names, as in some of the examples that follow.)
- It is read-only.
- It cannot be opened twice.
- It cannot contain NULL values.
- It cannot contain syntax that attempts to seek within the subquery.

A collection subquery returns a multiset of unnamed ROW data types. The fields of this ROW type are elements in the projection list of the subquery. The examples that follow access the schema created with these statements:

```
CREATE ROW TYPE rt1 (a INT);
CREATE ROW TYPE rt2 (x int, y rt1);
CREATE TABLE tab1 (col1 rt1, col2 rt2);
CREATE TABLE tab2 OF TYPE rt1;
CREATE TABLE tab3 (a ROW(x INT));
```

The following examples of collection subqueries return the MULTISET collections that are listed to the right of the subquery.

Collection Subquery	Resulting Collections
`MULTISET (SELECT * FROM tab1)...`	`MULTISET(ROW(col1 rt1, col2 rt2))`
`MULTISET (SELECT col2.y FROM tab1)...`	`MULTISET(ROW(y rt1))`
`MULTISET (SELECT * FROM tab2)...`	`MULTISET(ROW(a int))`
`MULTISET(SELECT p FROM tab2 p)...`	`MULTISET(ROW(p rt1))`
`MULTISET (SELECT * FROM tab3)...`	`MULTISET(ROW(a ROW(x int)))`

The following is another collection subquery:

```
SELECT f(MULTISET(SELECT * FROM tab1 WHERE tab1.x = t.y))
   FROM t WHERE t.name = 'john doe';
```

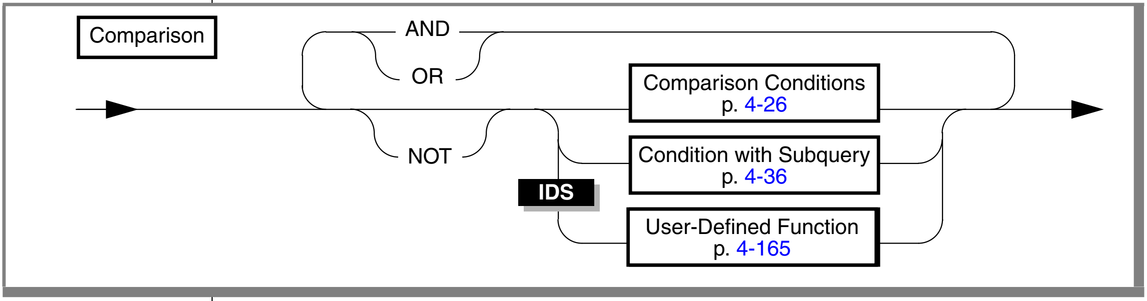
The following collection subquery includes the UNION operator:

```
SELECT f(MULTISET(SELECT id FROM tab1
UNION
SELECT id FROM tab2 WHERE tab2.id2 = tab3.id3)) FROM tab3;
```

Condition

Use a condition to test whether data meets certain qualifications. Use this segment wherever you see a reference to a condition in a syntax diagram.

Syntax


Comparison
AND
OR
NOT
IDS
Comparison Conditions p. 4-26
Condition with Subquery p. 4-36
User-Defined Function p. 4-165

Usage

A *condition* is a search criterion, or criteria connected by the logical operators AND or OR. Conditions can be classified into the following categories:

- Comparison conditions (also called filters or Boolean expressions)
- Conditions within a subquery
- User-defined functions (Dynamic Server only)

A condition can contain only an aggregate function if it is used in the HAVING clause of a SELECT statement or in the HAVING clause of a subquery.

No aggregate function can appear in a condition in the WHERE clause of a DELETE, SELECT, or UPDATE statement unless both of the following are TRUE:

- Aggregate is on a correlated column originating from a parent query.
- The WHERE clause appears in a subquery within a HAVING clause.

IDS

In Dynamic Server, user-defined functions are not valid as conditions in the following context:

- In the HAVING clause of a SELECT statement
- In the definition of a check constraint ♦

SPL routines are not valid as conditions in the following contexts:

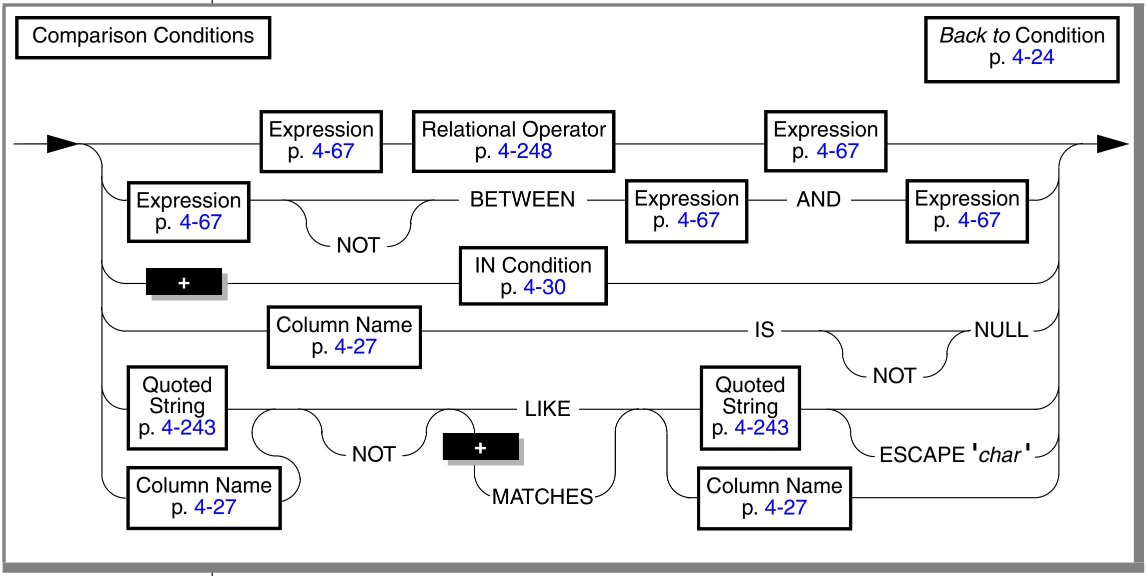
- In the definition of a check constraint
- In the ON clause of a SELECT statement
- In the WHERE clause of a DELETE, SELECT, or UPDATE statement

External routines are not valid as conditions in the following contexts:

- In the definition of a check constraint
- In the ON clause of a SELECT statement
- In the WHERE clause of a DELETE, SELECT, or UPDATE statement
- In the WHEN clause of CREATE TRIGGER
- In the IF, CASE, or WHILE statements of SPL

Comparison Conditions (Boolean Expressions)

Five kinds of comparison conditions exist: Relational Operator, BETWEEN, IN, IS NULL, and LIKE and MATCHES. Comparison conditions are often called Boolean expressions because they return a TRUE or FALSE result. Their syntax is summarized in this diagram and explained in the sections that follow.



Element	Purpose	Restrictions	Syntax
char	A single ASCII character to be the nondefault escape character in the quoted string Single (') and double quotation marks (") are not valid here as *char*.	See "ESCAPE with LIKE" on page 4-35 and "ESCAPE with MATCHES" on page 4-36.	Quoted String, p. 4-243

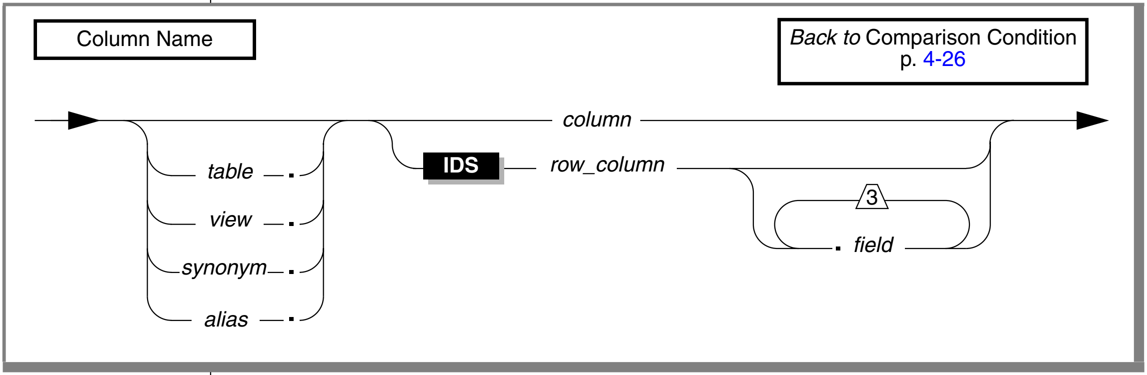
The following sections describe the different types of comparison conditions:

- "Relational-Operator Condition" on page 4-28
- "BETWEEN Condition" on page 4-29
- "IN Condition" on page 4-30
- "IS NULL Condition" on page 4-32
- "LIKE and MATCHES Condition" on page 4-32

For a discussion of comparison conditions in the context of the SELECT statement, see "Using a Condition in the WHERE Clause" on page 2-614.

***Warning:** A literal DATE or DATETIME value in a comparison condition should specify 4 digits for the year. When you specify a 4-digit year, the **DBCENTURY** environment variable has no effect on the result. When you specify a 2-digit year, **DBCENTURY** can affect how the database server interprets the comparison condition, which might not work as you intended. For more information, see the "IBM Informix Guide to SQL: Reference."*

Column Name



Element	Purpose	Restrictions	Syntax
alias	Temporary alternative name for table or view	Must be defined in the FROM clause of the SELECT statement.	Identifier, p. 4-189
column	Name of a column	Must exist in the specified table.	Identifier, p. 4-189
field	A field to compare in a ROW type column	Must be a component of *row-column name* or *field name* (for nested rows).	Identifier, p. 4-189
row_column	A column of type ROW	Must be an existing named ROW type or unnamed ROW type.	Identifier, p. 4-189
synonym, table, view	Name of a synonym, table, or view	The synonym and the table or view to which it points must exist in the database.	Database Object Name, p. 4-46

For more information on the meaning of the *column* name in these conditions, see the "IS NULL Condition" on page 4-32 and the "LIKE and MATCHES Condition" on page 4-32.

Quotation Marks in Conditions

When you compare a column expression with a constant expression in any type of comparison condition, observe the following rules:

- If the column has a numeric data type, do not surround the constant expression with quotation marks.
- If the column has a character data type, surround the constant expression with quotation marks.
- If the column has a time data type, surround the constant expression with quotation marks.

Otherwise, you might get unexpected results.

The following example shows the correct use of quotation marks in comparison conditions. Here the **ship_instruct** column has a character data type, the **order_date** column has a DATE data type, and the **ship_weight** column has a numeric data type.

```
SELECT * FROM orders
   WHERE ship_instruct = 'express'
   AND order_date > '05/01/98'
   AND ship_weight < 30
```

Relational-Operator Condition

The following examples show some relational-operator conditions:

```
city[1,3] = 'San'

o.order_date > '6/12/98'

WEEKDAY(paid_date) = WEEKDAY(CURRENT- (31 UNITS DAY))

YEAR(ship_date) < YEAR (TODAY)

quantity <= 3

customer_num <> 105

customer_num != 105
```

If either expression is NULL for a row, the condition evaluates to FALSE. For example, if the **paid_date** column has a NULL, you cannot use either of the following statements to retrieve that row:

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date = ''

SELECT customer_num, order_date FROM orders
   WHERE NOT PAID !=''
```

An IS NULL condition finds a NULL value, as the next example shows. The IS NULL condition is described in "IS NULL Condition" on page 4-32.

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date IS NULL
```

BETWEEN Condition

For a BETWEEN test to be TRUE, the value of the expression on the left of the BETWEEN keyword must be in the inclusive range of the values of the two expressions on the right of the BETWEEN keyword.

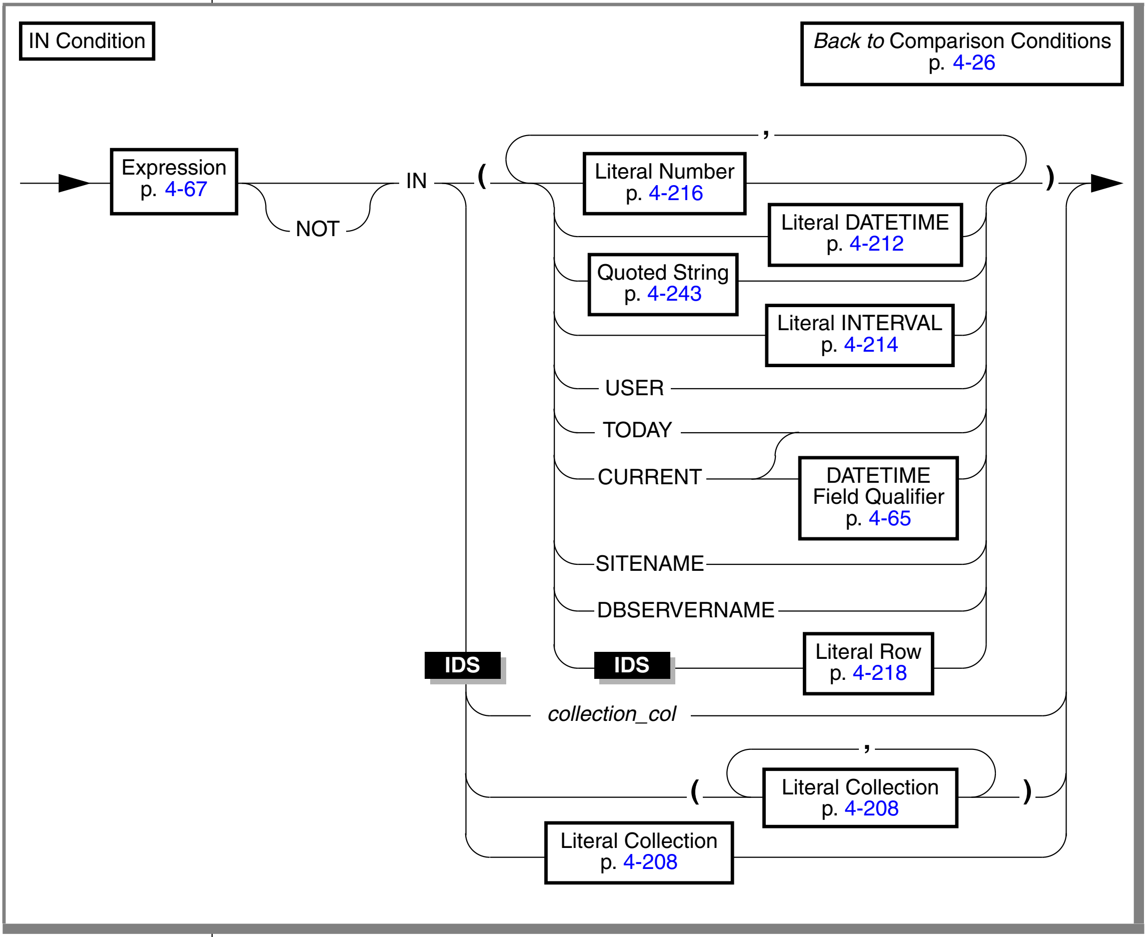
NULL values do not satisfy the condition, and you cannot use NULL for either expression that defines the range.

The following examples show some BETWEEN conditions:

```
order_date BETWEEN '6/1/97' and '9/7/97'

zipcode NOT BETWEEN '94100' and '94199'

EXTEND(call_dtime, DAY TO DAY) BETWEEN
   (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT

lead_time BETWEEN INTERVAL (1) DAY TO DAY
   AND INTERVAL (4) DAY TO DAY

unit_price BETWEEN loprice AND hiprice
```

IN Condition

The IN condition is satisfied when the expression to the left of the keyword IN is included in the list of items.


IN Condition
Back to Comparison Conditions p. 4-26
Expression p. 4-67
NOT
IN
(
,
Literal Number p. 4-216
Literal DATETIME p. 4-212
Quoted String p. 4-243
Literal INTERVAL p. 4-214
USER
TODAY
CURRENT
DATETIME Field Qualifier p. 4-65
SITENAME
DBSERVERNAME
IDS
IDS
Literal Row p. 4-218
)
collection_col
,
(
Literal Collection p. 4-208
)
Literal Collection p. 4-208

Element	Purpose	Restrictions	Syntax
collection_col	Name of a collection column that is used in an IN condition	The column must exist in the specified table.	Identifier, p. 4-189

If you specify the NOT operator, the IN condition is TRUE when the expression is not in the list of items. NULL values do not satisfy the IN condition.

The following examples show some IN conditions:

```
WHERE state IN ('CA', 'WA', 'OR')
WHERE manu_code IN ('HRO', 'HSK')
WHERE user_id NOT IN (USER)
WHERE order_date NOT IN (TODAY)
```

E/C

In ESQL/C, the built-in TODAY function is evaluated at execution time. The built-in CURRENT function is evaluated when a cursor opens or when the query executes, if it is a singleton SELECT statement. ♦

The built-in USER function is case sensitive; for example, it interprets **minnie** and **Minnie** as different values.

IDS

Using the IN Operator with Collection Data Types

You can use the IN operator to determine if an element is contained in a collection. The collection can be a simple or nested collection. (In a *nested* collection type, the element type of the collection is also a collection type.) When you use IN to search for an element of a collection, the expression to the left or right of the IN keyword cannot contain a BYTE or TEXT data type.

Suppose you create the following table that contains two collection columns:

```
CREATE TABLE tab_coll
(
set_num SET(INT NOT NULL),
list_name LIST(SET(CHAR(10) NOT NULL) NOT NULL)
);
```

The following partial examples show how you might use the IN operator for search conditions on the collection columns of the **tab_coll** table:

```
WHERE 5 IN set_num
WHERE 5.0::INT IN set_num
WHERE "5" NOT IN set_num
WHERE set_num IN ("SET{1,2,3}", "SET{7,8,9}")
WHERE "SET{'john', 'sally', 'bill'}" IN list_name
WHERE list_name IN ("LIST{""SET{'bill','usha'}"",
                  ""SET{'ann' 'moshi'}""}",
               "LIST{""SET{'bob','ramesh'}"",
                  ""SET{'bomani' 'ann'}""}")
```

In general, when you use the IN operator on a collection data type, the database server checks whether the value on the left of the IN operator is an element in the set of values on the right of the IN operator.

IS NULL Condition

The IS NULL condition is satisfied if the column contains a NULL value. If you use the IS NOT NULL operator, the condition is satisfied when the column contains a value that is not NULL. The following example shows an IS NULL condition:

```
WHERE paid_date IS NULL
```

Conditions that use the IS NULL operator are exceptions to the usual rule that SQL expressions in which an operator has a NULL operand return `FALSE`.

LIKE and MATCHES Condition

A LIKE or MATCHES condition tests for matching character strings. The condition is `TRUE`, or satisfied, when either of the following tests is `TRUE`:

- The value of the column on the left matches the pattern that the quoted string specifies. You can use wildcard characters in the string. NULL values do not satisfy the condition.
- The value of the column on the left matches the pattern that the column on the right specifies. The value of the column on the right serves as the matching pattern in the condition.

You can use the single quote (') only with the quoted string to match a literal quote; you cannot use the ESCAPE clause. You can use the quote character as the escape character in matching any other pattern if you write it as `''''`.

Important: *You cannot specify a row-type column in a LIKE or MATCHES condition. A row-type column is a column that is defined on a named or unnamed ROW type.*

NOT Operator

The NOT operator makes the search condition successful when the column on the left has a value that is not NULL and that does not match the pattern that the quoted string specifies.

For example, the following conditions exclude all rows that begin with the characters `Baxter` in the **lname** column:

```
WHERE lname NOT LIKE 'Baxter%'
WHERE lname NOT MATCHES 'Baxter*'
```

LIKE Operator

The LIKE operator supports these wildcard characters in the quoted string.

Wildcard	Effect
%	Matches zero or more characters
_	Matches any single character
\	Removes the special significance of the next character (to match a literal `%` or _ or \ by specifying \`%` or _ or \\ )

Using the backslash (\ ) symbol as the default escape character is an Informix extension to the ANSI/ISO-standard for SQL.

ANSI

In an ANSI-compliant database, you can only use an escape character to escape a percent sign (%), an underscore (_), or the escape character itself. ♦

The following condition tests for the string `tennis`, alone or in a longer string, such as `tennis ball` or `table tennis paddle`:

```
WHERE description LIKE '%tennis%'
```

The next example tests for **description** rows containing an underscore. Here backslash (\ ) is necessary because underscore (_) is a wildcard character.

```
WHERE description LIKE '%\_%'
```

The LIKE operator has an associated operator function called **like( )**. You can define a **like( )** function to handle your own user-defined data types. See also *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

MATCHES Operator

The MATCHES operator supports wildcard characters in the quoted string.

Wildcard	Effect
*	Matches any string of zero or more characters
?	Matches any single character
[. . .]	Matches any of the enclosed characters, including ranges, as in [a-z]. Characters within the brackets cannot be escaped.
^	As the first character within the brackets matches any character that is not listed. Thus, [^abc] matches any character except a, b, or c.
\	Removes the special significance of the next character (to match a literal \ or any other wildcard by specifying \\ or* or \? and so forth)

The following condition tests for the string `tennis`, alone or within a longer string, such as `tennis ball` or `table tennis paddle`:

```
WHERE description MATCHES '*tennis*'
```

The following condition is `TRUE` for the names `Frank` and `frank`:

```
WHERE fname MATCHES '[Ff]rank'
```

The following condition is `TRUE` for any name that begins with either `F` or `f`:

```
WHERE fname MATCHES '[Ff]*'
```

The next condition is `TRUE` for any name that ends with the letters `a`, `b`, `c`, or `d`:

```
WHERE fname MATCHES '*[a-d]'
```

MATCHES has an associated **matches( )** operator function. You can define a **matches( )** function for your own user-defined data types. For more information, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

GLS

If **DB_LOCALE** or SET COLLATION specifies a nondefault locale supporting a localized collation, and you specify a range for the MATCHES operator using bracket ([. . .]) symbols, the database server uses the localized collating order, instead of code-set order, to interpret the range and to sort values that have CHAR, CHARACTER VARYING, LVARCHAR, NCHAR, NVARCHAR, and VARCHAR data types.

This behavior is an exception to the usual rule that only NCHAR and NVARCHAR data types can be sorted in a localized collating order. For more information on the GLS aspects of conditions that include the MATCHES or LIKE operators, see the *IBM Informix GLS User's Guide*. ♦

ESCAPE with LIKE

The ESCAPE clause can specify a nondefault escape character. For example, if you specify z in the ESCAPE clause, then a quoted string operand that included z_ is interpreted as including a literal underscore (_) character, rather than _ as a wildcard. Similarly, z% would be interpreted as a literal percent (%) sign, rather than including % as a wildcard. Finally, the characters zz in a string would be interpreted as single literal z. The following statement retrieves rows from the **customer** table in which the **company** column includes a literal underscore character:

```
SELECT * FROM customer WHERE company LIKE '%z_%' ESCAPE 'z'
```

You can also use a host variable that contains a single character. The next statement uses a host variable to specify an escape character:

```
EXEC SQL BEGIN DECLARE SECTION;
   char escp='z';
   char fname[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL select fname from customer
   into :fname where company like '%z_%' escape :escp;
```

ESCAPE with MATCHES

The ESCAPE clause can specify a nondefault escape character. Use this as you would the backslash to include a question mark (?), an asterisk (*), a caret (^), or a left ([ ) or right ( ]) bracket as a literal character within the quoted string, to prevent them from being interpreted as special characters. If you choose to use `z` as the escape character, the characters `z?` in a string stand for the question mark (?). Similarly, the characters `z*` stand for the asterisk (*). Finally, the characters `zz` in the string stand for the single character `z`.

The following example retrieves rows from the **customer** table in which the value of the **company** column includes the question mark (?):

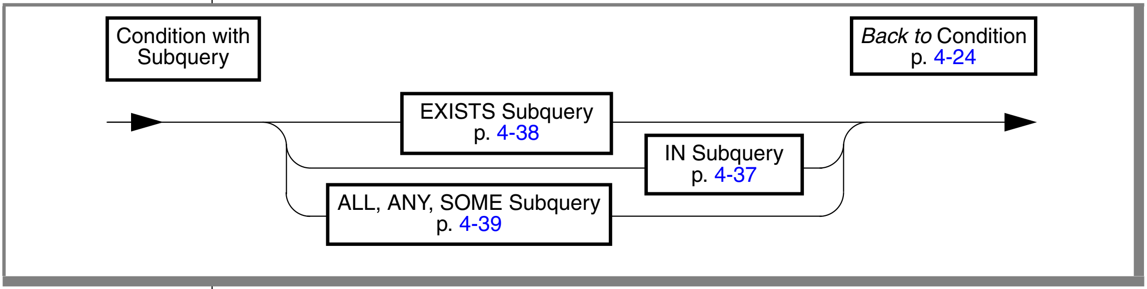
```
SELECT * FROM customer WHERE company MATCHES '*z?*' ESCAPE 'z'
```

Stand-Alone Condition

A stand-alone condition can be any expression that is not explicitly listed in the syntax for the comparison condition. Such an expression is valid as a condition only if it returns a BOOLEAN value. For example, the following example returns a value of the BOOLEAN data type:

```
funcname(x)
```

Condition with Subquery



You can include a SELECT statement within a condition; this combination is called a *subquery*. You can use a subquery in a SELECT statement to perform the following functions:

- Compare an expression to the result of another SELECT statement
- Determine whether an expression is included in the results of another SELECT statement
- Ask whether another SELECT statement selects any rows

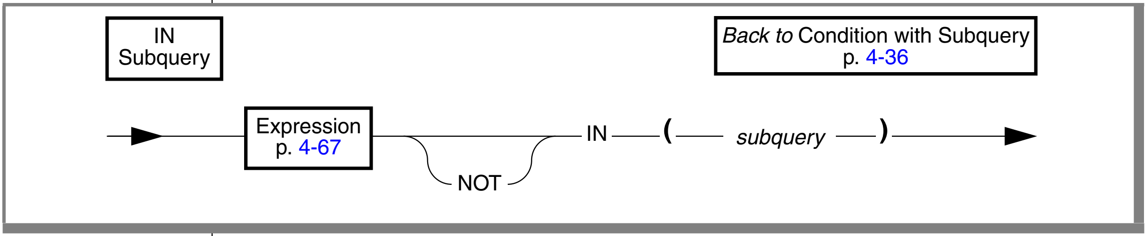
The subquery can depend on the current row that the outer SELECT statement is evaluating; in this case, the subquery is called a *correlated subquery*.

The following sections describe subquery conditions and their syntax. For a discussion of types of subquery conditions in the context of the SELECT statement, see "Using a Condition in the WHERE Clause" on page 2-614.

A subquery can return a single value, no value, or a set of values, depending on its context. If a subquery returns a value, it must select only a single column. If the subquery simply checks whether a row (or rows) exists, it can select any number of rows and columns.

A subquery cannot contain BYTE or TEXT data types, nor can it contain an ORDER BY clause. For a complete description of SELECT syntax and usage, see "SELECT" on page 2-581.

IN Subquery



Element	Purpose	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST nor the ORDER BY clause.	SELECT, p. 2-581

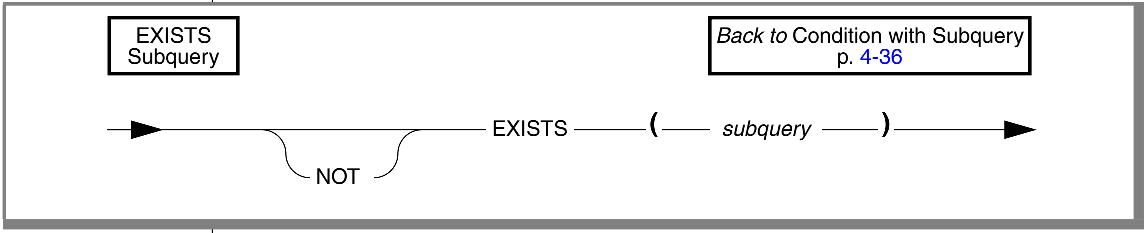
An IN subquery condition is TRUE if the value of the expression matches one or more of the values from the subquery. (The subquery must return only one column, but it can return more than one row.) The keyword IN is equivalent to the =ANY specification. The keywords NOT IN are equivalent to the !=ALL specification. See the "ALL, ANY, and SOME Subqueries" on page 4-39.

The following example of an IN subquery finds the order numbers for orders that do not include baseball gloves (stock_num = 1):

```
WHERE order_num NOT IN
   (SELECT order_num FROM items WHERE stock_num = 1)
```

Because the IN subquery tests for the presence of rows, duplicate rows in the subquery results do not affect the results of the main query. Therefore, the UNIQUE or DISTINCT keyword in the subquery has no effect on the query results, although not testing duplicates can improve query performance.

EXISTS Subquery



Element	Purpose	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST nor the ORDER BY clause.	SELECT, p. 2-581

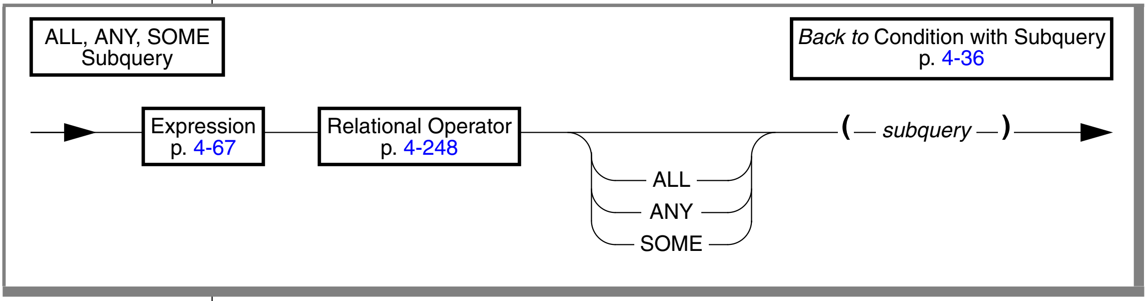
An EXISTS subquery condition evaluates to TRUE if the subquery returns a row. With an EXISTS subquery, one or more columns can be returned. The subquery always contains a reference to a column of the table in the main query. If you use an aggregate function in an EXISTS subquery that includes no HAVING clause, at least one row is always returned.

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table). You can appropriately use an EXISTS subquery in this SELECT statement because you use the subquery to test both **stock_num** and **manu_code** in **items**.

```
SELECT stock_num, manu_code FROM stock
   WHERE NOT EXISTS (SELECT stock_num, manu_code FROM items
      WHERE stock.stock_num = items.stock_num AND
      stock.manu_code = items.manu_code)
```

The preceding example works equally well if you use SELECT * in the subquery in place of the column names, because the existence of the entire row is tested; specific column values are not tested.

ALL, ANY, and SOME Subqueries


ALL, ANY, SOME Subquery
Back to Condition with Subquery p. 4-36
Expression p. 4-67
Relational Operator p. 4-248
ALL
ANY
SOME
(*subquery*)

Element	Purpose	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST or the ORDER BY clause.	SELECT, p. 2-581

Use the ALL, ANY, and SOME keywords to specify what makes the condition `TRUE` or `FALSE`. A search condition that is `TRUE` when the ANY keyword is used might not be `TRUE` when the ALL keyword is used, and vice versa.

Using the ALL Keyword

The ALL keyword specifies that the search condition is `TRUE` if the comparison is `TRUE` for every value that the subquery returns. If the subquery returns no value, the condition is `TRUE`.

In the following example, the first condition tests whether each **total_price** is greater than the total price of every item in order number 1023. The second condition uses the **MAX** aggregate function to produce the same results.

```
total_price > ALL (SELECT total_price FROM items
               WHERE order_num = 1023)

total_price > (SELECT MAX(total_price) FROM items
               WHERE order_num = 1023)
```

Using the NOT keyword with an ALL subquery tests whether an expression is not TRUE for at least one element that the subquery returns. For example, the following condition is TRUE when the expression **total_price** is not greater than all the selected values. That is, it is TRUE when **total_price** is not greater than the highest total price in order number 1023.

```
NOT total_price > ALL (SELECT total_price FROM items
                  WHERE order_num = 1023)
```

Using the ANY or SOME Keywords

The ANY keyword denotes that the search condition is TRUE if the comparison is TRUE for at least one of the values that is returned. If the subquery returns no value, the search condition is FALSE. The SOME keyword is a synonym for ANY.

The following conditions are TRUE when the total price is greater than the total price of at least one of the items in order number 1023. The first condition uses the ANY keyword; the second uses the MIN aggregate function.

```
total_price > ANY (SELECT total_price FROM items
               WHERE order_num = 1023)

total_price > (SELECT MIN(total_price) FROM items
               WHERE order_num = 1023)
```

Using the NOT keyword with an ANY subquery tests whether an expression is not TRUE for all elements that the subquery returns. For example, the following condition is TRUE when the expression **total_price** is not greater than any selected value. That is, it is TRUE when **total_price** is greater than none of the total prices in order number 1023.

```
NOT total_price > ANY (SELECT total_price FROM items
                  WHERE order_num = 1023)
```

Omitting the ANY, ALL, or SOME Keywords

You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery will return exactly one value. If you omit the ANY, ALL, or SOME keywords, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
   WHERE stock_num = 9 AND quantity =
      (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

NOT Operator

If you preface a condition with the keyword NOT, the test is `TRUE` only if the condition that NOT qualifies is `FALSE`. If the condition that NOT qualifies is `UNKNOWN` (that is, has a NULL value), the NOT operator has no effect.

The following truth table shows the effect of NOT. Here T represents a `TRUE` condition, F represents a `FALSE` condition, and a question mark (`?`) represents an `UNKNOWN` condition. (An `UNKNOWN` value can occur when an operand is NULL).

NOT	
T	F
F	T
?	?

The left column shows the value of the operand of the NULL operator, and the right column shows the returned value after NOT is applied to the operand.

Conditions with AND or OR

You can combine simple conditions with the logical operators AND or OR to form complex conditions. The following SELECT statements contain examples of complex conditions in their WHERE clauses:

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date > '1/1/97' OR paid_date IS NULL

SELECT order_num, total_price FROM items
   WHERE total_price > 200.00 AND manu_code LIKE 'H%'

SELECT lname, customer_num FROM customer
   WHERE zipcode BETWEEN '93500' AND '95700'
   OR state NOT IN ('CA', 'WA', 'OR')
```

The following truth tables show the effect of the AND and OR operators.The letter T represents a TRUE condition, F represents a FALSE condition, and the question mark (?) represents an UNKNOWN value. An UNKNOWN value can occur when part of an expression that uses a logical operator is NULL.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

The marginal values at the left represent the first operand, and values in the top row represent the second operand. Values within each 3x3 matrix shows the returned value after the operator is applied to those operands.

If the Boolean expression evaluates to UNKNOWN, the condition is not satisfied.

Consider the following example within a WHERE clause:

```
WHERE ship_charge/ship_weight < 5
   AND order_num = 1023
```

The row where **order_num** = 1023 is the row where **ship_weight** is NULL. Because **ship_weight** is NULL, **ship_charge**/**ship_weight** is also NULL; therefore, the truth value of **ship_charge**/**ship_weight** < 5 is UNKNOWN. Because **order_num** = 1023 is TRUE, the AND table states that the truth value of the entire condition is UNKNOWN. Consequently, that row is not chosen. If the condition used an OR in place of the AND, the condition would be TRUE.

Related Information

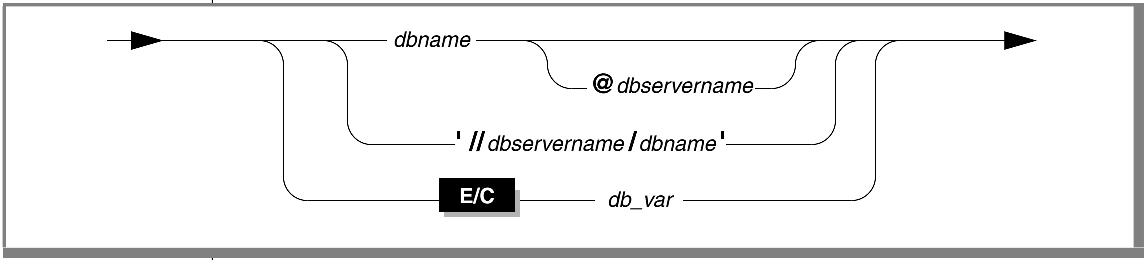
For discussions of comparison conditions in the SELECT statement and of conditions with a subquery, see the *IBM Informix Guide to SQL: Tutorial*.

For information on the GLS aspects of conditions, see the *IBM Informix GLS User's Guide*.

Database Name

Use the Database Name segment to specify the name of a database. Use this segment when you see a reference to a database name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dbname	Database name (with no pathname nor database server name)	Must be unique among database names on the database server.	Identifier, p. 4-189
dbservername	Database server on which the database *dbname* resides	Must exist. No blank space can separate @ from *dbservername*.	Identifier, p. 4-189
db_var	Host variable whose value specifies a database environment	Variable must be a fixed-length character data type.	Language specific

Usage

Database names are not case sensitive. You cannot use delimited identifiers for a database name.

The identifiers *dbname* and *dbservername* can each have a maximum of 128 bytes.

GLS

In a nondefault locale, *dbname* can include alphabetic characters from the code set of the locale. In a locale that supports a multibyte code set, keep in mind that the maximum length of the database name refers to the number of bytes, not the number of characters. For more information on the GLS aspects of naming databases, see the *IBM Informix GLS User's Guide*. ♦

Specifying the Database Server

You can choose a database on another database server as your current database by specifying a database server name. The database server that *dbservername* specifies must match the name of a database server that is listed in your **sqlhosts** information.

Using the @ Symbol

The @ symbol is a literal character. If you specify a database server name, do not put any blank spaces between the @ symbol and the database server name. Either put a blank space between *dbname* and the @ symbol, or omit the blank space.

The following examples show valid database specifications, qualified by the database server name:

```
empinfo@personnel
empinfo @personnel
```

In these examples, **empinfo** is the name of the database and **personnel** is the name of the database server.

Using a Path-Type Naming Notation

If you specify a pathname, do not put blank spaces between the quotes, slashes, and names. The following example includes a valid pathname:

```
'//personnel/empinfo'
```

Here **empinfo** is the *dbname* and **personnel** is the name of the database server.

E/C

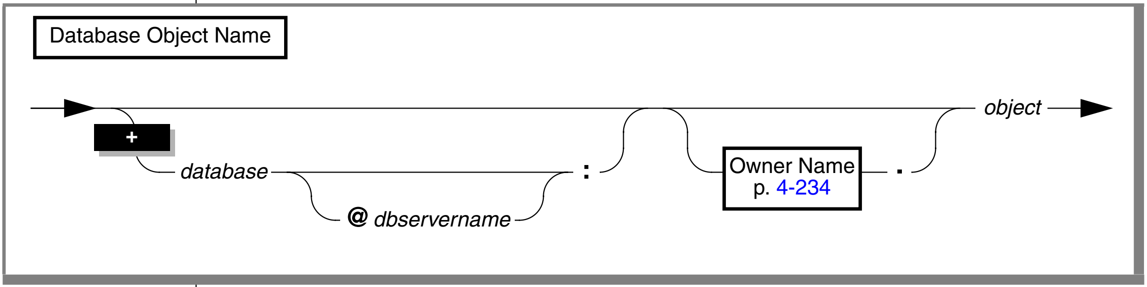
Using a Host Variable

You can use a host variable within an ESQL/C application to store a value that represents a database environment.

Database Object Name

Use the Database Object Name segment to specify the name of a database object, such as a constraint, index, opaque type, operator class, table, trigger, sequence, synonym, user-defined routine (UDR), or view. Use this segment whenever you see a reference to a database object name.

Syntax



Element	Purpose	Restrictions	Syntax
database	Database where *object* resides	Must exist.	Identifier, p. 4-189
dbservername	Database server of *database*	Must exist. No blank space after @.	Identifier, p. 4-189
object	Name of the database object	See "Usage."	Identifier, p. 4-189

Usage

If you are creating or renaming a database object, the name that you specify must be unique in relation to other database objects of the same type in the database. For example, a new constraint name must be unique among constraint names that exist in the database, and a new name or synonym for a table or view must be unique among the names and synonyms of tables and views, and temporary tables that already exist in the database.

ANSI

In an ANSI-compliant database, the *ownername***.***object* combination must be unique in a database. A database object specification must include the owner name for a database object that you do not own. For example, if you specify a table that you do not own, you must also specify the owner of the table. The owner of all the system catalog tables is **informix**. ♦

IDS

In Dynamic Server, the uniqueness requirement does not apply to user defined routine (UDR) names. For more information, see "Routine Overloading and Naming UDRs with a Routine Signature" on page 4-48. ♦

GLS

Characters from the code set of your locale are valid in database object names. For more information, see *IBM Informix GLS User's Guide*. ♦

Specifying a Database Object in an External Database

You can specify a database object in either an external database on the local database server or in an external database on a remote database server.

Specifying a Database Object in a Cross-Database Query

To specify an object in another database of the local database server, you must qualify the identifier of the object with the name of the database (and of the owner, if the external database is ANSI compliant), as in this example:

```
corp_db:hrdirector.executives
```

In this example, the name of the external database is **corp_db**. The name of the owner of the table is **hrdirector**. The name of the table is **executives**. Here the colon (**:**) separator is required after the *database* qualifier.

Specifying a Database Object in a Cross-Server Query

To specify an object in an external database on a remote database server, you must use a *fully-qualified identifier that* includes the names of the database, database server, and owner (if the external database is ANSI compliant) in addition to the database object name. This is a fully-qualified table name:

```
hr_db@remoteoffice:hrmanager.employees
```

Here the database name is **hr_db**, the database server is **remoteoffice**, the table owner is **hrmanager**, and the table name is **employees**. The at (**@**) separator is required between the *database* and *database server* qualifiers. Cross-server queries cannot access columns that have opaque data types.

IDS

If a UDR exists on a remote database server, you must specify a fully-qualified identifier for the UDR. A remote UDR is limited to built-in data types for its arguments, parameters, and returned values; these can include opaque , DISTINCT, collection, and ROW data types.

You can refer to a remote database object in the following statements only.

CREATE DATABASE
CREATE SYNONYM
CREATE VIEW
DATABASE
DELETE
EXECUTE FUNCTION
EXECUTE PROCEDURE
INFO
INSERT
LOAD
LOCK TABLE
SELECT
UNLOAD
UNLOCK TABLE
UPDATE

For information on the support in these statements across databases or across database servers, refer to the *IBM Informix Guide to SQL: Tutorial*. ♦

IDS

Routine Overloading and Naming UDRs with a Routine Signature

Because of routine overloading, the name of a UDR (that is, a user-defined function or a user-defined procedure) does not have to be unique to the database. You can define more than one UDR with the same name as long as the *routine signature* for each UDR is different.

UDRs are uniquely identified by their signatures. The signature of a UDR includes the following items of information:

- The type of routine (function or procedure)
- The identifier of the routine
- The quantity, data type, and order of the parameters
- In an ANSI-compliant database, the owner name ♦

ANSI

For any given UDR, at least one item in the signature must be unique among all the UDRs stored in a name space or database.

Specifying an Existing UDR

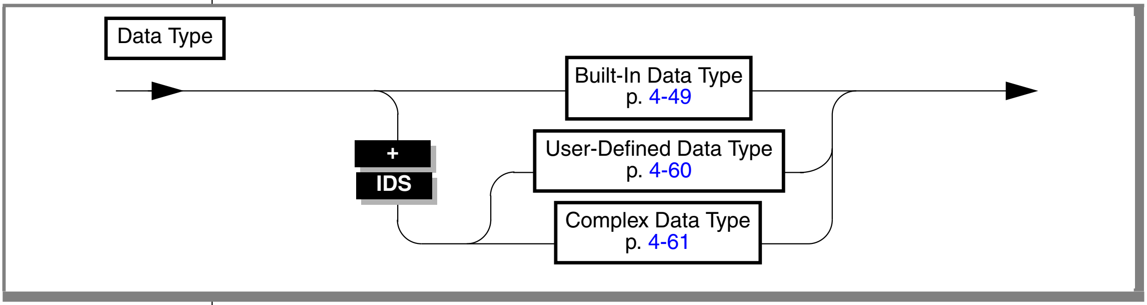
When you are specifying the name of an existing UDR, if the name you specify does not uniquely identify the UDR, you must also specify the parameter data types after the UDR name in the same order that they were specified when the UDR was created. The database server then uses routine resolution to identify the instance of the UDR to alter, drop, or execute. As an alternative, you can specify the specific name for the UDR if one was declared when the UDR was created.

For more information about routine resolution, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Data Type

The Data Type segment specifies the data type of a column or value. Use this segment whenever you see a reference to a data type in a syntax diagram.

Syntax

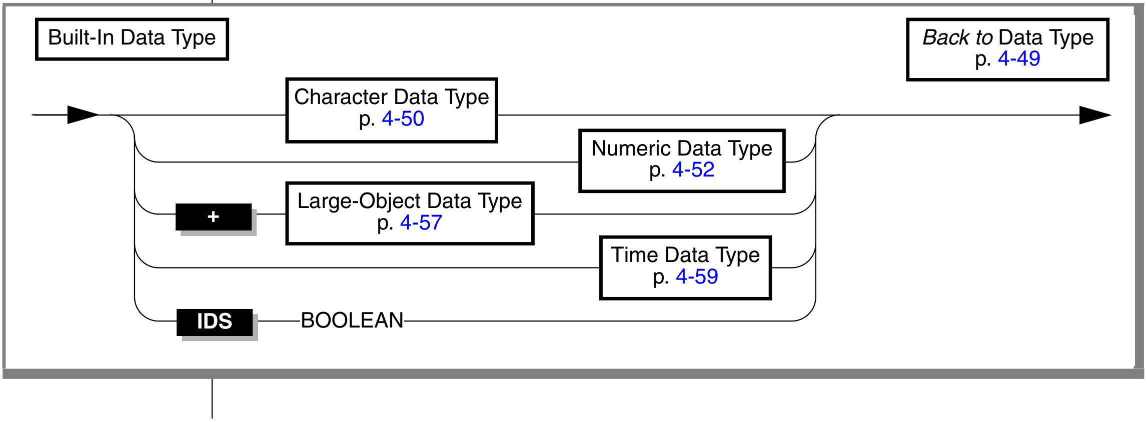

Data Type
Built-In Data Type p. 4-49
+
IDS
User-Defined Data Type p. 4-60
Complex Data Type p. 4-61

Usage

Sections that follow summarize these data types. For more information, see the chapter about data types in the *IBM Informix Guide to SQL: Reference*.

Built-In Data Types

Built-in data types are data types that are defined by the database server.


Built-In Data Type
Back to Data Type p. 4-49
Character Data Type p. 4-50
Numeric Data Type p. 4-52
+
Large-Object Data Type p. 4-57
Time Data Type p. 4-59
IDS
BOOLEAN

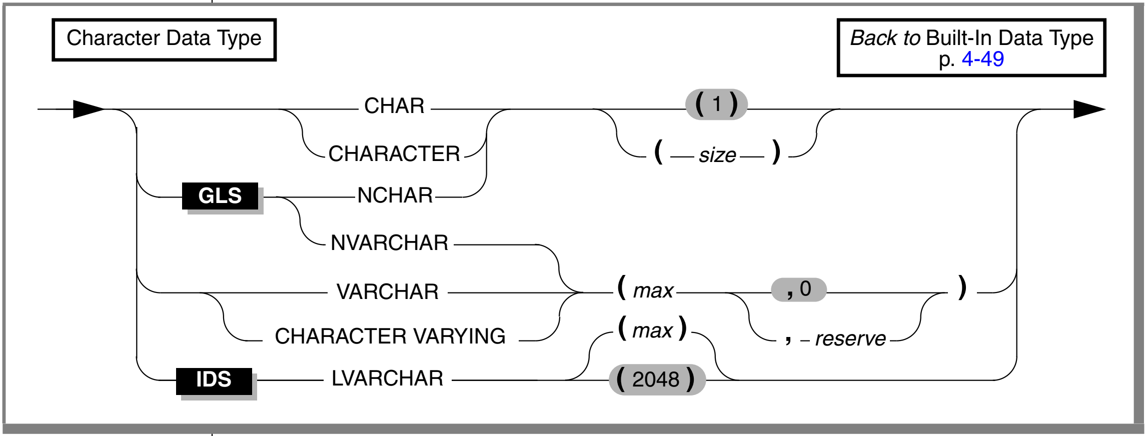
These are "built into the database server" in the sense that the information required to interpret and transfer these data types is part of the database server software, which supports *character*, *numeric*, *large-object*, and *time* categories of built-in data types. These are described in sections that follow.

IDS

Dynamic Server also supports the BOOLEAN data type. The BOOLEAN type is a built-in opaque data type that can store `TRUE`, `FALSE`, or NULL values. The symbol `t` represents BOOLEAN `TRUE` and `f` represents BOOLEAN `FALSE`.

Like other opaque data types, BOOLEAN column values cannot be retrieved by a distributed query of a remote database. ♦

Character Data Types



Element	Purpose	Restrictions	Syntax
max	Maximum size in bytes. For VARCHAR, this is mandatory. For LVARCHAR, default is 2048	VARCHAR: Integer; 1 ≤ *max* ≤ 255 (or 1 ≤ *max* ≤ 254, if indexed) LVARCHAR: 1 ≤ *max* ≤ 32,739	Literal Number, p. 4-216
reserve	Bytes reserved. Default is 0.	Integer; 0 ≤ *reserve* ≤ *max*	Literal Number, p. 4-216
size	Size in bytes. Default is 1.	Integer; 1 ≤ *size* ≤ 32,767	Literal Number, p. 4-216

The database server issues an error if the data type declaration includes empty parentheses, such as LVARCHAR( ). To declare a CHAR or LVARCHAR data type of the default length, simply omit any *(size)* or *(max)* specification.

The following table summarizes the built-in character data types.

Data Type	Description
CHAR	Stores single-byte or multibyte text strings of up to 32,767 bytes of text data; supports code-set collation of text data. Default is 1 byte.
CHARACTER	Synonym for CHAR
CHARACTER VARYING	ANSI-compliant synonym for VARCHAR
LVARCHAR (IDS)	Stores variable-length strings of up to 32,739 bytes. (Size of other columns in the same table can further reduce this upper limit. Default is 2,048 bytes, if no explicit maximum size is specified.)
NCHAR	Stores single-byte or multibyte text strings of up to 32,767 bytes in length; supports localized collation of text data
NVARCHAR	Stores single-byte or multibyte text strings of varying length (up to 255 bytes); supports localized collation of text data
VARCHAR	Stores single-byte or multibyte text strings of varying length up to 255 bytes); supports code-set order collation of text data

The TEXT and CLOB data types also support character data. For more information, see "Large-Object Data Types" on page 4-57.

Fixed- and Varying-Length Character Data Types

The database server supports storage of fixed-length and varying-length character data. A *fixed-length* column requires the defined number of bytes regardless of the actual size of the character data. The CHAR data type is of fixed-length. For example, a CHAR(25) column requires 25 bytes of storage for all values, so the string `"This is a text string"` uses 25 bytes of storage.

A *varying-length* column size can be the number of bytes occupied by its data. NVARCHAR, VARCHAR, and (for Dynamic Server only) the LVARCHAR data types are varying-length character data types. For example, a VARCHAR(25) column reserves up to 25 bytes of storage for the column value, but the character string `"This is a text string"` uses only 21 bytes of the reserved 25 bytes. The VARCHAR data type can store up to 255 bytes of data.

IDS

The LVARCHAR data type can store up to 32,739 bytes of text data. ♦

IDS

The LVARCHAR data type is a built-in opaque type. Like other opaque types, it cannot be retrieved by a distributed query of a remote database.

A single table cannot be created with more than 195 LVARCHAR columns. (The same restriction applies to all varying-length and ROW data types.) ♦

IDS

Light scans are not supported on tables having VARCHAR data types. ♦

NCHAR and NVARCHAR Data Types

The character data types CHAR, LVARCHAR, and VARCHAR support code-set order collation of data. The database server collates text data in columns of these types by the order that their characters are defined in the code set.

To accommodate locale-specific order of characters, use the NCHAR and NVARCHAR data types. The NCHAR data type is the fixed-length character data type that supports localized collation. The NVARCHAR data type is the varying-length character data type that can store up to 255 bytes of text data and supports localized collation. A single table cannot be created with more than 195 NVARCHAR or VARCHAR columns.

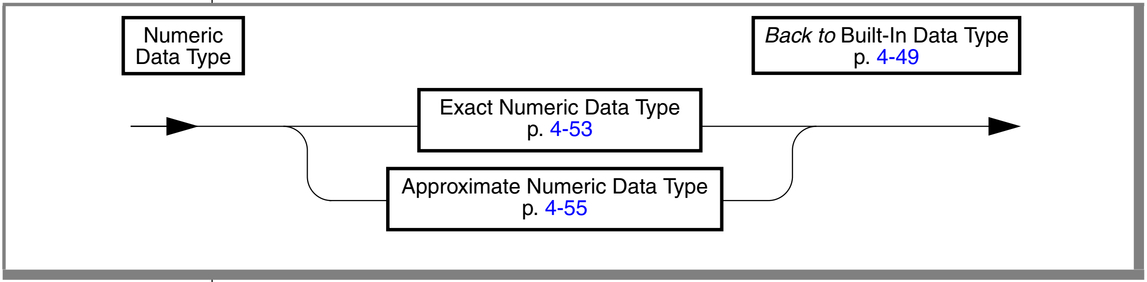
IDS

Light scans are not supported on tables having NVARCHAR data types. ♦

For more information, see the *IBM Informix GLS User's Guide*.

Numeric Data Types

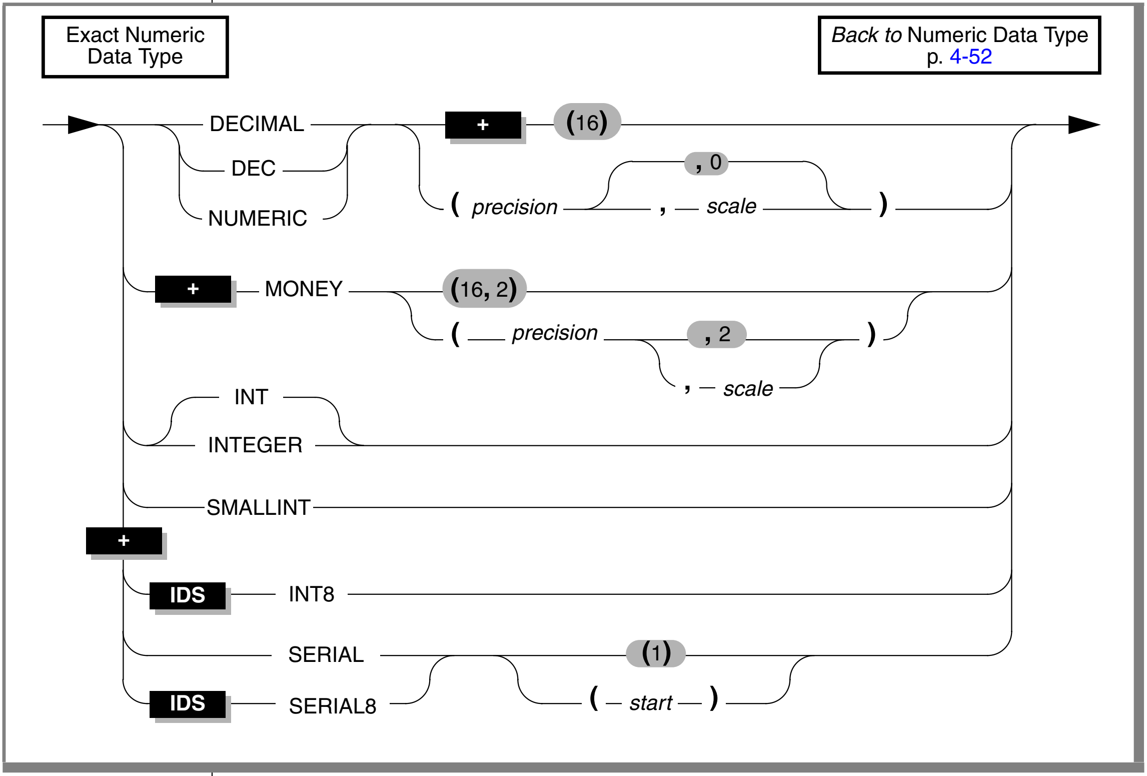
Numeric data types allow the database server to store numbers such as integers and real numbers in a column.


Numeric Data Type
Back to Built-In Data Type p. 4-49
Exact Numeric Data Type p. 4-53
Approximate Numeric Data Type p. 4-55

The values of numbers are stored either as *exact numeric* data types or as *approximate numeric* data types.

Exact Numeric Data Types

An *exact numeric* data type stores numbers of a specified precision and scale.



Element	Purpose	Restrictions	Syntax
precision	Significant digits	An integer; 1 ≤ *precision* ≤ 32	Literal Number, p. 4-216
scale	Digits in fractional part	An integer; 1 ≤ *scale* ≤ *precision*	Literal Number, p. 4-216
start	Integer starting value	For SERIAL: 0 ≤ *start* ≤ 2,147,483,64; For SERIAL8: 0 ≤ *start* ≤ 9,223,372,036,854,775,807	Literal Number, p. 4-216

The *precision* of a data type is the number of digits that the data type stores. The *scale* is the number of digits to the right of the decimal separator.

The following table summarizes the exact numeric data types available.

Data Type	Description
DEC(*p,s*)	Synonym for DECIMAL(p,s)
DECIMAL(*p,s*)	Stores fixed-point decimal values of real numbers, with up to 30 significant digits in the fractional part, or up to 32 significant digits to the left of the decimal point.
INT	Synonym for INTEGER
INTEGER	Stores a 4-byte integer value. These values can be in the range -((2**31)-1) to (2**31)-1 (the range -2,147,483,647 to 2,147,483,647).
INT8 (IDS)	Stores an 8-byte integer value. These values can be in the range -((2**63)-1) to (2**63)-1 (the range -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807).
MONEY(*p,s*)	Stores fixed-point currency values. These values have same internal data format as a fixed-point DECIMAL(p,s) value.
NUMERIC(*p,s*)	ANSI-compliant synonym for DECIMAL(p,s)
SERIAL	Stores a 4-byte integer value that the database server generates. These values can be in the range -((2**31)-1) to (2**31)-1 (the values -2,147,483,647 to 2,147,483,647).
SERIAL8 (IDS)	Stores an 8-byte integer value that the database server generates. These values can be in the range -((2**63)-1) to (2**63)-1 (the values -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807).
SMALLINT	Stores a 2-byte integer value. These values can be in the range -((2**15)-1) to (2**15)-1 (that is, from -32,767 to 32,767).

Notes on DECIMAL(p,s) Data Types

The first DECIMAL(*p, s*) parameter (*p*) specifies the *precision* and the second (*s*) parameter specifies the *scale*. If you provide only one parameter, an ANSI-compliant database interprets it as the precision of a fixed-point number and the default scale is 0. If you specify no parameters, and the database is ANSI-compliant, then the default precision is 16, and the default scale is 0.

If the database is not ANSI-compliant, specifying fewer than 2 parameters, declares a floating-point DECIMAL, which is not an exact number data type.

DECIMAL(p,s) values are stored internally with the first byte representing a sign bit and a 7-bit exponent in excess-65 format. The other bytes express the mantissa as base-100 digits. This implies that DECIMAL(32,s) data types store only *s*-1 decimal digits to the right of the decimal point, if *s* is an odd number.

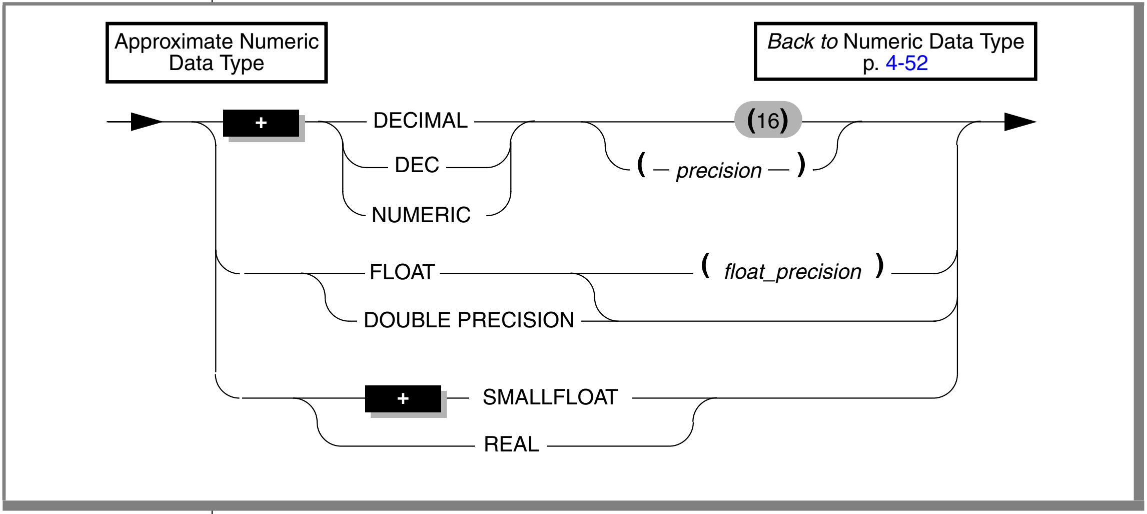
Notes on SERIAL and SERIAL8E Data Types

If you want to insert an explicit value into a SERIAL or SERIAL8 column, you can use any nonzero number. You cannot, however, start or reset the value of a SERIAL or SERIAL8 column with a negative number.

A SERIAL or SERIAL8 column is not unique unless you set a unique index on the column. (The index can also be in the form of a primary key or unique constraint.) With such an index, values in SERIAL OR SERIAL8 columns are guaranteed to be unique but not contiguous.

Approximate Numeric Data Types

An *approximate numeric* data type represents numeric values approximately.



Element	Purpose	Restrictions	Syntax
float_precision	The *float_precision* is ignored, but is ANSI/ISO compliant	Must be a positive integer. Specified value has no effect.	Literal Number, p. 4-216
precision	Significant digits. Default is 16.	An integer; 1 ≤ *precision* ≤ 32	Literal Number, p. 4-216

Use approximate numeric data types for very large and very small numbers that can tolerate some degree of rounding during arithmetic operations.

This table summarizes the approximate numeric data types.

Data Type	Description
DEC(*p*)	Synonym for DECIMAL(p)
DECIMAL(*p*)	Stores floating-point decimal values in the approximate range from 1.0E-130 to 9.99E+126 The *p* parameter specifies the precision. If no precision is specified, the default is 16. This floating-point data type is available as an approximate numeric type only in a database that is not ANSI-compliant. In an ANSI-compliant database, DECIMAL(*p*) is implemented as a fixed-point DECIMAL; see "Exact Numeric Data Types" on page 4-53.
DOUBLE PRECISION	ANSI-compliant synonym for FLOAT. The *float_precision* term is not valid when you use this synonym in data type declarations.
FLOAT	Stores double-precision floating-point numbers with up to 16 significant digits. The *float-precision* parameter is accepted in data-type declarations for compliance with the ANSI/ISO standard for SQL, but this parameter has no effect on the actual precision of values that the database server stores.
NUMERIC(*p*)	ANSI-compliant synonym for DECIMAL(*p*) In an ANSI-compliant database, this is implemented as an exact numeric type, with the specified precision and a scale of zero, rather than an approximate numeric (floating-point) data type.
REAL	ANSI-compliant synonym for SMALLFLOAT
SMALL-FLOAT	Stores single-precision floating-point numbers with approximately 8 significant digits

The built-in data types of Informix database servers support real numbers. They cannot directly store imaginary or complex numbers.

IDS

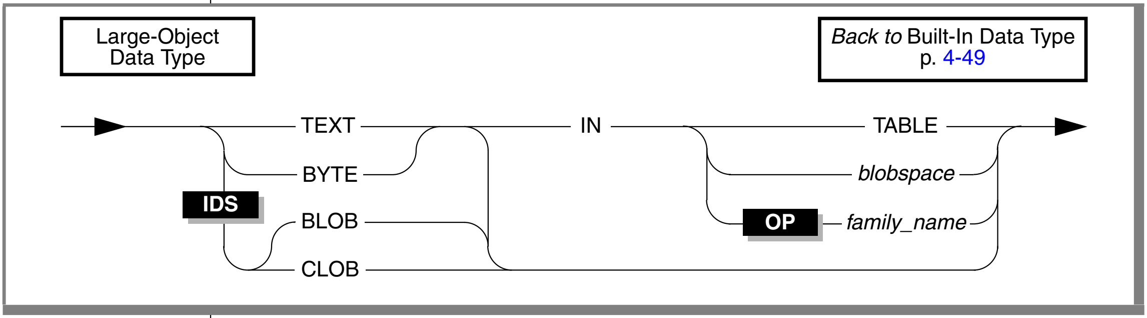
You must create a user-defined data type for applications that support values that can have an imaginary part. ♦

Ext

No more than nine arguments to a UDR can be DECIMAL data types of SQL that the UDR declares as BigDecimal data types of the Java language. ♦

Large-Object Data Types

Large-object data types can store extremely large column values, such as images and documents, independently of the column.



Element	Purpose	Restrictions	Syntax
blobspace	Name of an existing blobspace	Must exist.	Identifier, p. 4-189
family_name	Family name or variable in the optical family	Must exist.	Quoted String, p. 4-243.

The large object data types can be classified in two categories:

- Simple large objects: TEXT and BYTE
- Smart large objects: CLOB and BLOB

Simple-Large-Object Data Types

These are the simple-large-object data types:

Data Type	Description
TEXT	Stores text data of up to 2^{31} bytes
BYTE	Stores any digitized data of up to 2^{31} bytes

These data types are not recoverable. Do not supply a BYTE value where TEXT is expected. No built-in cast supports BYTE to TEXT data-type conversion.

You cannot create a table with more than 195 BYTE or TEXT columns. (The same restriction applies to all varying-length and ROW data types.)

Storing BYTE and TEXT Data

A simple-large-object data type stores text or binary data in blobspaces or in tables. The database server can access a BYTE or TEXT value in one piece. When you specify a BYTE or TEXT data type, you can specify the location in which it is stored. You can store data with the table or in a separate blobspace.

IDS

If you are creating a named ROW data type that includes a BYTE or TEXT column, you cannot use the IN clause to specify a separate storage space. ♦

This example shows how blobspaces and dbspaces are specified. The user creates the **resume** table. The data values are stored in the **employ** dbspace. The data in the **vita** column is stored with the table, but the data associated with the **photo** column is stored in a blobspace named **photo_space**.

```
CREATE TABLE resume
   (
   fname          CHAR(15),
   lname          CHAR(15),
   phone          CHAR(18),
   recd_date      DATETIME YEAR TO HOUR,
   contact_date   DATETIME YEAR TO HOUR,
   comments       VARCHAR(250, 100),
   vita           TEXT IN TABLE,
   photo          BYTE IN photo_space
   )
   IN employ
```

IDS

Smart-Large-Object Data Types

A smart-large-object data type stores text or binary data in sbspaces. The database server can provide random access to a smart-large-object value. That is, it can access any portion of the smart-large-object value. These data types are recoverable. The following table summarizes the smart-large-object data types that Dynamic Server supports.

Data Type	Description
BLOB	Stores binary data of up to 4 terabytes ($4*2^{40}$ bytes)
CLOB	Stores text data of up to 4 terabytes ($4*2^{40}$ bytes)

For more information, see the *IBM Informix Guide to SQL: Reference*.

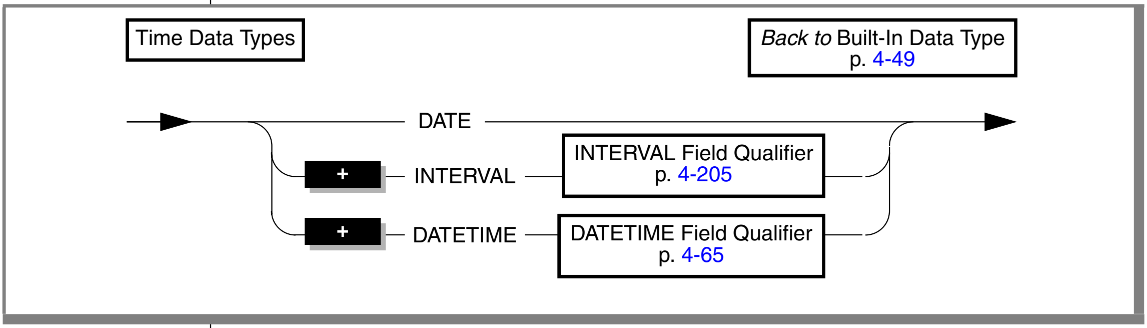
For information on how to create blobspaces, see your *Administrator's Guide*.

For information about optical families, see the *IBM Informix Optical Subsystem Guide*.

For information about the built-in functions that you can use to import, export, and copy smart large objects, see "Smart-Large-Object Functions" on page 4-134 and the *IBM Informix Guide to SQL: Tutorial*.

Time Data Types

The time data types allow the database server to store increments of time.


Time Data Types
Back to Built-In Data Type p. 4-49
DATE
+
INTERVAL
INTERVAL Field Qualifier p. 4-205
+
DATETIME
DATETIME Field Qualifier p. 4-65

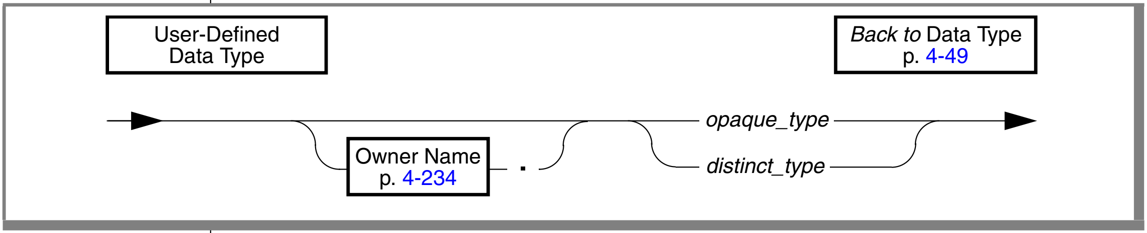
The following table summarizes the built-in time data types.

Data Type	Description
DATE	Stores a date value as a Julian date in the range from January 1 of the year 1 up to December 31, *9999*.
DATETIME	Stores a point-in-time date (*year*, *month*, *day*) and time-of-day (*hour*, *minute*, *second*, and *fraction* of second), in the range of years 1 to 9999. Also supports contiguous subsets of these time units.
INTERVAL	Stores spans of time, in *years* and/or *months*, or in smaller time units (*days*, *hours*, *minutes*, *seconds*, and/or *fractions* of second), with up to 9 digits of precision in the largest time unit, if this is not FRACTION Also supports contiguous subsets of these time units.

IDS

User-Defined Data Type

A user-defined data type is one that a user defines for the database server. Dynamic Server supports two categories of user-defined data types, namely-*distinct data types* and *opaque data types*. This is the declaration syntax for user-defined data types:



Element	Purpose	Restrictions	Syntax
distinct_type	Distinct data type with same structure as an existing data type	Must be unique among data type names in the database.	Identifier, p. 4-189
opaque_type	Name of the opaque data type	Must be unique among data type names in the database.	Identifier, p. 4-189

In this manual, *user-defined data type* is usually abbreviated as UDT.

Distinct Data Types

A distinct data type is a user-defined data type that is based on an existing built-in type, opaque type, named row type, or distinct type. To create a distinct type, you must use the CREATE DISTINCT TYPE statement. (For more information, see"CREATE DISTINCT TYPE" on page 2-115.)

Opaque Data Types

An opaque data type is a user-defined data type that can be used in the same way as a built-in data type. To create an opaque type, you must use the CREATE OPAQUE TYPE statement. Because an opaque type is encapsulated, you create support functions to access the individual components of an opaque type. The internal storage details of the type are hidden or opaque.

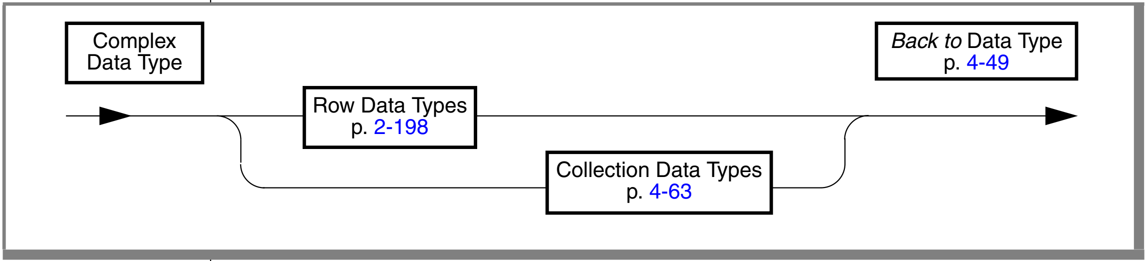
For more information about how to create an opaque data type and its support functions, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Because of the maximum row size limit of 32,767 bytes, when you create a new table, no more than 195 columns can be varying-length opaque or distinct user-defined data types. (The same restriction applies to BYTE, TEXT, VARCHAR, LVARCHAR, NVARCHAR, and ROW columns. See "Row Data Types" on page 4-62 for additional information about ROW data types.)

IDS

Complex Data Type

Complex data types are ROW types or COLLECTION types that you create from built-in types, opaque types, distinct types, or other complex types.



A single complex data type can include multiple components. When you create a complex type, you define the components of the complex type. Unlike an opaque type, however, a complex type is not encapsulated. You can use SQL to access the individual components of a complex data type. The individual components of a complex data type are called *elements*.

Dynamic Server supports the following categories of complex data types:

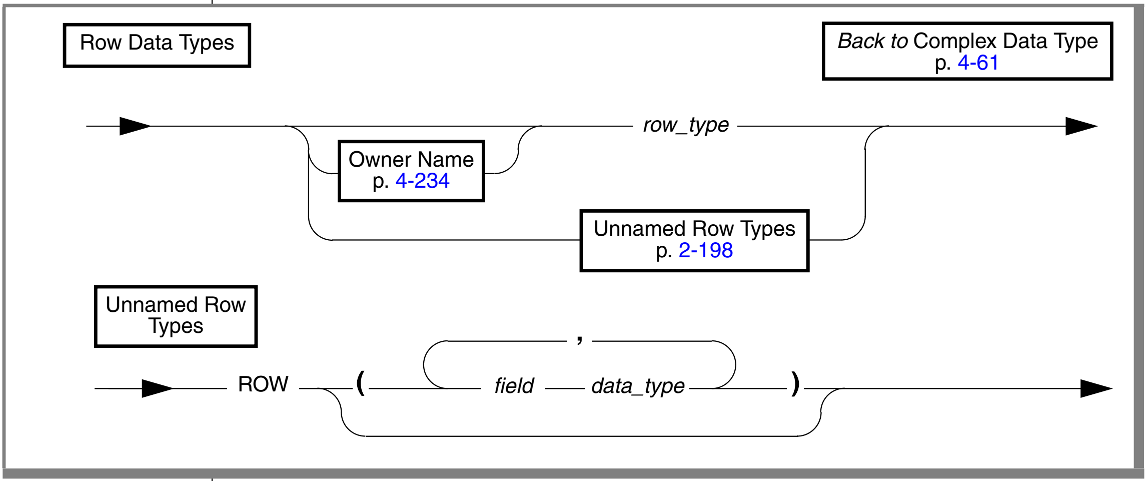
- ROW types: Named ROW types and unnamed ROW types
- COLLECTION data types: SET, MULTISET, and LIST

The elements of a COLLECTION data type must all be of the same data type. You can use the keyword COLLECTION in SPL data type declarations to specify an untyped collection variable. NULL values are not supported in elements of COLLECTION data types.

The elements of a ROW data type can be of different data types, but the pattern of data types from the first to the last element cannot vary for a given ROW data type. NULL values are supported in elements of ROW data types, unless you specify otherwise in the data type declaration or in a constraint.

Row Data Types

This is the syntax to define a column as a named or unnamed ROW type.



Element	Purpose	Restrictions	Syntax
data_type	Data type of *field*	Any data type except BYTE or TEXT	Data Type, p. 4-49
field	Field in the row	Name must be unique among fields of the same ROW type	Identifier, p. 4-189
row_type	Name of ROW data type created by the CREATE ROW TYPE statement	ROW type must exist	Identifier, p. 4-189; Data type, p. 4-49

You can assign a named ROW type to a table, to a column, or to an SPL variable. A named ROW type that you use to create a typed table or to define a column must already exist. For information on how to create a named ROW data type, see "CREATE ROW TYPE" on page 2-198.

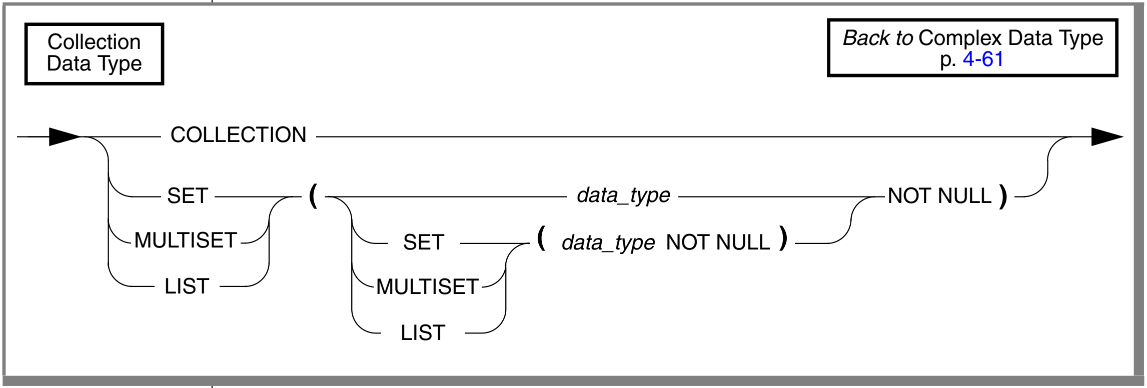
ANSI

To specify a named ROW data type in an ANSI-compliant database, you must specify the *owner* of *row_type*, if you are not the owner of *row_type*. ♦

An unnamed ROW data type is identified by its structure, which specifies fields that you create with its ROW constructor. You can define a column or an SPL variable as an unnamed ROW data type. For the syntax to specify values for an unnamed ROW type, see "ROW Constructors" on page 4-106.

Collection Data Types

This diagram shows the syntax to define a column or of an SPL variable as a collection data type. (A table can include no more than 97 columns of collection data types.) For the syntax to specify values of collection elements, see "Collection Constructors" on page 4-108.



Element	Purpose	Restrictions	Syntax
data_type	Data type of each of the elements of the collection	Can be any data type except TEXT, BYTE, SERIAL, or SERIAL8	Data Type, p. 4-49

A SET is an unordered collection of elements, each of which has a unique value. Define a column as a SET data type when you want to store collections whose elements contain no duplicate values and have no associated order.

A MULTISET is an unordered collection of elements in which elements can have duplicate values. You can define a column as a MULTISET collection type when you want to store collections whose elements might not be unique and have no specific order associated with them.

A LIST is an ordered collection of elements that allows duplicate elements. A LIST differs from a MULTISET in that each element in a LIST collection has an ordinal position in the collection. You can define a column as a LIST collection type when you want to store collections whose elements might not be unique but have a specific order associated with them.

The keyword COLLECTION can be used in SPL data type declarations to specify an untyped collection variable.

Defining the Element Type

The element type can be any data type except TEXT, BYTE, SERIAL, or SERIAL8. You can nest collection types, using elements of a collection type.

Every element must be of the same type. For example, if the element type of a collection data type is INTEGER, every element must be of type INTEGER.

An exception to this restriction occurs if the database server determines that some elements of a collection of character strings are VARCHAR data types (whose length is limited to 255 or fewer bytes) but other elements are longer than 255 bytes. In this case, the collection constructor can assign a CHAR(*n*) data type to all elements, for *n* the length in bytes of the longest element. If this is undesirable, you can cast the collection to LVARCHAR, to prevent padding extra length in elements of the collection, as in this example:

```
LIST {'first character string longer than 255 bytes . . . ',
   'second character string longer than 255 bytes . . . ',
   'another character string'} ::LIST (LVARCHAR NOT NULL)
```

See "Collection Constructors" on page 4-108 for additional information.

If the element type of a collection is an unnamed ROW type, the unnamed ROW type cannot contain fields that hold unnamed ROW types. That is, a collection cannot contain nested unnamed ROW types.

The elements of a collection cannot be NULL. When you define a column as a collection data type, you must use the NOT NULL keywords to specify that the elements of the collection cannot be NULL.

Privileges on a collection data type are those of the database column. You cannot specify privileges on individual elements of a collection.

Related Information

For more information about choosing a data type for your database, see the *IBM Informix Database Design and Implementation Guide*.

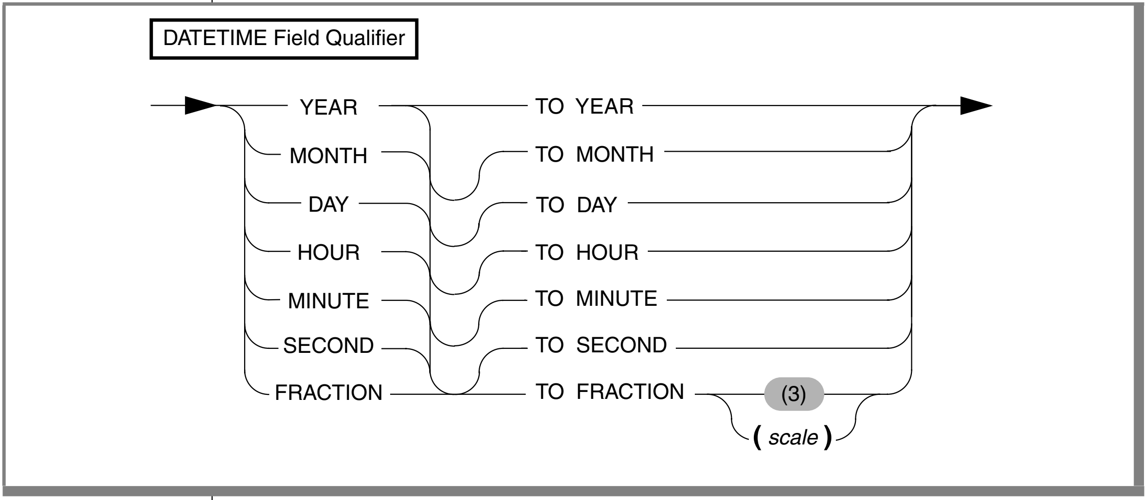
For more information about the specific qualities of individual data types, see the chapter on data types in the *IBM Informix Guide to SQL: Reference*.

For more information about multibyte data types, see the discussion of the NCHAR and NVARCHAR data types and the GLS aspects of other character data types in the *IBM Informix GLS User's Guide*.

DATETIME Field Qualifier

Use a DATETIME Field Qualifier to specify the largest and smallest unit of time in a DATETIME column or value. Use this segment whenever you see a reference to a DATETIME Field Qualifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
scale	Fraction of a second. The default is 3.	Integer (1 ≤ *scale* ≤ 5)	Literal Number, p. 4-216

Usage

This segment specifies the precision and scale of a DATETIME data type.

Specify as the first keyword the largest time unit that the DATETIME column will store. After the keyword TO, specify the smallest unit as the last keyword. These can be the same keyword. If they are different, the qualifier implies that any intermediate time units between the first and last are also recorded by the DATETIME data type.

The keywords can specify the following time units for the DATETIME column.

Unit of Time	Description
YEAR	Specifies a year, in the range from A.D. 1 to 9999
MONTH	Specifies a month, in the range from 1 (January) to 12 (December)
DAY	Specifies a day, in the range from 1 to 28, 29, 30, or 31 (depending on the specific month)
HOUR	Specifies an hour, in the range from 0 (midnight) to 23
MINUTE	Specifies a minute, in the range from 0 to 59
SECOND	Specifies a second, in the range from 0 to 59
FRACTION	Specifies a fraction of a second, with up to five decimal places The default scale is three digits (thousandth of a second).

Unlike INTERVAL qualifiers, DATETIME qualifiers cannot specify nondefault precision (except for FRACTION, when FRACTION is the smallest unit in the qualifier). Some examples of DATETIME qualifiers follow:

```
YEAR TO MINUTE             MONTH TO MONTH
DAY TO FRACTION(4)         MONTH TO DAY
```

An error results if the first keyword represents a smaller time unit than the last, or if you use the plural form of a keyword (such as MINUTES).

Operations on DATETIME values that do not include YEAR in their qualifier use values from the system clock-calendar to supply any additional precision. If the first term in the qualifier is DAY, and the current month has fewer than 31 days, unexpected results can occur.

Related Information

For an explanation of the DATETIME Field Qualifier, see the discussion of the DATETIME data type in the *IBM Informix Guide to SQL: Reference*.

For important differences between the syntax of DATETIME and INTERVAL field qualifiers, see "INTERVAL Field Qualifier" on page 4-205.

Expression

Data values in SQL statements must be represented as expressions. An *expression* is a specification, which can include operators, operands, and parentheses, that the database server can evaluate to one or more values, or to a reference to some database object.

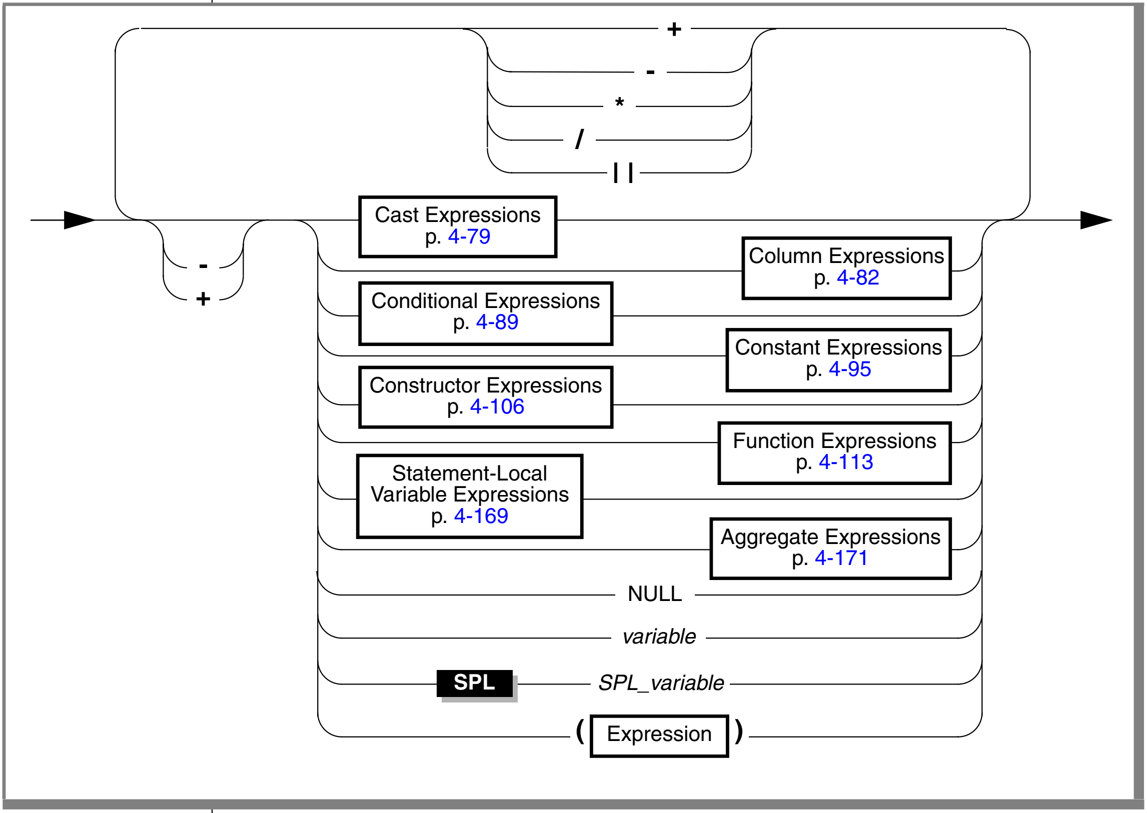
Expressions can refer to values already in a table of the database, or to values derived from such data, but some expressions (such as TODAY, CURRENT, or literal values) can return values that are independent of the database. You can use expressions to specify values in data-manipulation statements, to define fragmentation strategies, and in other contexts. Use the Expression segment whenever you see a reference to an expression in a syntax diagram.

In most contexts, however, you are restricted to expressions whose returned value is of some specific data type, or of a data type that can be converted by the database server to some required data type.

For an alphabetical listing of the built-in operators and functions that are described in this segment, see "List of Expressions" on page 4-69.

Syntax

This segment describes SQL expressions. An *expression* is a specification that returns one or more values or references to database objects. IBM Informix database servers support the following types of expressions:



+
-
*
/
||
-
+
Cast Expressions p. 4-79
Column Expressions p. 4-82
Conditional Expressions p. 4-89
Constant Expressions p. 4-95
Constructor Expressions p. 4-106
Function Expressions p. 4-113
Statement-Local Variable Expressions p. 4-169
Aggregate Expressions p. 4-171
NULL
variable
SPL
SPL_variable
(
Expression
)

Element	Purpose	Restrictions	Syntax
SPL_variable	In an SPL routine, a variable that contains some expression type that the syntax diagram shows	Must conform to the rules for expressions of that type	Identifier, p. 4-189
variable	Host or program variable that contains some expression type that the syntax diagram shows	Must conform to the rules for expressions of that type	Language-specific rules for names

Usage

This table lists the different types of SQL expressions, as identified in the diagram for "Syntax" on page 4-68, and states the purpose of each type.

Expression Type	Purpose
Arithmetic operators	Supports arithmetic operations on one (unary operators) or two (binary operators) numeric operands
Concatenation operator	Concatenates two string values
Cast operators	Explicitly casts from one data type to another
Column expressions	Full or partial column values
Conditional expressions	Returns values that depend on conditional tests
Constant expressions	Literal values in DML statements
Constructor expressions	Dynamically creates values for complex data types
Function expressions	Returns values from built-in or user-defined functions
Statement-Local-Variable expressions	Specifies how you can use a defined statement-local variable (SLV) elsewhere in an SQL statement
Aggregate functions	Returns values from built-in or user-defined aggregates

You can also use host variables or SPL variables as expressions. For a complete list, see "List of Expressions" on the pages that follow.

List of Expressions

Each category of SQL expressions includes many individual expressions. The following table lists all the SQL expressions (and some operators) in alphabetical order. The columns in this table have the following meanings:

- **Name** gives the name of each expression.
- **Purpose** gives a short description of each expression.
- **Syntax** lists the page that shows the syntax of the expression.
- **Usage** shows the page that describes the usage of the expression.

Each expression listed in the following table is supported on all database servers unless otherwise noted. When an expression is not supported on all database servers, the **Name** column notes in parentheses the database server or servers that do support the expression.

Name	Purpose	Syntax	Usage
ABS function	Returns the absolute value of a given expression	p. 4-114	p. 4-115
ACOS function	Returns the arc cosine of a numeric expression	p. 4-149	p. 4-150
Addition (+) operator	Returns the sum of two numeric expression operands	p. 4-68	p. 4-77
ASIN function	Returns the arc sine of a numeric expression	p. 4-149	p. 4-151
ATAN function	Returns the arc tangent of a numeric expression	p. 4-149	p. 4-151
ATAN2 function	Computes the angular component of the polar coordinates (*r*, q) associated with (*x*, *y)*	p. 4-149	p. 4-151
AVG function	Returns the mean value of a set of numeric expressions	p. 4-171	p. 4-175
CARDINALITY function (IDS)	Returns the number of elements in a collection column (SET, MULTISET, or LIST)	p. 4-118	p. 4-118
CASE expression	Returns one of several possible results, depending on which of several conditional tests evaluates to true	p. 4-89	p. 4-89
CAST expression (IDS)	Casts an expression to another specified data type	p. 4-79	p. 4-79
Cast (**::**) operator	See double-colon cast operator.	p. 4-79	p. 4-79
CHARACTER_LENGTH function	See CHAR_LENGTH function.	p. 4-131	p. 4-132
CHAR_LENGTH function	Counts the logical characters (not bytes) in a string	p. 4-131	p. 4-132
Column expression	Complete or partial column value from a table	p. 4-82	p. 4-82
Concatenation (||) operator	Concatenates the results of two expressions	p. 4-68	p. 4-78
Constant expression	Expression that returns a literal, fixed, or variant value	p. 4-95	p. 4-95
COS function	Returns the cosine of a radian expression	p. 4-149	p. 4-150

(1 of 7)

Name	Purpose	Syntax	Usage
COUNT (as a set of functions)	Functions that return specified frequency counts Each form of the COUNT function is listed below.	p. 4-171	p. 4-175
COUNT (ALL *column*) function	See COUNT (*column*) function.	p. 4-171	p. 4-176
COUNT (*column*) function	Returns the number of non-NULL values in a column	p. 4-171	p. 4-176
COUNT DISTINCT function	Returns the number of unique non-NULL values in a specified column	p. 4-171	p. 4-176
COUNT UNIQUE function	See COUNT DISTINCT function.	p. 4-171	p. 4-176
COUNT (*) function	Returns the number of rows that satisfy a query	p. 4-171	p. 4-175
CURRENT operator)	Returns the current instant by returning a DATETIME value that consists of the date and the time of day	p. 4-95	p. 4-99
sequence.CURRVAL (IDS	Returns the current value of the specified sequence.	p. 4-96	p. 4-103
DATE function	Converts to a DATE value a nondate expression argument	p. 4-143	p. 4-144
DAY function	Returns an integer that represents the day of the month	p. 4-143	p. 4-145
DBINFO (as a set of functions)	Provides a set of functions for retrieving different types of database information. To invoke each function, specify the appropriate DBINFO option. Each option is listed below.	p. 4-119	p. 4-120
DBINFO ('coserverid' string followed by *table***.** *column* and the 'currentrow' string) (XPS)	Returns the coserver ID of the coserver where each row of a specified table is located	p. 4-119	p. 4-127
DBINFO ('coserverid' string with no other arguments) (XPS)	Returns the coserver ID of the coserver to which the user who entered the query is connected	p. 4-119	p. 4-126
DBINFO ('dbhostname' option)	Returns the hostname of the database server to which a client application is connected	p. 4-119	p. 4-123
DBINFO ('dbspace' string followed by *table.column* and the 'currentrow' string) (XPS)	Returns the name of the dbspace where each row of a specified table is located	p. 4-119	p. 4-128

(2 of 7)

Name	Purpose	Syntax	Usage
DBINFO ('dbspace' string followed by a tblspace number)	Returns the name of a dbspace corresponding to a tblspace number	p. 4-119	p. 4-121
DBINFO ('serial8' option) (IDS)	Returns the most recently inserted SERIAL8 value	p. 4-119	p. 4-126
DBINFO ('sessionid' option)	Returns the session ID of the current session	p. 4-119	p. 4-122
DBINFO ('sqlca.sqlerrd1' option)	Returns the last serial value inserted in a table	p. 4-119	p. 4-121
DBINFO ('sqlca.sqlerrd2' option)	Returns the number of rows processed by selects, inserts, deletes, updates, EXECUTE PROCEDURE statements, and EXECUTE FUNCTION statements	p. 4-119	p. 4-122
DBINFO ('version' option)	Returns the exact version of the database server to which a client application is connected	p. 4-119	p. 4-124
DBSERVERNAME function	Returns the name of the database server	p. 4-95	p. 4-98
DECODE function	Evaluates one or more expression pairs and compares the *when* expression in each pair with a specified value expression	p. 4-93	p. 4-93
Division (/) operator	Returns the quotient of two numeric expression operands	p. 4-68	p. 4-77
Double-colon (::) cast operator (IDS)	Casts a value to a specified data type	p. 4-79	p. 4-79
EXP function	Returns the exponent of a numeric expression	p. 4-129	p. 4-129
EXTEND function	Adjusts the precision of a DATETIME or DATE value	p. 4-143	p. 4-146
FILETOBLOB function (IDS)	Creates a BLOB value for data that is stored in a specified operating-system file	p. 4-134	p. 4-135
FILETOCLOB function (IDS)	Creates a CLOB value for data that is stored in a specified operating-system file	p. 4-134	p. 4-135
HEX function	Returns the hexadecimal encoding of an integer value	p. 4-130	p. 4-130
Host variable	See variable.	p. 4-68	p. 4-68

(3 of 7)

Name	**Purpose**	**Syntax**	**Usage**
IFX_ALLOW_NEWLINE function	Sets a newline mode that allows newline characters in quoted strings or disallows newline characters in quoted strings within a given session	p. 4-164	p. 4-164
IFX_REPLACE_MODULE function (IDS)	Replaces a loaded shared-object file with a new version that has a different name or location	p. 4-132	p. 4-132
INITCAP function	Converts a source expression so that every word in the source expression begins with an initial capital letter and all remaining letters in each word are lowercase	p. 4-161	p. 4-163
LENGTH function	Returns the number of bytes in a character column, not including any trailing blank spaces	p. 4-131	p. 4-131
LIST collection constructor (IDS)	Enables you to specify values for collections whose elements are ordered and can contain duplicate values	p. 4-108	p. 4-108
Literal BOOLEAN	Provides a literal representation of a BOOLEAN value	p. 4-95	p. 4-95
Literal collection (IDS)	Provides the values of elements in a collection data type	p. 4-95	p. 4-105
Literal DATETIME	Provides a DATETIME value	p. 4-95	p. 4-101
Literal INTERVAL	Provides an INTERVAL value	p. 4-95	p. 4-101
Literal number	Provides a numeric value	p. 4-95	p. 4-97
Literal opaque type (IDS)	Provides a literal representation of an opaque data type	p. 4-95	p. 4-95
Literal row (IDS)	Provides the values of elements in a ROW data type	p. 4-95	p. 4-105
LOCOPY function (IDS)	Creates a copy of a smart large object	p. 4-134	p. 4-140
LOGN function	Returns the natural log of a numeric expression	p. 4-129	p. 4-130
LOG10 function	Returns the log of a value to the base 10	p. 4-129	p. 4-129
LOTOFILE function (IDS)	Copies a smart large object to an operating-system file	p. 4-134	p. 4-138
LOWER function	Converts a source expression to lowercase characters	p. 4-161	p. 4-163
LPAD function	Returns a copy of a source string that is left-padded by a specified number of pad characters	p. 4-159	p. 4-159
MAX function	Returns the largest value in the specified set of values	p. 4-171	p. 4-180

(4 of 7)

Name	Purpose	Syntax	Usage
MDY function	Returns a DATE value with the month, day, and year	p. 4-143	p. 4-147
MIN function	Returns the lowest value in a specified set of values	p. 4-171	p. 4-180
MOD function	Returns the modulus (the integer-divisiom remainder value) from two numeric arguments.	p. 4-114	p. 4-115
MONTH function	Returns an integer that corresponds to the month portion of its DATE or DATETIME argument	p. 4-143	p. 4-145
Multiplication (*) operator	Multiplies two numeric operands and returns the product	p. 4-68	p. 4-77
MULTISET collection constructor (IDS)	Enables you to specify values for collection columns. The MULTISET constructor indicates a collection of elements with the following qualities: ■ The collection can contain duplicate values. ■ Elements have no specific order associated with them.	p. 4-108	p. 4-108
sequence.NEXTVAL (IDS)	Increments the current value of the specified sequence.	p. 4-96	p. 4-102
NULL keyword	Specifies unknown, missing, or logically undefined value.	p. 4-110	p. 4-112
NVL function	Evaluates an expression and returns the value of the expression if the value of the expression is not NULL. If the value of the expression is NULL, the NVL function returns a specified result.	p. 4-92	p. 4-92
OCTET_LENGTH function	Returns the number of bytes in a character column, including any trailing spaces	p. 4-131	p. 4-132
POW function	Raises a base value to a specified power	p. 4-114	p. 4-115
Procedure-call expression	See user-defined function.	p. 4-165	p. 4-165
Program variable	See variable.	p. 4-68	p. 4-68
Quoted string	Literal character string	p. 4-95	p. 4-97
RANGE function	Computes the range for a sample of a population	p. 4-171	p. 4-180
REPLACE function	Replaces specified characters in a source string with different characters	p. 4-158	p. 4-158
ROOT function	Returns the root value of a numeric expression	p. 4-114	p. 4-115

(5 of 7)

Name	Purpose	Syntax	Usage
ROUND function	Returns the rounded value of an expression	p. 4-114	p. 4-116
ROW constructor (IDS)	Enables you to specify values for columns that are named ROW data types	p. 4-106	p. 4-106
RPAD function	Returns a copy of a source string that is right-padded by a specified number of pad characters	p. 4-160	p. 4-160
SET collection constructor (IDS)	Enables you to specify values for collection columns. The SET constructor indicates a collection of elements with the following qualities: ■ The collection must contain unique values. ■ Elements have no specific order associated with them.	p. 4-108	p. 4-108
SIN function	Returns the sine of a radian expression	p. 4-149	p. 4-150
SITENAME function	See DBSERVERNAME function.	p. 4-95	p. 4-98
SPL routine expression	See user-defined functions.	p. 4-165	p. 4-165
SPL variable	SPL variable that stores an expression	p. 4-68	p. 4-68
SQRT function	Returns the square root of a numeric expression	p. 4-114	p. 4-117
Statement-Local-Variable expression	Specifies how you can use a defined statement-local variable (SLV) elsewhere in an SQL statement	p. 4-169	p. 4-169
STDEV function	Computes the standard deviation for a specified sample	p. 4-171	p. 4-181
SUBSTR function	Returns a subset of a source string	p. 4-156	p. 4-156
SUBSTRING function	Returns a subset of a source string	p. 4-154	p. 4-154
Substring ([*first*, *last*]) operator	Returns the *first* through *last* bytes of its string operand, for *first* and *last* positive integers specifying byte positions	p. 4-82	p. 2-614
Subtraction (-) operator	Returns the difference between two numeric expressions	p. 4-68	p. 4-77
SUM function	Returns the sum of a specified set of values	p. 4-171	p. 4-180
TAN function	Returns the tangent of a radian expression	p. 4-149	p. 4-150
TO_CHAR function	Converts a DATE or DATETIME value to a string	p. 4-143	p. 4-147
TO_DATE function	Converts a character string to a DATETIME value	p. 4-143	p. 4-148

(6 of 7)

Name	Purpose	Syntax	Usage
TODAY operator	Returns the system date	p. 4-95	p. 4-99
TRIM function	Removes leading or trailing (or both) pad characters from a string	p. 4-152	p. 4-152
TRUNC function	Returns the truncated value of a numeric expression	p. 4-114	p. 4-117
UNITS operator	Enables you to display a simple interval or increase or decrease a specific interval or datetime value	p. 4-95	p. 4-101
UPPER function	Converts a source expression to uppercase characters	p. 4-161	p. 4-162
User-defined aggregate (IDS)	An aggregate that you define (as opposed to built-in aggregates that the database server provides)	p. 4-185	p. 4-185
User-defined function	A function that you write (as opposed to the built-in functions that the database server provides)	p. 4-165	p. 4-165
USER operator	Returns the login name of the current user	p. 4-95	p. 4-97
Variable	Host or program variable that stores an expression	p. 4-68	p. 4-68
VARIANCE function	Returns the variance for a sample of values as an unbiased estimate of the variance of a population	p. 4-171	p. 4-182
WEEKDAY function	Returns an integer that represents the day of the week	p. 4-143	p. 4-145
YEAR function	Returns a four-digit integer that represents the year	p. 4-143	p. 4-145
* symbol	See multiplication operator.	p. 4-68	p. 4-77
+ symbol	See addition operator. Also, unary plus sign.	p. 4-68	p. 4-77
- symbol	See subtraction operator. Also, unary minus sign.	p. 4-68	p. 4-77
/ symbol	See division operator.	p. 4-68	p. 4-77
:: symbols	See double-colon cast operator.	p. 4-79	p. 4-79
|| symbol	See double-pipe concatenation operator.	p. 4-68	p. 4-78
[*first*, *last*] symbols	See substring operator	p. 4-68	p. 4-77

(7 of 7)

Sections that follow describe the syntax and usage of each expression that appears in the preceding table.

Arithmetic Operators

Binary arithmetic operators can combine expressions that return numbers.

Arithmetic Operation	Arithmetic Operator	Operator Function	Arithmetic Operation	Arithmetic Operator	Operator Function
Addition	+	**plus( )**	Multiplication	*	**times( )**
Subtraction	–	**minus( )**	Division	/	**divide( )**

The following examples use binary arithmetic operators:

```
quantity * total_price
price * 2
COUNT(*) + 2
```

If you combine a DATETIME value with one or more INTERVAL values, all the fields of the INTERVAL value must be present in the DATETIME value; no implicit EXTEND function is performed. In addition, you cannot use YEAR TO MONTH intervals with DAY TO SECOND intervals. For additional information about binary arithmetic operators, see the *IBM Informix Guide to SQL: Reference*.

The binary arithmetic operators have associated operator functions, as the preceding table shows. Connecting two expressions with a binary operator is equivalent to invoking the associated operator function on the expressions. For example, the following two statements both select the product of the **total_price** column and 2. In the first statement, the * operator implicitly invokes the **times( )** function.

```
SELECT (total_price * 2) FROM items
   WHERE order_num = 1001
SELECT times(total_price, 2) FROM items
   WHERE order_num = 1001
```

You cannot use arithmetic operators to combine expressions that use aggregate functions with column expressions.

The database server provides the operator functions associated with the relational operators for all built-in data types. You can define new versions of these operator functions to handle your own user-defined data types.

For more information, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

The database server also supports the following unary arithmetic operators.

Sign of Number	Unary Arithmetic Operator	Operator Function
Positive	+	positive( )
Negative	–	negate( )

The unary arithmetic operators have the associated operator functions that the preceding table shows. You can define new versions of these functions to handle your own user-defined data types. For more information on this topic, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

If any value that participates in an arithmetic expression is NULL, the value of the entire expression is NULL, as the following example shows:

```
SELECT order_num, ship_charge/ship_weight FROM orders
   WHERE order_num = 1023
```

If either **ship_charge** or **ship_weight** is NULL, the value returned for the expression **ship_charge**/**ship_weight** is also NULL. If the expression **ship_charge**/**ship_weight** is used in a condition, its truth value is unknown.

Concatenation Operator

You can use the concatenation operator (||) to concatenate two expressions. These examples show some possible concatenated-expression combinations.

- The first example concatenates the **zipcode** column to the first three letters of the **lname** column.
- The second example concatenates the suffix **.dbg** to the contents of a host variable called **file_variable**.
- The third example concatenates the value that the **TODAY** operator returns to the string `Date`.

```
lname[1,3] || zipcode

:file_variable || '.dbg'

'Date:' || TODAY
```

E/C

You cannot use the concatenation operator in an embedded-language-only statement. The ESQL/C-only statements appear in the following list:

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
CLOSE
CREATE FUNCTION FROM
CREATE PROCEDURE FROM
CREATE ROUTINE FROM
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DECLARE
DESCRIBE
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
OPEN
PREPARE
PUT
SET AUTOFREE
SET CONNECTION
SET DESCRIPTOR
WHENEVER

You can use the concatenation operator in the SELECT, INSERT, or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement in the DECLARE statement.

You can use the concatenation operator in the SQL statement or statements in the PREPARE statement. ♦

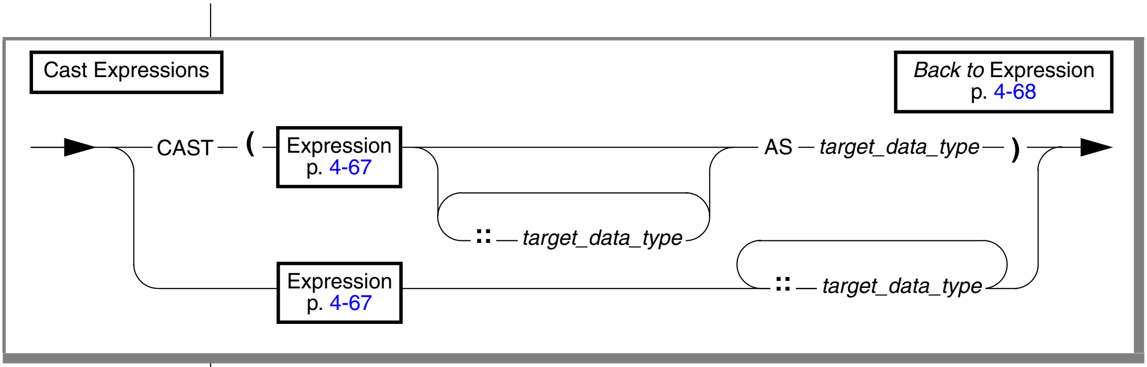
The concatenation operator (||) has an associated operator function called **concat( )**. You can define a **concat( )** function to handle your own string-based user-defined data types. For more information, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

IDS

Cast Expressions

You can use the CAST AS keywords or the double-colon cast operator (**::**) to cast an expression to another data type. Both the operator and the keywords invoke a cast from the data type of the expression to the target data type.

To invoke an explicit cast, you can use either the cast operator or the CAST AS keywords. If you use the cast operator or CAST AS keywords, but no explicit or implicit cast was defined to perform the conversion between two data types, the statement returns an error.



Element	Purpose	Restrictions	Syntax
target_data_type	Data type that results after the cast is applied	See "Rules for the Target Data Type" on page 4-80.	Data type, p. 4-49

Rules for the Target Data Type

The following rules restrict the *target data type* in cast expressions:

- The target data type must be either a built-in type, a user-defined type, or a named ROW type in the database.
- The target data type cannot be an unnamed ROW or a collection type.
- The target data type can be a BLOB data type under the following conditions:
  - The source expression (the expression to be cast to another data type) is a BYTE data type.
  - The source expression is a user-defined type and the user has defined a cast from the user-defined type to the BLOB type.
- The target data type can be a CLOB type under these conditions:
  - The source expression is a TEXT data type.
  - The source expression is a user-defined type and the user has defined a cast from the user-defined type to the CLOB type.
- You cannot cast a BLOB data type to a BYTE data type.
- You cannot cast a CLOB data type to a TEXT data type.
- An explicit or implicit cast must exist that can convert the data type of the source expression to the target data type.

Examples of Cast Expressions

The following examples show two different ways to convert the sum of *x* and *y* to a user-defined data type, **user_type**. The two methods produce identical results. Both require the existence of an explicit or implicit cast from the type returned by *x* + *y* to the user-defined type.

```
CAST ((x + y) AS user_type)
(x + y)::user_type
```

The following examples show two different ways of finding the integer equivalent of the expression **expr**. Both require the existence of an implicit or explicit cast from the data type of **expr** to the INTEGER data type.

```
CAST (expr AS INTEGER)
expr::INTEGER
```

In the following example, the user casts a BYTE column to the BLOB type and copies the BLOB data to an operating-system file:

```
SELECT LOTOFILE(mybytecol::blob, 'fname', 'client')
   FROM mytab
   WHERE pkey = 12345
```

In the following example, the user casts a TEXT column to a CLOB value and then updates a CLOB column in the same table to have the CLOB value derived from the TEXT column:

```
UPDATE newtab SET myclobcol = mytextcol::clob
```

The Keyword NULL in Cast Expressions

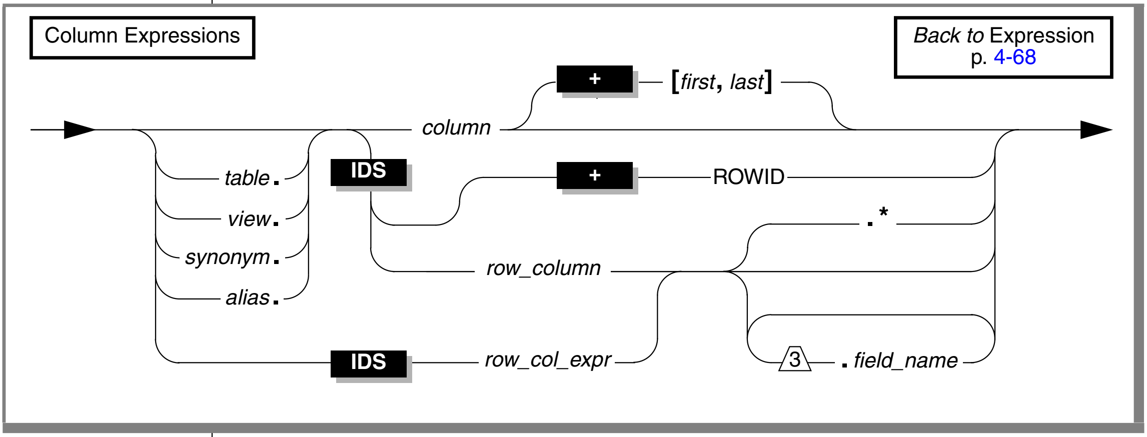
Cast expressions can appear in the SELECT list, including expressions of the form NULL**::***datatype*, where *datatype* is any data type known to the database.

```
SELECT newtable.col0, null::int FROM newtable;
```

The keyword NULL has a global scope of reference within expressions.
In SQL, the keyword NULL is the only syntactic mechanism for accessing a NULL value. Any attempt to redefine or restrict the global scope of the keyword NULL (for example, declaring an SPL variable called **null**) disables any cast expression that involves a NULL value. Make sure that the keyword NULL receives its global scope in all expression contexts.

Column Expressions

A *column expression* specifies a data value in a column in the database, or a substring of the value, or (for Dynamic Server only) a field within a ROW-type column. This is the syntax for column expressions.



Element	Purpose	Restrictions	Syntax
alias	Temporary alternative name for a table or view, declared in the FROM clause of a query	Restrictions depend on the clause of the SELECT statement in which *alias* occurs.	Identifier, p. 4-189
column	Name of a column	Restrictions depend on the SQL statement where *column* occurs.	Identifier, p. 4-189
field_name	Name of a ROW field in the ROW column or ROW-column expression	Must be a member of the ROW that *row-column name* or *row_col_expr* or *field name* (for nested rows) specifies.	Identifier, p. 4-189
first,last	Integers indicating positions of first and last characters within *column*	The *column* must be of type CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, or TEXT, and $0 < \mathit{first} \leq \mathit{last}$	Literal Number, p. 4-216
row_col_expr	Expression that returns ROW-type values	Must return a ROW data type.	Expression, p. 4-67
row_column	Name of a ROW-type column	Must be a named ROW type or an unnamed ROW type.	Identifier, p. 4-189
synonym, table, view	Table, view, or synonym (for the table or view) that contains *column*	Synonym and the table or view to which it points must exist.	Database Object Name, p. 4-46

The following examples show column expressions:

```
company

items.price

cat_advert [1,15]
```

You must qualify the *column* name with a *table* name or *alias* whenever it is necessary to distinguish between columns that have the same name but are in different tables. The SELECT statements that the following example shows use **customer_num** from the **customer** and **orders** tables. The first example precedes the column names with table names. The second example precedes the column names with table aliases.

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num

SELECT * FROM customer c, orders o
   WHERE c.customer_num = o.customer_num
```

Using Dot Notation

Dot notation (sometimes called the *membership operator*) allows you to qualify an SQL identifier with another SQL identifier of which it is a component. You separate the identifiers with the period (**.**) symbol. For example, you can qualify a column name with any of the following SQL identifiers:

- Table name: *table_name***.***column_name*
- View name: *view_name***.***column_name*
- Synonym name: *syn_name***.***column_name*

These forms of dot notation are called *column projections*.

You can also use dot notation to directly access the fields of a named or unnamed ROW column, as in the following example:

```
row-column name.field name
```

This use of dot notation is called a *field projection*. For example, suppose you have a column called **rect** with the following definition:

```
CREATE TABLE rectangles
(
   area float,
   rect ROW(x int, y int, length float, width float)
)
```

The following SELECT statement uses dot notation to access field **length** of the **rect** column:

```
SELECT rect.length FROM rectangles
   WHERE area = 64
```

Selecting All Fields of a Column with Asterisk Notation

If you want to select all fields of a column that has a row type, you can specify the column name without dot notation. For example, you can select all fields of the **rect** column as follows:

```
SELECT rect FROM rectangles
   WHERE area = 64
```

You can also use asterisk (*) notation to project all the fields of a column that has a ROW data type. For example, if you want to use asterisk notation to select all fields of the **rect** column, you can enter the following statement:

```
SELECT rect.* FROM rectangles
   WHERE area = 64
```

Asterisk notation is easier than specifying each field of the **rect** column individually:

```
SELECT rect.x, rect.y, rect.length,
   rect.width
   FROM rectangles
   WHERE area = 64
```

Asterisk notation is valid only in the select list of a SELECT statement. It can specify all fields of a ROW-type column or the data that a ROW-column expression returns.

Asterisk notation is not necessary with ROW-type columns because you can specify the column name alone to project all of its fields. Asterisk notation is quite helpful, however, with ROW-type expressions such as subqueries and user-defined functions that return ROW-type values. For more information, see "Using Dot Notation with Row-Type Expressions" on page 4-86.

You can use asterisk notation with columns and expressions of ROW data types in the select list of a SELECT statement only. You cannot use asterisk notation with columns and expressions of ROW type in any other clause of a SELECT statement.

Selecting Nested Fields

When the ROW type that defines a column itself contains other ROW types, the column contains nested fields. Use dot notation to access these nested fields within a column.

For example, assume that the **address** column of the **employee** table contains the fields: **street**, **city**, **state**, and **zip**. In addition, the **zip** field contains the nested fields: **z_code** and **z_suffix**. A query on the **zip** field returns values for the **z_code** and **z_suffix** fields. You can specify, however, that a query returns only specific nested fields. The following example shows how to use dot notation to construct a SELECT statement that returns rows for the **z_code** field of the **address** column only:

```
SELECT address.zip.z_code
   FROM employee
```

Rules of Precedence

The database server uses the following precedence rules to interpret dot notation:

1. schema *name_a* . table *name_b* . column *name_c* . field *name_d*
2. table *name_a* . column *name_b* . field *name_c* . field *name_d*
3. column *name_a* . field *name_b* . field *name_c* . field *name_d*

When the meaning of a particular identifier is ambiguous, the database server uses precedence rules to determine which database object the identifier specifies. Consider the following two tables:

```
CREATE TABLE b (c ROW(d INTEGER, e CHAR(2));
CREATE TABLE c (d INTEGER);
```

In the following SELECT statement, the expression `c.d` references column **d** of table **c** (rather than field **d** of column **c** in table **b**) because a table identifier has a higher precedence than a column identifier:

```
SELECT *
FROM b,c
WHERE c.d = 10
```

For more information about precedence rules and how to use dot notation with row columns, see the *IBM Informix Guide to SQL: Tutorial*.

Using Dot Notation with Row-Type Expressions

Besides specifying a column of a ROW data type, you can also use dot notation with any expression that evaluates to a ROW type. In an INSERT statement, for example, you can use dot notation in a subquery that returns a single row of values. Assume that you created a ROW type named **row_t**:

```
CREATE ROW TYPE row_t (part_id INT, amt INT)
```

Also assume that you created a typed table named **tab1** that is based on the **row_t** ROW type:

```
CREATE TABLE tab1 OF TYPE row_t
```

Assume also that you inserted the following values into table **tab1**:

```
INSERT INTO tab1 VALUES (ROW(1,7));
INSERT INTO tab1 VALUES (ROW(2,10));
```

Finally, assume that you created another table named **tab2**:

```
CREATE TABLE tab2 (colx INT)
```

Now you can use dot notation to insert the value from only the **part_id** column of table **tab1** into the **tab2** table:

```
INSERT INTO tab2
   VALUES ((SELECT t FROM tab1 t
      WHERE part_id = 1).part_id)
```

The asterisk form of dot notation is not necessary when you want to select all fields of a row-type column because you can specify the column name alone to select all of its fields. The asterisk form of dot notation can be quite helpful, however, when you use a subquery, as in the preceding example, or when you call a user-defined function to return row-type values.

Suppose that a user-defined function named **new_row** returns ROW-type values, and you want to call this function to insert the ROW-type values into a table. Asterisk notation makes it easy to specify that all the ROW-type values that the **new_row** function returns are to be inserted into the table:

```
INSERT INTO mytab2 SELECT new_row (mycol).* FROM mytab1
```

References to the fields of a ROW-type column or a ROW-type expression are not allowed in fragment expressions. A fragment expression is an expression that defines a table fragment or an index fragment in statements like CREATE TABLE, CREATE INDEX, and ALTER FRAGMENT.

Using Subscripts on Character Columns

You can use subscripts on CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, and TEXT columns. The subscripts indicate the starting and ending character positions in the expression. Together the column subscripts define a *column substring* as the portion of the column that is contained in the expression.

For example, if a value in the **lname** column of the **customer** table is `Greenburg`, the following expression evaluates to `burg`:

```
lname[6,9]
```

A conditional expression can include a column expression that uses the substring operator ([*first*, *last*]), as in the following example:

```
SELECT lname FROM customer WHERE phone[5,7] = '356'
```

Here the quotes are required, to prevent the database server from applying a numeric filter to the digits in the criterion value.

GLS

For information on the GLS aspects of column subscripts and substrings, see the *IBM Informix GLS User's Guide*. ♦

IDS

Using Rowids

In Dynamic Server, you can use the **rowid** column that is associated with a table row as a property of the row. The **rowid** column is essentially a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The **rowid** column is unique for each row, but it is not necessarily sequential. It is recommended, however, that you use primary keys as an access method rather than exploiting the **rowid** column.

The following examples use the ROWID keyword in a SELECT statement:

```
SELECT *, ROWID FROM customer

SELECT fname, ROWID FROM customer ORDER BY ROWID

SELECT HEX(rowid) FROM customer WHERE customer_num = 106
```

The last example shows how to get the page number (the first six digits after 0x) and the slot number (the last two digits) of the location of your row.

You cannot use the ROWID keyword in the select list of the Projection clause of a query that contains an aggregate function.

IDS

Using Smart Large Objects

The SELECT, UPDATE, and INSERT statements do not manipulate the values of smart large objects directly. Instead, they use a *handle value*, which is a type of pointer, to access the BLOB or CLOB value, as follows:

- The SELECT statement returns a handle value to the BLOB or CLOB value that the select list specifies. SELECT does not return the actual data for the BLOB or CLOB column that the select list specifies. Instead, it returns a handle value to the column data.
- The INSERT and UPDATE statements do not send the actual data for the BLOB or CLOB column to the database server. Instead, they accept a handle value to this data as the value to be inserted or updated.

To access the data of a smart-large-object column, you must use one of the following application programming interfaces (APIs):

- From within an IBM Informix ESQL/C program, use the ESQL/C library functions that access smart large objects. For more information, see the *IBM Informix ESQL/C Programmer's Manual*.
- From within a C program such as a DataBlade module, use the Client and Server API. For more information, see your *DataBlade Developer's Kit User's Guide*.

You cannot use the name of a smart-large-object column in expressions that involve arithmetic operators. For example, operations such as addition or subtraction on the smart-large-object handle value have no meaning.

When you select a smart-large-object column, you can assign the handle value to any number of columns: all columns with the same handle value share the CLOB or BLOB value. This storage arrangement reduces the amount of disk space that the CLOB or BLOB value, but when several columns share the same smart-large-object value, the following conditions result:

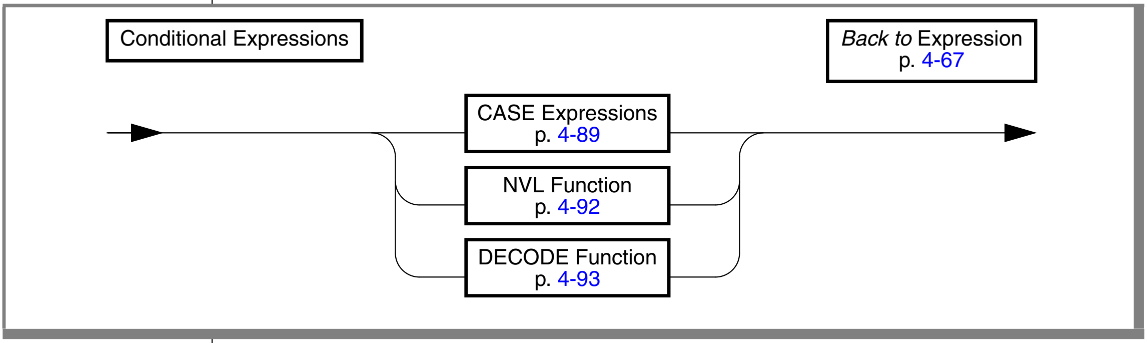
- The chance of lock contention on a CLOB or BLOB column increases. If two columns share the same smart-large-object value, the data might be locked by either column that needs to access it.
- The CLOB or BLOB value can be updated from a number of points.

To remove these constraints, you can create separate copies of the BLOB or CLOB data for each column that needs to access it. You can use the **LOCOPY** function to create a copy of an existing smart large object.

You can also use the built-in functions **LOTOFILE**, **FILETOCLOB**, and **FILETOBLOB** to access smart-large-object values, as described in "Smart-Large-Object Functions" on page 4-134. For more information on the BLOB and CLOB data types, see the *IBM Informix Guide to SQL: Reference*.

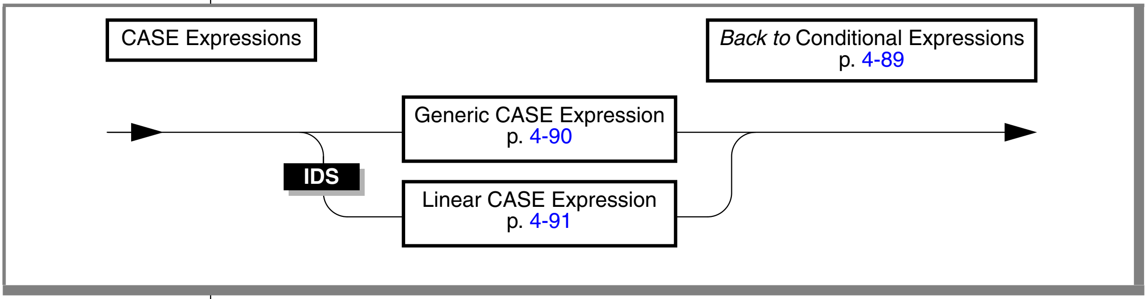
Conditional Expressions

Conditional expressions return values that depend on the outcome of conditional tests. This diagram shows the syntax for Conditional Expressions.



CASE Expressions

The CASE expression allows an SQL statement such as the SELECT statement to return one of several possible results, depending on which of several condition evaluates to true. The CASE expression has two forms: generic CASE expressions and linear CASE expressions.



You must include at least one WHEN clause in the CASE expression. Subsequent WHEN clauses and the ELSE clause are optional.

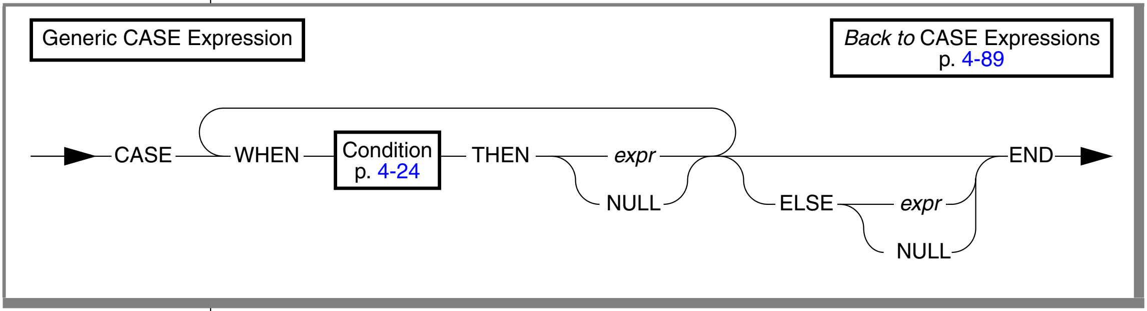
You can use a generic or linear CASE expression wherever you can use a column expression in an SQL statement (for example, in the select list of the Projection clause a SELECT statement).

Expressions in the search condition or the result value expression can contain subqueries, and you can nest a CASE expression in another CASE expression.

When a CASE expression appears in an aggregate expression, you cannot use aggregate functions in the CASE expression.

Generic CASE Expressions

A generic CASE expression tests for a TRUE condition in a WHEN clause. If it finds a TRUE condition, it returns the result specified in the THEN clause.



Element	Purpose	Restrictions	Syntax
expr	Expression that returns some data type	Data type of *expr* in a THEN clause must be compatible with data types of expressions in other THEN clauses.	Expression, p. 4-67

The database server processes the WHEN clauses in the order that they appear in the statement. If the search condition of a WHEN clause evaluates to `TRUE` , the database server uses the value of the corresponding THEN expression as the result, and stops processing the CASE expression.

If no WHEN condition evaluates to `TRUE`, the database server uses the ELSE expression as the overall result. If no WHEN condition evaluates to `TRUE`, and no ELSE clause was specified, the returned CASE expression value is NULL. You can use the IS NULL condition to handle NULL results. For information on how to handle NULL values, see "IS NULL Condition" on page 4-32.

The next example shows a generic CASE expression in the Projection clause.

In this example the user retrieves the name and address of each customer as well as a calculated number that is based on the number of problems that exist for that customer.

```
SELECT cust_name,
   CASE
   WHEN number_of_problems = 0
      THEN 100
   WHEN number_of_problems > 0 AND number_of_problems < 4
      THEN number_of_problems * 500
   WHEN number_of_problems >= 4 and number_of_problems <= 9
      THEN number_of_problems * 400
   ELSE
      (number_of_problems * 300) + 250
   END,
   cust_address
FROM custtab
```

In a generic CASE expression, all the results should be of the same type, or they should evaluate to a common compatible type. If the results in all the WHEN clauses are not of the same type, or if they do not evaluate to values of mutually compatible types, an error occurs.

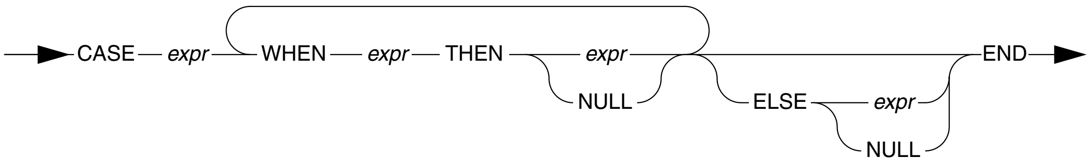
IDS

Linear CASE Expressions

A linear CASE expression compares the value of the expression that follows the CASE keyword with an expression in a WHEN clause.

Linear CASE Expression

Back to CASE Expressions p. 4-89



Element	Purpose	Restrictions	Syntax
expr	Expression that returns value of some data type	Data type of *expr* that follows the WHEN keyword must be compatible with data type of the expression that follows the CASE keyword. Data type of *expr* in the THEN clause must be compatible with data types of expressions in other THEN clauses.	Expression, p. 4-67

The database server evaluates the expression that follows the CASE keyword, and then processes the WHEN clauses sequentially. If an expression after the WHEN keyword returns the same value as the expression that follows the CASE keyword, the database server uses the value of the expression that follows the THEN keyword as the overall result of the CASE expression. Then the database server stops processing the CASE expression.

If none of the WHEN expressions return the same value as the expression that follows the CASE keyword, the database server uses the expression of the ELSE clause as the overall result of the CASE expression (or, if no ELSE clause was specified, the returned value of the CASE expression is NULL).

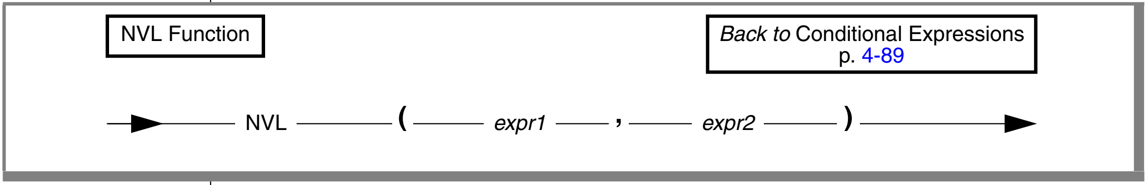
The next example shows a linear CASE expression in the select list of the Projection clause of a SELECT statement. For each movie in a table of movie titles, the query returns the title, the cost, and the type of the movie. The statement uses a CASE expression to derive the type of each movie.

```
SELECT title, CASE movie_type
      WHEN 1 THEN 'HORROR'
      WHEN 2 THEN 'COMEDY'
      WHEN 3 THEN 'ROMANCE'
      WHEN 4 THEN 'WESTERN'
      ELSE 'UNCLASSIFIED'
   END,
   our_cost FROM movie_titles
```

In linear CASE expressions, the data types of WHEN clause expressions must be compatible with that of the expression that follows the CASE keyword.

NVL Function

The **NVL** expression returns different results, depending on whether its first argument evaluates to NULL.



Element	Purpose	Restrictions	Syntax
expr1 *expr2*	Expressions that return values of a compatible data type	Cannot be a host variable or a BYTE or TEXT data type.	Expression, p. 4-67

NVL evaluates *expression1*. If *expression1* is not NULL, then NVL returns the value of *expression1*. If *expression1* is NULL, NVL returns the value of *expression2*. The expressions *expression1* and *expression2* can be of any data type, as long as they can be cast to a common compatible data type.

Suppose that the **addr** column of the **employees** table has NULL values in some rows, and the user wants to be able to print the label `Address unknown` for these rows. The user enters the following SELECT statement to display the label `Address unknown` when the **addr** column has a NULL value.

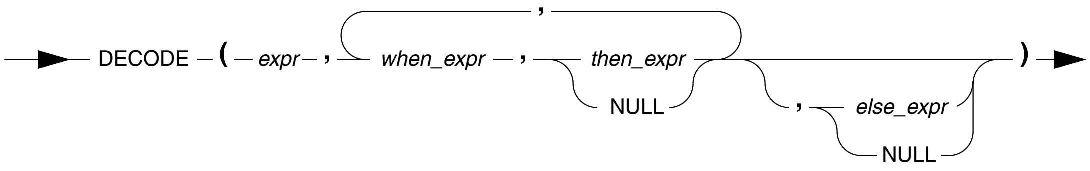
```
SELECT fname, NVL (addr, 'Address unknown') AS address
   FROM employees
```

DECODE Function

The **DECODE** expression is similar to the CASE expression in that it can print different results depending on the values found in a specified column.

DECODE Function

Back to Conditional Expressions p. 4-89



Element	Purpose	Restrictions	Syntax
expr, else_expr, then_expr, when_expr	Expressions whose values and data types can be evaluated	Data types of *when_expr* and *expr* must be compatible, as must *then_expr* and *else_expr*. Value of *when_expr* cannot be a NULL.	Expression, p. 4-67

The expressions *expr*, *when_expr*, and *then_expr* are required. **DECODE** evaluates *expr* and compares it to *when_expr*. If the value of *when_expr* matches the value of *expr*, then **DECODE** returns *then_expr*.

The expressions *when_expr* and *then_expr* are an expression pair, and you can specify any number of expression pairs in the **DECODE** function. In all cases, **DECODE** compares the first member of the pair against *expr* and returns the second member of the pair if the first member matches *expr*.

If no expression matches *expr*, **DECODE** returns *else_expr*. If no expression matches *expr* and you specified no *else_expr*, then **DECODE** returns NULL.

You can specify any data type for the arguments, but two restrictions exist:

- All instances of *when_expr* must have the same data type, or a common compatible type must exist. All instances of *when_expr* must also have the same (or a compatible) data type as *expr*.
- All instances of *then_expr* must have the same data type, or a common compatible type must exist. All instances of *then_expr* must also have the same (or a compatible) data type as *else_expr*.

Suppose that a user wants to convert descriptive values in the **evaluation** column of the **students** table to numeric values in the output. The following table shows the contents of the **students** table.

firstname	evaluation	firstname	evaluation
Edward	Great	Mary	Good
Joe	Not done	Jim	Poor

The user now enters a query with the **DECODE** function to convert the descriptive values in the **evaluation** column to numeric equivalents.

```
SELECT firstname, DECODE(evaluation,
   'Poor', 0,
   'Fair', 25,
   'Good', 50,
   'Very Good', 75,
   'Great', 100,
   -1) as grade
FROM students
```

The following table shows the output of this SELECT statement.

firstname	evaluation	firstname	evaluation
Edward	100	Mary	50
Joe	-1	Jim	0

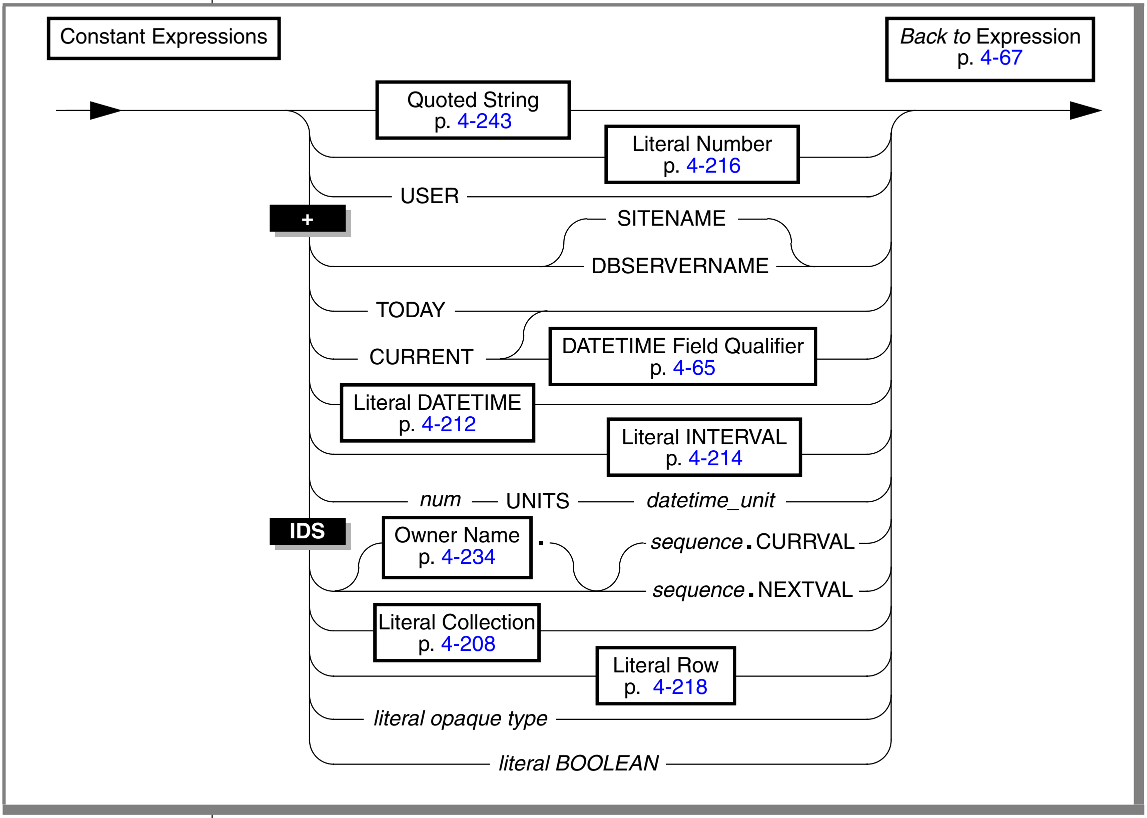
Constant Expressions

Certain expressions that return a fixed value are called *constant expressions*. Among these are four operators (or *system constants*) whose returned values are determined at runtime:

- **DBSERVERNAME** returns the name of the current database server.
- **SITENAME** is a synonym for **DBSERVERNAME.**
- **TODAY** returns the current calendar date, from the system clock.
- **USER** returns the login name (also called the *authorization identifier*) of the current user.

Besides these system constants, the term *constant expression* can also refer to a quoted string, a literal value, or to expressions that include the **CURRENT** or **UNITS** operator with its operands.

The Constant Expression segment has the following syntax.



Element	Purpose	Restrictions	Syntax
literal Boolean	Literal representation of a BOOLEAN value	Must be either *t* (TRUE) or *f* (FALSE).	Quoted string, p. 4-243
literal opaque type	Literal representation of value of an opaque data type	Must be recognized by input support function of opaque type.	Defined by UDT developer.
num	How many of the specified time units. See "UNITS Operator" on page 4-101.	If *num* is not an integer, the fractional part is truncated.	Literal Number, p. 4-216
sequence	Name or synonym of a sequence object	Must exist in current database	Identifier, p. 4-189
time_unit	Keyword to specify time unit: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, *or* FRACTION	Must be one of the keywords at left. Case insensitive but cannot be enclosed within quotes.	See the **Restrictions** column.

Quoted String

The following examples show quoted strings as expressions:

```
SELECT 'The first name is ', fname FROM customer

INSERT INTO manufact VALUES ('SPS', 'SuperSport')

UPDATE cust_calls SET res_dtime = '1997-1-1 10:45'
   WHERE customer_num = 120 AND call_code = 'B'
```

For more information, see "Quoted String" on page 4-243.

Literal Number

The following examples show literal numbers as expressions:

```
INSERT INTO items VALUES (4, 35, 52, 'HRO', 12, 4.00)

INSERT INTO acreage VALUES (4, 5.2e4)

SELECT unit_price + 5 FROM stock

SELECT -1 * balance FROM accounts
```

For more information, see "Literal Number" on page 4-216.

USER Operator

The **USER** operator returns a string that contains the login name (or authorization identifier) of the current user who is running the process.

The following statements show how you might use the **USER** operator:

```
INSERT INTO cust_calls VALUES
   (221,CURRENT,USER,'B','Decimal point off', NULL, NULL)

SELECT * FROM cust_calls WHERE user_id = USER

UPDATE cust_calls SET user_id = USER WHERE customer_num = 220
```

The **USER** operator does not change the lettercase of a user ID. If you use **USER** in an expression and the current user is **Robertm**, the **USER** operator returns **Robertm**, not **robertm**.

If you specify **USER** as the default value for a column, the column must have a CHAR, VARCHAR, NCHAR, or NVARCHAR data type.

IDS

If you specify **USER** as the default value for a column, the size of the column should be at least 32 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the column is too small to store the default value. ♦

XPS

If you specify **USER** as the default value for a column, the size of the column should be at least 8 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the column is too small to store the default value. ♦

ANSI

In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored as uppercase letters. If you use the USER keyword as part of a condition, you must be sure that the way the user name is stored agrees with the values that the **USER** operator returns, with respect to lettercase. ♦

DBSERVERNAME and SITENAME Operators

The **DBSERVERNAME** operator returns the database server name, as defined in the ONCONFIG file for the installation where the current database resides or as specified in the **INFORMIXSERVER** environment variable. The two operators, **DBSERVERNAME** and **SITENAME** are synonymous. You can use the **DBSERVERNAME** operator to determine the location of a table, to put information into a table, or to extract information from a table. You can insert **DBSERVERNAME** into a simple character field or use it as a default value for a column.

If you specify **DBSERVERNAME** as a default value for a column, the column must have a CHAR, VARCHAR, NCHAR, or NVARCHAR data type.

IDS

If you specify **DBSERVERNAME** as the default value for a column, the size of the column should be at least 128 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the column is too small to store the default value. ♦

XPS

If you specify **DBSERVERNAME** as the default value for a column, the size of the column should be at least 18 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the column is too small to store the default value. ♦

In the following example, the first statement returns the name of the database server where the **customer** table resides. Because the query is not restricted with a WHERE clause, it returns **DBSERVERNAME** for every row in the table. If you add the DISTINCT keyword to the SELECT clause, the query returns **DBSERVERNAME** once. The second statement adds a row that contains the current site name to a table. The third statement returns all the rows that have the site name of the current system in **site_col**. The last statement changes the company name in the **customer** table to the current system name.

```
SELECT DBSERVERNAME FROM customer

INSERT INTO host_tab VALUES ('1', DBSERVERNAME)

SELECT * FROM host_tab WHERE site_col = DBSERVERNAME

UPDATE customer SET company = DBSERVERNAME
    WHERE customer_num = 120
```

TODAY Operator

Use the **TODAY** operator to return the system date as a DATE data type. If you specify **TODAY** as a default value for a column, it must be a DATE column.

The following examples show how you might use the **TODAY** operator in an INSERT, UPDATE, or SELECT statement:

```
UPDATE orders (order_date) SET order_date = TODAY
    WHERE order_num = 1005

INSERT INTO orders VALUES
   (0, TODAY, 120, NULL, N, '1AUE217', NULL, NULL, NULL, NULL)

SELECT * FROM orders WHERE ship_date = TODAY
```

CURRENT Operator

The **CURRENT** operator returns a DATETIME value with the date and time of day, showing the current instant.

If you do not specify a DATETIME qualifier, the default qualifier is YEAR TO FRACTION(3). The USEOSTIME configuration parameter specifies whether or not the database server uses subsecond precision when it obtains the current time from the operating system. For more information on the USEOSTIME configuration parameter, see your *Administrator's Reference*.

You can use **CURRENT** in any context where a literal DATETIME is valid. See "Literal DATETIME" on page 4-212). If you specify **CURRENT** as the default value for a column, it must be a DATETIME column and the qualifier of **CURRENT** must match the column qualifier, as the following example shows:

```
CREATE TABLE new_acct (col1 int, col2 DATETIME YEAR TO DAY
   DEFAULT CURRENT YEAR TO DAY)
```

If you use the **CURRENT** keyword in more than one place in a single statement, identical values can be returned at each point of the call. You cannot rely on **CURRENT** to provide distinct values each time it executes.

The returned value comes from the system clock and is fixed when any SQL statement starts. For example, any call to **CURRENT** from inside the SPL function that an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement invokes returns the value of the system clock when the SPL function starts.

CURRENT is always evaluated in the database server where the current database is located. If the current database is in a remote database server, the returned value is from the remote host.

CURRENT might not execute in the physical order in which it appears in a statement. You should not use **CURRENT** to mark the start, end, or a specific point in the execution of a statement.

If your platform does not provide a system call that returns the current time with subsecond precision, **CURRENT** returns a zero for the FRACTION field.

In the following example, the first statement uses **CURRENT** in a WHERE condition. The second statement uses **CURRENT** as an argument to the **DAY** function. The last query selects rows whose **call_dtime** value is within a range from the beginning of 1997 to the current instant.

```
DELETE FROM cust_calls WHERE
   res_dtime < CURRENT YEAR TO MINUTE

SELECT * FROM orders WHERE DAY(ord_date) < DAY(CURRENT)

SELECT * FROM cust_calls WHERE call_dtime
   BETWEEN '1997-1-1 00:00:00' AND CURRENT
```

For more information, see "DATETIME Field Qualifier" on page 4-65.

Literal DATETIME

The following examples show literal DATETIME as an expression:

```
SELECT DATETIME (1997-12-6) YEAR TO DAY FROM customer

UPDATE cust_calls SET res_dtime = DATETIME (1998-07-07 10:40)
         YEAR TO MINUTE
   WHERE customer_num = 110
   AND call_dtime = DATETIME (1998-07-07 10:24) YEAR TO MINUTE

SELECT * FROM cust_calls
   WHERE call_dtime
   = DATETIME (1998-12-25 00:00:00) YEAR TO SECOND
```

For more information, see "Literal DATETIME" on page 4-212.

Literal INTERVAL

The following examples show literal INTERVAL as an expression:

```
INSERT INTO manufact VALUES ('CAT', 'Catwalk Sports',
   INTERVAL (16) DAY TO DAY)

SELECT lead_time + INTERVAL (5) DAY TO DAY FROM manufact
```

The second statement in the preceding example adds five days to each value of **lead_time** selected from the **manufact** table.

For more information, see "Literal INTERVAL" on page 4-214.

UNITS Operator

The UNITS operator specifies an interval whose precision has only one time unit. You can use UNITS in arithmetic expressions that increase or decrease one of the time units in an INTERVAL or DATETIME value.

If the *n* operand is not an integer, it is rounded down to the nearest whole number when the database server evaluates the expression.

In the following example, the first SELECT statement uses the UNITS keyword to select all the manufacturer lead times, increased by five days. The second SELECT statement finds all the calls that were placed more than 30 days ago.

If the expression in the WHERE clause returns a value greater than 99 (maximum number of days), the query fails. The last statement increases the lead time for the ANZA manufacturer by two days.

```
SELECT lead_time + 5 UNITS DAY FROM manufact

SELECT * FROM cust_calls WHERE (TODAY - call_dtime) > 30 UNITS DAY

UPDATE manufact SET lead_time = 2 UNITS DAY + lead_time
   WHERE manu_code = 'ANZ'
```

IDS

NEXTVAL and CURRVAL Operators

You can access the value of a sequence using the NEXTVAL or CURRVAL operators in SQL or SPL statements. You must qualify NEXTVAL or CURRVAL with the name (or synonym) of a sequence that resides on the same database (for example, *sequence***.**NEXTVAL or *sequence***.**CURRVAL). An expression can also qualify *sequence* by the *owner* name, as in **zelaine.myseq.**CURRVAL. You can specify the identifier of *sequence* or a valid synonym, if one exists.

ANSI

In an ANSI-compliant database, you must qualify the name of the *sequence* with the name of its owner (*owner.sequence*) if you are not the owner. ♦

To use NEXTVAL or CURRVAL with a sequence, you must have the Select privilege on the sequence or have the DBA privilege on the database. For information about sequence-level privileges, see the GRANT statement.

Using NEXTVAL

To access a sequence for the first time, you must refer to *sequence*.NEXTVAL before you can refer to *sequence*.CURRVAL. The first reference to NEXTVAL returns the initial value of the sequence. Each subsequent reference to NEXTVAL increments the value of the sequence by the defined *step* and returns a new incremented value of the sequence.

You can increment a given sequence only once within a single SQL statement. Even if you specify *sequence***.**NEXTVAL more than once within a single statement, the sequence is incremented only once, so that every occurrence of *sequence***.**NEXTVAL in the same SQL statement returns the same value.

Except for the case of multiple occurrences within the same statement, every *sequence***.**NEXTVAL expression increments the *sequence*, regardless of whether you subsequently commit or roll back the current transaction.

If you specify *sequence*.NEXTVAL in a transaction that is ultimately rolled back, some sequence numbers might be skipped.

Using CURRVAL

Any reference to CURRVAL returns the current value of the sequence, which is the value that your last reference to NEXTVAL returns. After you generate a new value with NEXTVAL, you can continue to access that value using CURRVAL, regardless of whether another user increments the sequence.

If both *sequence*.CURRVAL and *sequence*.NEXTVAL occur in an SQL statement, the sequence is incremented once. In this case, each *sequence*.CURRVAL and *sequence*.NEXTVAL expression returns the same value, regardless of the order of *sequence*.CURRVAL and *sequence*.NEXTVAL within the statement.

Concurrent Access of a Sequence

A sequence always generates unique values within a database without perceptible waiting or locking, even when multiple users refer to the same sequence concurrently. When multiple users use NEXTVAL to increment the sequence, each user generates a unique value that other users cannot see.

When multiple users concurrently increment the same sequence, the values that each user sees will have gaps. For example, one user might generate a series of values, such as `1`, `4`, `6`, and `8`, from a sequence, while another user concurrently generates the values `2`, `3`, `5`, and `7` from the same sequence.

Restrictions

Several restrictions apply to NEXTVAL and CURRVAL in SQL statements:

- You must have Select privilege on the *sequence*.
- In a CREATE TABLE or ALTER TABLE statement, you cannot specify NEXTVAL or CURRVAL in the following contexts:
  - In the DEFAULT clause
  - In a check constraint

- In a SELECT statement, you cannot specify NEXTVAL or CURRVAL in the following contexts:
  - In the Select List when the DISTINCT keyword is used
  - In the WHERE, GROUP BY, or ORDER BY clauses
  - In a subquery
  - When the UNION operator combines SELECT statements
- You also cannot specify NEXTVAL or CURRVAL in these contexts:
  - In fragmentation expressions
  - In reference to a remote sequence

Examples

In the following examples, it is assumed that no other user is concurrently accessing the sequence and that the user executes the statements consecutively.

The examples are based on the following sequence and table:

```
CREATE SEQUENCE seq_2
   INCREMENT BY 1 START WITH 1
   MAXVALUE 30 MINVALUE 0
   NOCYCLE CACHE 10 ORDER;

CREATE TABLE tab1 (col1 int, col2 int);
INSERT INTO tab1 VALUES (0, 0);
```

You can use NEXTVAL (or CURRVAL) in the VALUES clause of an INSERT statement, as the following example shows:

```
INSERT INTO tab1 (col1, col2)
   VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)
```

In the previous example, the database server inserts an incremented value (or the first value of the sequence, which is `1`) into the **col1** and **col2** columns of the table.

You can use NEXTVAL (or CURRVAL) in the SET clause of an UPDATE statement, as the following example shows:

```
UPDATE tab1
   SET col2 = seq_2.NEXTVAL
   WHERE col1 = 1;
```

In the previous example, the incremented value of the **seq_2** sequence, which is `2`, replaces the value in **col2** where **col1** is equal to `1`.

The following example shows how you can use NEXTVAL and CURRVAL in the SELECT clause of a SELECT statement:

```
SELECT seq_2.CURRVAL, seq_2.NEXTVAL FROM tab1;
```

In the previous example, the database server returns two incremented values, `3` and `4`, under both the **currval** and **nextval** columns. For the first row of **tab1**, the database server returns the incremented value `3` for **currval** and **nextval**; for the second row of **tab1**, it returns the incremented value `4`.

You can use NEXTVAL (or CURRVAL) in an SPL routine, as the following example shows:

```
LET var = seq_2.NEXTVAL
```

For more examples on how to use NEXTVAL and CURRVAL, see the *IBM Informix Guide to SQL: Tutorial*

IDS

Literal Collection

The following examples show literal collections as expressions:

```
INSERT INTO tab_a (set_col) VALUES ("SET{6, 9, 3, 12, 4}")

INSERT INTO TABLE(a_set) VALUES (9765)

UPDATE table1 SET set_col = "LIST{3}"

SELECT set_col FROM table1
   WHERE SET{17} IN (set_col)
```

For more information, see “Literal Collection” on page 4-208. For the syntax of element values, see “Collection Constructors” on page 4-108.

IDS

Literal Row

The following examples show literal rows as expressions:

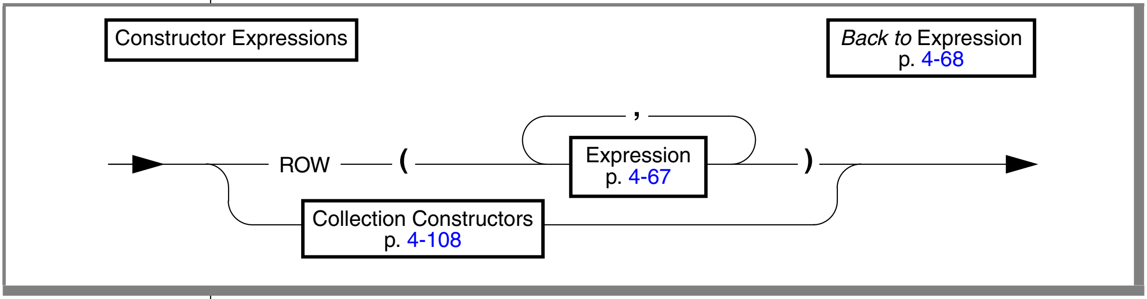
```
INSERT INTO employee VALUES
   (ROW('103 Baker St', 'San Francisco',
      'CA', 94500))

UPDATE rectangles
   SET rect = ROW(8, 3, 7, 20)
   WHERE area = 140

EXEC SQL update table(:a_row)
   set x=0, y=0, length=10, width=20;

SELECT row_col FROM tab_b
   WHERE ROW(17, 'abc') IN (row_col)
```

For more information, see "Literal Row" on page 4-218. For syntax that allows you to use expressions that evaluate to field values, see "ROW Constructors" on page 4-106.

IDS

Constructor Expressions

A *constructor* is a function that the database server uses to create an instance of a specific data type. The database server supports ROW and collection constructors.



ROW Constructors

You use ROW constructors to generate values for ROW-type columns. Suppose you create the following named ROW type and a table that contains the named ROW type **row_t** and an unnamed ROW type:

```
CREATE ROW TYPE row_t ( x INT, y INT);
CREATE TABLE new_tab
(
col1 row_t,
col2 ROW( a CHAR(2), b INT)
)
```

When you define a column as a named ROW type or unnamed ROW type, you must use a ROW constructor to generate values for the ROW-type column. To create a value for either a named ROW type or unnamed ROW type, you must complete the following steps:

- Begin the expression with the ROW keyword.
- Specify a value for each field of the ROW type.
- Enclose the field values within parentheses.

The format of the value for each field must be compatible with the data type of the ROW field to which it is assigned.

You can use any kind of expression as a value with a ROW constructor, including literals, functions, and variables. The following examples show the use of different types of expressions with ROW constructors to specify values:

```
ROW(5, 6.77, 'HMO')

ROW(col1.lname, 45000)

ROW('john davis', TODAY)

ROW(USER, SITENAME)
```

The following statement uses literal numbers and quoted strings with ROW constructors to insert values into **col1** and **col2** of the **new_tab** table:

```
INSERT INTO new_tab
VALUES
(
ROW(32, 65)::row_t,
ROW('CA', 34)
)
```

When you use a ROW constructor to generate values for a named ROW type, you must explicitly cast the ROW value to the appropriate named ROW type. The cast is necessary to generate a value of the named ROW type. To cast the ROW value as a named ROW type, you can use the cast operator (**::**) or the CAST AS keywords, as the following examples show:

```
ROW(4,5)::row_t
CAST (ROW(3,4) AS row_t)
```

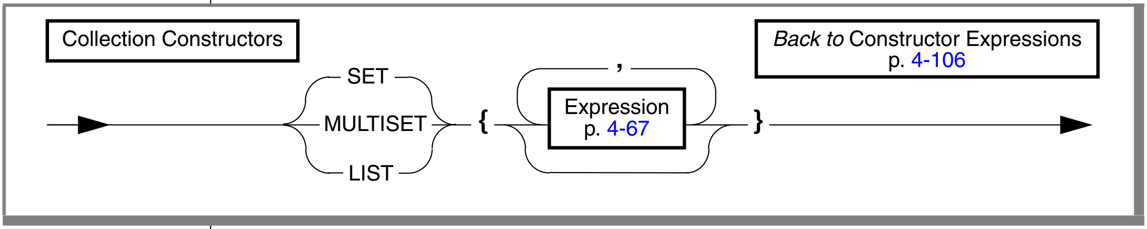
You can use a ROW constructor to generate row type values in INSERT, UPDATE, and SELECT statements. In the next example, the WHERE clause of a SELECT statement specifies a ROW type value that is cast as type **person_t**:

```
SELECT * FROM person_tab
   WHERE col1 = ROW('charlie','hunter')::person_t
```

For more information on using ROW constructors in INSERT and UPDATE statements, see the INSERT and UPDATE statements in this manual. For information on named ROW types, see the CREATE ROW TYPE statement. For information on unnamed ROW types, see the discussion of the ROW data type in the *IBM Informix Guide to SQL: Reference*. For task-oriented information on named ROW types and unnamed ROW types, see the *IBM Informix Database Design and Implementation Guide*.

Collection Constructors

Use a collection constructor to specify values for a collection column.



You can use collection constructors in the WHERE clause of the SELECT statement and the VALUES clause of the INSERT statement. You can also pass collection constructors to UDRs.

This table differentiates the types of collections that you can construct.

Keyword	Description
SET	Indicates a collection of elements with the following qualities: ■ The collection must contain unique values. ■ Elements have no specific order associated with them.
MULTISET	Indicates a collection of elements with the following qualities: ■ The collection can contain duplicate values. ■ Elements have no specific order associated with them.
LIST	Indicates a collection of elements with the following qualities: ■ The collection can contain duplicate values. ■ Elements have ordered positions.

The element type of the collection can be any built-in or extended data type. You can use any kind of expression with a collection constructor, including literals, functions, and variables.

When you use a collection constructor with a list of expressions, the database server evaluates each expression to its equivalent literal form and uses the literal values to construct the collection.

You specify an empty collection with a set of empty braces ({ }).

Elements of a collection cannot be NULL. If a collection element evaluates to a NULL value, the database server returns an error.

The element type of each expression must all be exactly the same type. To accomplish this, cast the entire collection constructor expression to a collection type, or cast individual element expressions to the same type. If the database server cannot determine that the collection type and the element types are not homogeneous, then the collection constructor returns an error. In the case of host variables, this determination is made at bind time when the client declares the element type of the host variable.

An exception to this restriction can occur when some elements of a collection are VARCHAR data types but others are longer than 255 bytes. Here the collection constructor can assign a CHAR(n) type to all elements, for n the length in bytes of the longest element. (But see "Collection Data Types" on page 4-63 for an example based on this exception, where the user avoids fixed-length CHAR elements by an explicit cast to the LVARCHAR data type.)

Examples of Collection Constructors

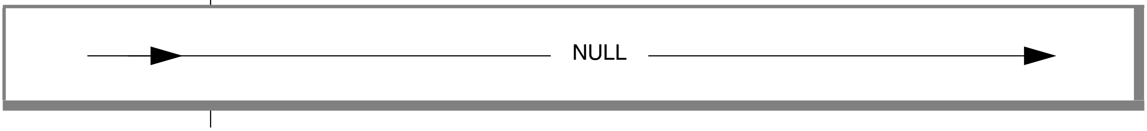
The following example shows that you can construct a collection with many various expressions as long as the resulting values are of the same data type:

```
CREATE FUNCTION f (a int RETURNS int;
   RETURN a+1;
END FUNCTION;
CREATE TABLE tab1 (x SET(INT NOT NULL));
INSERT INTO tab1 VALUES
(
SET{10,
   1+2+3,
   f(10)-f(2),
   SQRT(100) +POW(2,3),
   (SELECT tabid FROM systables WHERE tabname = ‘sysusers’),
   ‘T’::BOOLEAN::INT}
)
SELECT * FROM tab1 WHERE
x=SET{10,
   1+2+3,
   f(10)-f(2),
   SQRT(100) +POW(2,3),
   (SELECT tabid FROM systables WHERE tabname = ‘sysusers’),
   ‘T’::BOOLEAN::INT}
}
```

This assumes that a cast from BOOLEAN to INT exists. (For a more restrictive syntax to specify collection values , see "Literal Collection" on page 4-208.)

NULL Keyword

The NULL keyword is valid in most contexts where you can specify a value. What it specifies, however, is the absence of any value (or an unknown or missing value).



Within SQL, the keyword NULL is the only syntactic mechanism for accessing a NULL value. NULL is not equivalent to zero, nor to any specific value. In ascending ORDER BY operations, NULL values precede any non-NULL value; in descending sorts, NULL values follow any non-NULL value. In GROUP BY operations, all NULL values are grouped together. (Such groups may in fact be logically heterogeneous, if they include missing or unknown values.)

The keyword NULL is a global symbol in the syntactic context of expressions, meaning that its scope of reference is global.

Every data type, whether built-in or user-defined, can represent a NULL value. IBM Informix Dynamic Server supports cast expressions in the SELECT list. This means that users can write expressions of the form NULL**::***datatype*, in which *datatype* is any data type known to the database server.

IBM Informix Dynamic Server prohibits the redefinition of NULL because allowing such definition would restrict the global scope of the NULL keyword. For this reason, any mechanism that restricts the global scope or redefines the scope of the keyword NULL will syntactically disable any cast expression involving a NULL value. You must ensure that the occurrence of the keyword NULL receives its global scope in all expression contexts.

For example, consider the following SQL code:

```
CREATE TABLE newtable
(
null int
);

SELECT null, null::int FROM newtable;
```

The CREATE TABLE statement is valid, because the column identifiers have a scope of reference that is restricted to the table definition; they can be accessed only within the scope of a table.

The SELECT statement in the example, however, poses some syntactic ambiguities. Does the identifier **null** appearing in the SELECT list refer to the global keyword NULL, or does it refer to the column name **null** that was defined in the CREATE TABLE statement?

- If the identifier **null** is interpreted as the column name, the global scope of cast expressions with the NULL keyword will be restricted.
- If the identifier **null** is interpreted as the NULL keyword, the SELECT statement must generate a syntactic error for the first occurrence of **null** because the NULL keyword can appear only as a cast expression in the select list.

A SELECT statement of the following form is valid because the NULL column of **newtable** is qualified with the table name:

```
SELECT newtable.null, null::int FROM newtable;
```

More involved syntactic ambiguities arise in the context of an SPL routine that has a variable named **null**. An example follows:

```
CREATE FUNCTION nulltest() RETURNING INT;
DEFINE a INT;
DEFINE null INT;
DEFINE b INT;
LET a = 5;
LET null = 7;
LET b = null;
RETURN b;
END FUNCTION;

EXECUTE FUNCTION nulltest();
```

When the preceding function executes in DB-Access, in the expressions of the LET statement, the identifier **null** is treated as the keyword NULL. The function returns a NULL value instead of 7.

Using **null** as a variable of an SPL routine would restrict the use of a NULL value in the body of the SPL routine. Therefore, the preceding SPL code is not valid, and causes IBM Informix Dynamic Server to return the following error:

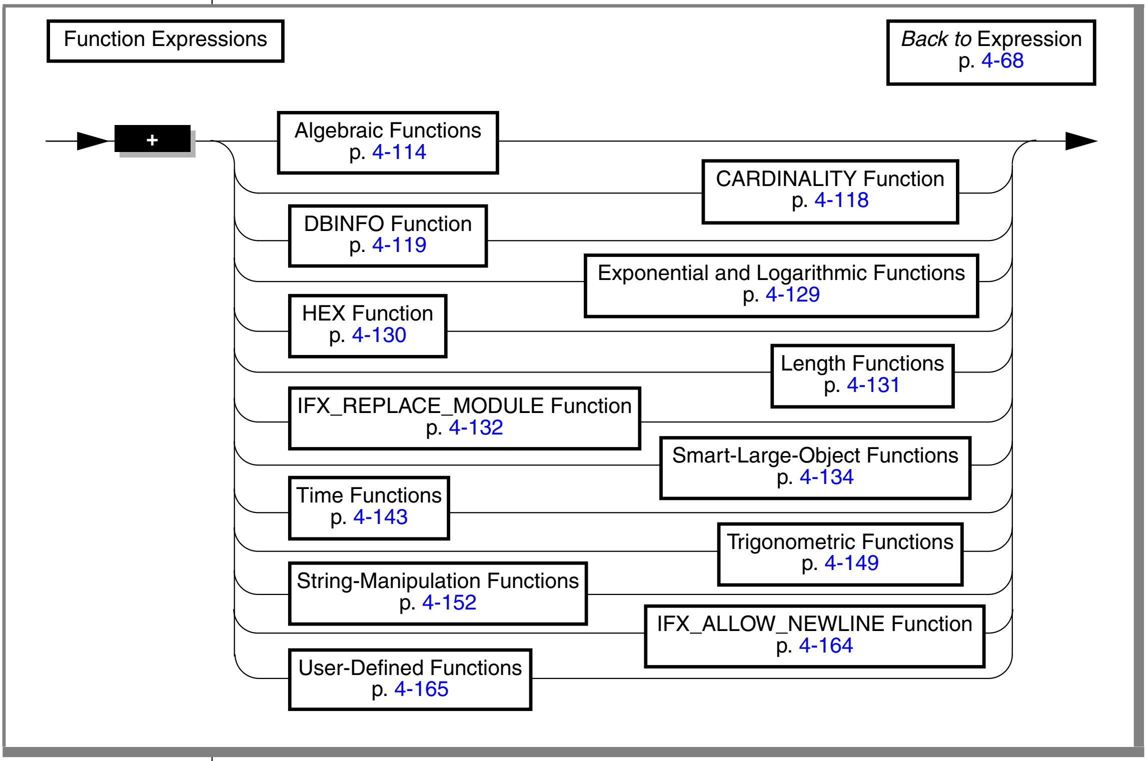
```
-947   Declaration of an SPL variable named 'null' conflicts
with SQL NULL value.
```

E/C

You should use an indicator variable if there is the possibility that a SELECT statement will return a NULL value. ♦

Function Expressions

A function expression can call built-in functions or user-defined functions, as the following diagram shows.

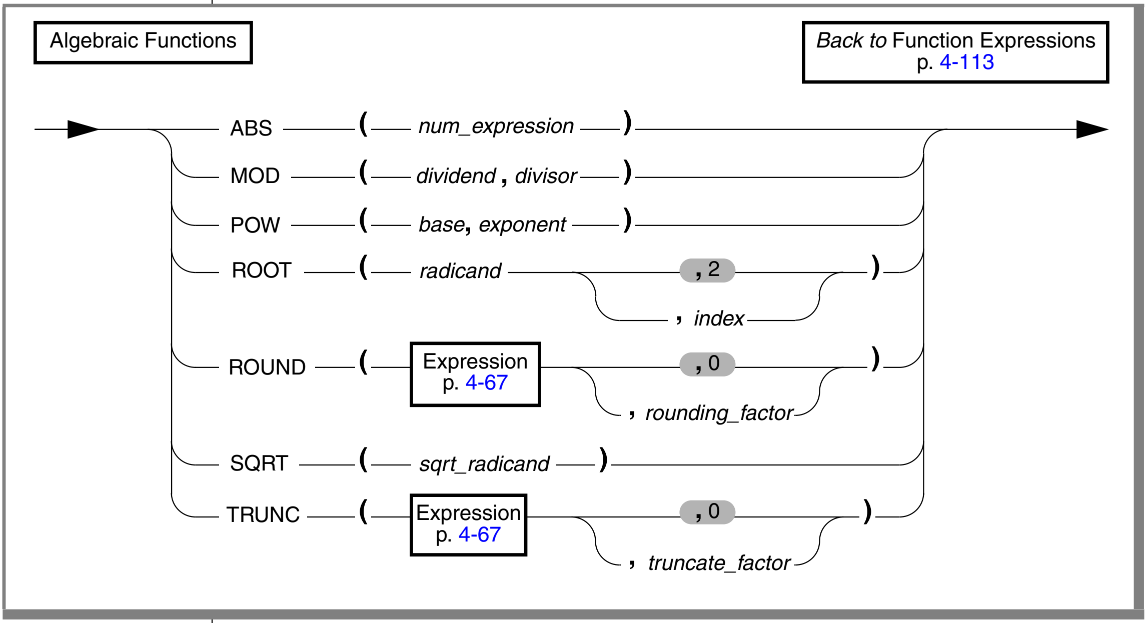


The following examples show function expressions:

```
EXTEND (call_dtime, YEAR TO SECOND)
MDY (12, 7, 1900 + cur_yr)
DATE (365/2)
LENGTH ('abc') + LENGTH (pvar)
HEX (customer_num)
HEX (LENGTH(123))
TAN (radians)
ABS (-32)
EXP (3)
MOD (10,3)
```

Algebraic Functions

Algebraic functions take one or more arguments of numeric data types.


Algebraic Functions
Back to Function Expressions p. 4-113
ABS (num_expression)
MOD (dividend , divisor)
POW (base, exponent)
ROOT (radicand , 2 , index)
ROUND (Expression p. 4-67 , 0 , rounding_factor)
SQRT (sqrt_radicand)
TRUNC (Expression p. 4-67 , 0 , truncate_factor)

Element	Purpose	Restrictions	Syntax
base	Value to be raised to the power specified in *exponent*	Must return a real number.	Expression, p. 4-67
dividend	Value to be divided by *divisor*	Must return a real number.	Expression, p. 4-67
divisor	Value by which to divide *dividend*	Must return a nonzero number.	Expression, p. 4-67
exponent	Power to which to raise *base*	Must return a real number.	Expression, p. 4-67
index	Root to extract. The default is 2.	Must return a nonzero number.	Expression, p. 4-67
num_expression	Number with an absolute value	Must return a real number.	Expression, p. 4-67
radicand	Value whose root is to be returned	Must return a real number.	Expression, p. 4-67
rounding_factor	Position to which a number is to be rounded. The default is zero.	Integer in range +32 to -32; see "ROUND Function" on page 116.	Literal Number, p. 4-216
sqrt_radicand	Number with real square roots	Must be a positive real number.	Expression, p. 4-67
truncate_factor	Position to which a number is to be truncated. The default is zero.	Integer in range +32 to -32; see "TRUNC Function" on page 117.	Literal Number, p. 4-216

ABS Function

The **ABS** function returns the absolute value of a numeric expression, returning the same data type as its single argument. The following example shows all orders of more than $20 paid in cash (+) or store credit (-). The stores_demo database does not contain any negative balances, but you might have negative balances in your application.

```
SELECT order_num, customer_num, ship_charge
  FROM orders WHERE ABS(ship_charge) > 20
```

MOD Function

The **MOD** function returns the remainder from integer division of two real number operands, after the integer part of the first argument (the *dividend*) is divided by the integer part of the second argument (the *divisor*) as an INT data type (or INT8 on Dynamic Server, for remainders outside the range of INT). The quotient and any fractional part of the remainder are discarded. The *divisor* cannot be 0. Thus, `MOD (x,y)` returns `y (modulo x)`. Make sure that any variable that receives the result can store the returned value.

This example tests to see if the current date is within a 30-day billing cycle:

```
SELECT MOD(TODAY - MDY(1,1,YEAR(TODAY)),30) FROM orders
```

POW Function

The **POW** function raises the *base* to the *exponent.* This function requires two numeric arguments. The returned data type is FLOAT. The following example returns data for circles whose areas are less than 1,000 square units:

```
SELECT * FROM circles WHERE (3.1416 * POW(radius,2)) < 1000
```

To use *e*, the base of natural logarithms, see "EXP Function" on page 4-129.

ROOT Function

The **ROOT** function returns the root value of a numeric expression. This function requires at least one numeric argument (the *radicand* argument) and allows no more than two (the *radicand* and *index* arguments). If only the *radicand* argument is supplied, the value `2` is used as a default value for the *index* argument. The value `0` cannot be used as the value of *index*. The value that the **ROOT** function returns is a FLOAT data type.

The first SELECT statement in the following example takes the square root of the expression. The second takes the cube root of the expression.

```
SELECT ROOT(9) FROM angles          -- square root of 9
SELECT ROOT(64,3) FROM angles       -- cube root of 64
```

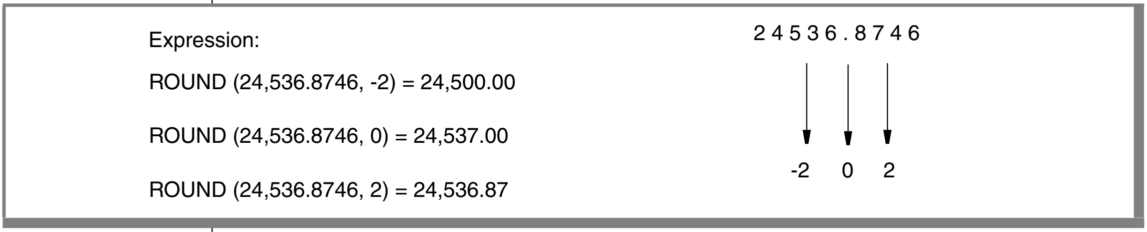
The **SQRT** function uses the form SQRT(x)=ROOT(x) if no index is given.

ROUND Function

The **ROUND** function returns the rounded value of an expression. The expression must be numeric or must be converted to numeric. If you omit the digit indication, the value is rounded to zero digits or to the units place. The digit range of 32 (+ and -) refers to the entire decimal value.

Positive-digit values indicate rounding to the right of the decimal point; negative-digit values indicate rounding to the left of the decimal point, as Figure 4-1 shows.

Figure 4-1
ROUND Function



Expression:

ROUND (24,536.8746, -2) = 24,500.00

ROUND (24,536.8746, 0) = 24,537.00

ROUND (24,536.8746, 2) = 24,536.87

The following example shows how you can use the **ROUND** function with a column expression in a SELECT statement. This statement displays the order number and rounded total price (to zero places) of items whose rounded total price (to zero places) is equal to 124.00.

```
SELECT order_num , ROUND(total_price) FROM items
   WHERE ROUND(total_price) = 124.00
```

If you use a MONEY data type as the argument for the **ROUND** function and you round to zero places, the value displays with .00. The SELECT statement in the following example rounds an INTEGER value and a MONEY value. It displays 125 and a rounded price in the form xxx.00 for each row in **items**.

```
SELECT ROUND(125.46), ROUND(total_price) FROM items
```

SQRT Function

The **SQRT** function returns the square root of a numeric expression. The next example returns the square root of 9 for each row of the angles table:

```
SELECT SQRT(9) FROM angles
```

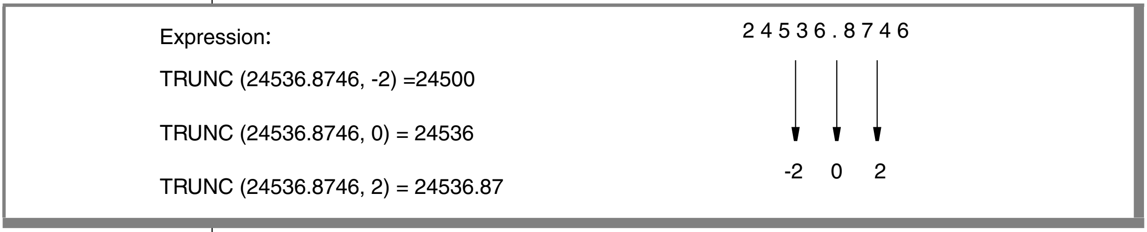
TRUNC Function

The **TRUNC** function returns the truncated value of a numeric expression.

The expression must be numeric or a form that can be converted to a numeric expression. If you omit the digit indication, the value is truncated to zero digits or to the unit's place. The digit limitation of 32 (+ and -) refers to the entire decimal value.

Positive digit values indicate truncating to the right of the decimal point; negative digit values indicate truncating to the left, as Figure 4-2 shows.

Figure 4-2
TRUNC Function



Expression:

TRUNC (24536.8746, -2) =24500

TRUNC (24536.8746, 0) = 24536

TRUNC (24536.8746, 2) = 24536.87

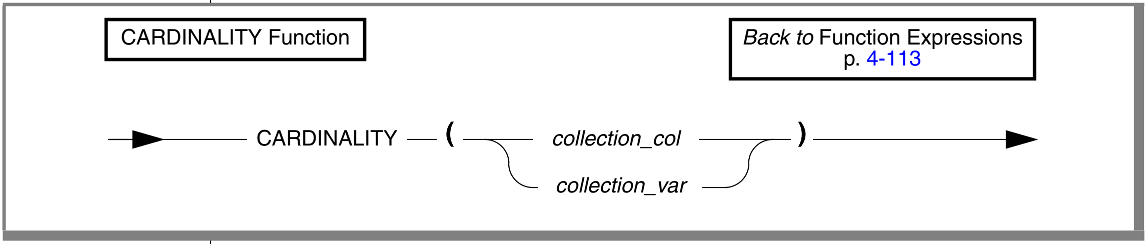
If a MONEY data type is the argument for the **TRUNC** function that specifies zero places, the fractional places are removed. For example, the following SELECT statement truncates a MONEY value and an INTEGER value. It displays 125 and a truncated price in integer format for each row in **items**.

```
SELECT TRUNC(125.46), TRUNC(total_price) FROM items
```

IDS

CARDINALITY Function

The **CARDINALITY** function returns the number of elements in a collection column (SET, MULTISET, LIST).



Element	Purpose	Restrictions	Syntax
collection_col	An existing collection column	Must return an integer.	Expression, p. 4-67
collection_var	Host or program collection variable	Must exist.	Language specific

Suppose that the **set_col** SET column contains the following value:

```
{3, 7, 9, 16, 0}
```

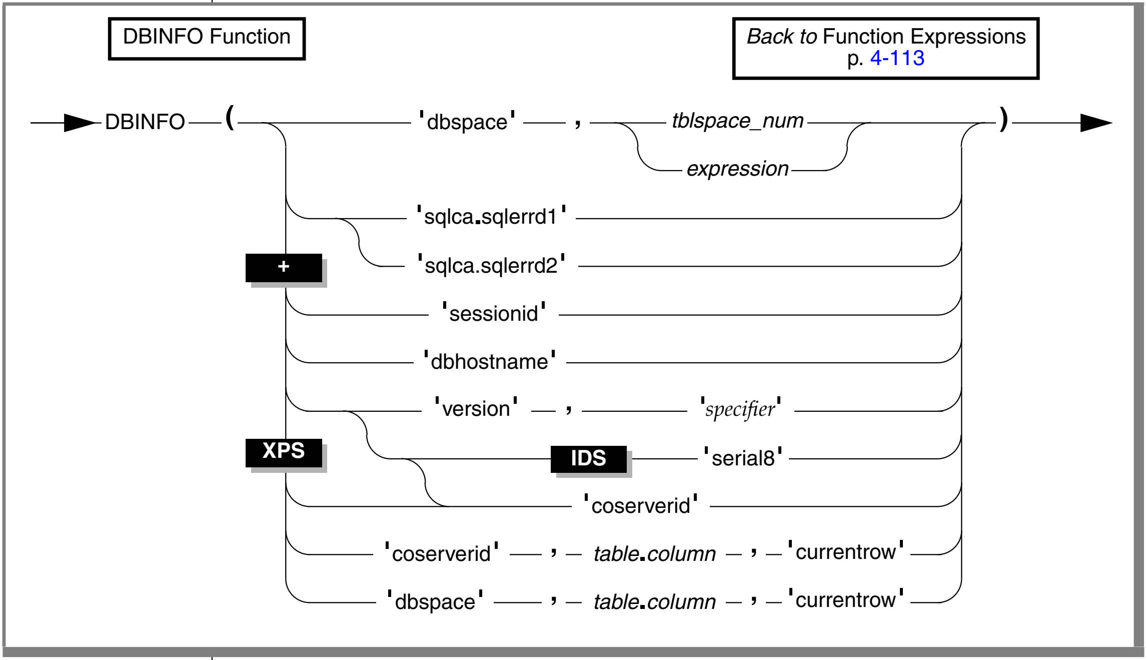
The following SELECT statement returns 5 as the number of elements in the **set_col** column:

```
SELECT CARDINALITY(set_col)
   FROM table1
```

If the collection contains duplicate elements, **CARDINALITY** counts each individual element.

DBINFO Function

The following diagram shows the syntax of the **DBINFO** function.



Element	Purpose	Restrictions	Syntax
column	Name of a column in the table that you specify in *table*	Must exist in *table*.	Database Object Name, p. 4-46
expression	Expression that evaluates to *tblspace_num*	Can contain column names, SPL variables, host variables, or subqueries, but must return a numeric value.	Expression, p. 4-67
specifier	Literal value that specifies which part of version string to return	For valid *specifier* values, see "Using the 'version' Option" on page 4-124.	Expression, p. 4-67
table	Table for which to display the dbspace name or coserver ID corresponding to each row	Must match the name of a table in the FROM clause of the query.	Database-Object Name, p. 4-46
tblspace_ num	Tblspace number (partition number) of a table	Must exist in the **partnum** column of the **systables** table for the database.	Literal Number, p. 4-216

DBINFO Options

The **DBINFO** function is actually a set of functions that return different types of information about the database. To invoke each function, specify a particular option after the **DBINFO** keyword. You can use any **DBINFO** option anywhere within SQL statements and within UDRs.

The following table shows the different types of database information that you can retrieve with the **DBINFO** options. The **Option** column shows the name of each **DBINFO** option. The **Effect** column shows the type of database information that the option retrieves.

The **Page** column shows the page where you can find more information about a given option.

Option	Effect	Page
'dbspace' *tblspace_num*	Returns the name of a dbspace corresponding to a tblspace number	4-121
'sqlca.sqlerrd1'	Returns the last serial value inserted in a table	4-121
'sqlca.sqlerrd2'	Returns the number of rows processed by selects, inserts, deletes, updates, EXECUTE PROCEDURE statements, and EXECUTE FUNCTION statements	4-122
'sessionid'	Returns the session ID of the current session	4-122
'dbhostname'	Returns the hostname of the database server to which a client application is connected	4-123
'version'	Returns the exact version of the database server to which a client application is connected	4-124
'serial8' (IDS)	Returns last SERIAL8 value inserted in a table	4-126
'coserverid' (XPS)	Returns the coserver ID of coserver to which the user who entered the query is connected	4-126
'coserverid' *table.column* 'currentrow' (XPS)	Returns the coserver ID of the coserver where each row of a specified table is located	4-127
'dbspace' *table.column* 'currentrow' (XPS)	Returns the name of the dbspace where each row of a specified table is located	4-128

Using the 'dbspace' Option Followed by a Tblspace Number

The **'dbspace'** option returns a character string containing the name of the dbspace that corresponds to a tblspace number. You must supply an additional parameter, either *tblspace_num* or an expression that evaluates to *tblspace_num*. The next example uses the **'dbspace'** option. First, it queries the **systables** system catalog table to determine the *tblspace_num* for the table **customer**, then it executes the function to determine the dbspace name.

```
SELECT tabname, partnum FROM systables
   where tabname = 'customer'
```

If the statement returns a partition number of 1048892, you insert that value into the second argument to find which dbspace contains the **customer** table, as the following example shows:

```
SELECT DBINFO ('dbspace', 1048892) FROM systables
   where tabname = 'customer'
```

If the table for which you want to know the dbspace name is fragmented, you must query the **sysfragments** system catalog table to find out the tblspace number of each table fragment. Then you must supply each tblspace number in a separate **DBINFO** query to find out all the dbspaces across which a table is fragmented.

Using the 'sqlca.sqlerrd1' Option

The **'sqlca.sqlerrd1'** option returns a single integer that provides the last serial value that is inserted into a table. To ensure valid results, use this option immediately following a singleton INSERT statement that inserts a single row with a serial value into a table.

Tip: *To obtain the value of the last SERIAL8 value that is inserted into a table, use the* ***'serial8'*** *option of* **DBINFO***. For more information, see "Using the 'serial8' Option" on page 4-126.*

The following example uses the **'sqlca.sqlerrd1'** option:

```
EXEC SQL create table fst_tab (ordernum serial, partnum int);
EXEC SQL create table sec_tab (ordernum serial);
EXEC SQL insert into fst_tab VALUES (0,1);
EXEC SQL insert into fst_tab VALUES (0,4);
EXEC SQL insert into fst_tab VALUES (0,6);
EXEC SQL insert into sec_tab values (dbinfo('sqlca.sqlerrd1'));
```

This example inserts a row that contains a primary-key serial value into the **fst_tab** table, and then uses the **DBINFO** function to insert the same serial value into the **sec_tab** table. The value that the **DBINFO** function returns is the serial value of the last row that is inserted into **fst_tab**.

Using the 'sqlca.sqlerrd2' Option

The **'sqlca.sqlerrd2'** option returns a single integer that provides the number of rows that SELECT, INSERT, DELETE, UPDATE, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements processed. To ensure valid results, use this option after SELECT, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements have completed executing. In addition, to ensure valid results when you use this option within cursors, make sure that all rows are fetched before the cursors are closed.

The following example shows an SPL routine that uses the **'sqlca.sqlerrd2'** option to determine the number of rows that are deleted from a table:

```
CREATE FUNCTION del_rows (pnumb int)
RETURNING int;

DEFINE nrows int;

DELETE FROM fst_tab WHERE part_number = pnumb;
LET nrows = DBINFO('sqlca.sqlerrd2');
RETURN nrows;

END FUNCTION
```

Using the 'sessionid' Option

The **'sessionid'** option of the **DBINFO** function returns the session ID of your current session. When a client application makes a connection to the database server, the database server starts a session with the client and assigns a session ID for the client. The session ID serves as a unique identifier for a given connection between a client and a database server.

The database server stores the value of the session ID in a data structure in shared memory that is called the *session control block*. The session control block for a given session also includes the user ID, the process ID of the client, the name of the host computer, and a variety of status flags.

When you specify the **'sessionid'** option, the database server retrieves the session ID of your current session from the session control block and returns this value to you as an integer. Some of the System-Monitoring Interface (SMI) tables in the **sysmaster** database include a column for session IDs, so you can use the session ID that the **DBINFO** function obtained to extract information about your own session from these SMI tables. For further information on the session control block, see the *Administrator's Guide*. For further information on the **sysmaster** database and the SMI tables, see the *Administrator's Reference*.

In the following example, the user specifies the **DBINFO** function in a SELECT statement to obtain the value of the current session ID. The user poses this query against the **systables** system catalog table and uses a WHERE clause to limit the query result to a single row.

```
SELECT DBINFO('sessionid') AS my_sessionid
   FROM systables
   WHERE tabname = 'systables'
```

In the preceding example, the SELECT statement queries against the **systables** system catalog table. You can, however, obtain the session ID of the current session by querying against any system catalog table or user table in the database. For example, you can enter the following query to obtain the session ID of your current session:

```
SELECT DBINFO('sessionid') AS user_sessionid
   FROM customer
   WHERE customer_num = 101
```

You can use the **DBINFO 'sessionid'** option not only in SQL statements but also in SPL routines. The following example shows an SPL function that returns the value of the current session ID to the calling program or routine:

```
CREATE FUNCTION get_sess()
   RETURNING INT;
   RETURN DBINFO('sessionid');
END FUNCTION;
```

Using the 'dbhostname' Option

You can use the **'dbhostname'** option to retrieve the hostname of the database server to which a database client is connected. This option retrieves the physical computer name of the computer on which the database server is running.

In the following example, the user enters the **'dbhostname'** option of **DBINFO** in a SELECT statement to retrieve the hostname of the database server to which DB-Access is connected:

```
SELECT DBINFO('dbhostname')
   FROM systables
   WHERE tabid = 1
```

The following table shows the result of this query.

(constant)
rd_lab1

Using the 'version' Option

You can use the **'version'** option of the **DBINFO** function to retrieve the exact version number of the database server against which the client application is running. This option retrieves the exact version string from the message log. The value of the full version string is the same as that displayed by the **-V** option of the **oninit** utility.

Use the *specifier* parameter of the **'version'** option to specify which part of the version string you want to retrieve. The following table lists the values that you can enter in the *specifier* parameter, shows which part of the version string is returned for each *specifier* value, and gives an example of what is returned by each value of *specifier*.

Each example returns part of the complete version string Dynamic Server `Version 9.40.UC1`.

Specifier Parameter	**Part of Version String Returned**	**Example of Returned Value**
'server-type'	Type of database server	`Dynamic Server`
'major'	Major version number of the current database server version	9
'minor'	Minor version number of the current database server version	40

(1 of 2)

Specifier Parameter	Part of Version String Returned	Example of Returned Value
'os'	Operating-system identifier within the version string: `T` = Windows `U` = UNIX 32-bit running on a 32-bit operating system `H` = UNIX 32-bit running on a 64-bit operating system `F` = UNIX 64-bit running on a 64-bit operating system	`U`
'level'	Interim release level of the current database server version	C1
'full'	Complete version string as it would be returned by **oninit -V**	`Dynamic Server, Version 9.40.UC1`

(2 of 2)

Important: *Not all UNIX environments fit the word-length descriptions of operating- system (*os*) codes in the preceding table. For example, some* `U` *versions can run on 64-bit operating systems. Similarly, some* `F` *versions can run on operating systems with 32-bit kernels that support 64-bit applicatio*ns.

The following example shows how to use the **'version'** option of **DBINFO** in a SELECT statement to retrieve the major version number of the database server that the DB-Access client is connected to:

```
SELECT DBINFO('version', 'major')
   FROM systables
   WHERE tabid = 1
```

The following table shows the result of this query.

(constant)
7

IDS

Using the 'serial8' Option

The **'serial8'** option returns a single integer that provides the last SERIAL8 value that is inserted into a table. To ensure valid results, use this option immediately following an INSERT statement that inserts a SERIAL8 value.

Tip: *To obtain the value of the last SERIAL value that is inserted into a table, use the* ***'sqlca.sqlerrd1'*** *option of* ***DBINFO()****. For more information, see "Using the 'sqlca.sqlerrd1' Option" on page 4-121.*

The following example uses the **'serial8'** option:

```
EXEC SQL create table fst_tab
   (ordernum serial8, partnum int);
EXEC SQL create table sec_tab (ordernum serial8);

EXEC SQL insert into fst_tab VALUES (0,1);
EXEC SQL insert into fst_tab VALUES (0,4);
EXEC SQL insert into fst_tab VALUES (0,6);

EXEC SQL insert into sec_tab
   select dbinfo('serial8')
   from sec_tab where partnum = 6;
```

This example inserts a row that contains a primary-key SERIAL8 value into the **fst_tab** table and then uses the **DBINFO** function to insert the same SERIAL8 value into the **sec_tab** table. The value that the **DBINFO** function returns is the SERIAL8 value of the last row that is inserted into **fst_tab**. The subquery in the last line contains a WHERE clause so that a single value is returned.

XPS

Using the 'coserverid' Option with No Other Arguments

The **'coserverid'** option with no other arguments returns a single integer that corresponds to the coserver ID of the coserver to which the user who entered the query is connected.

Suppose that you use the following statement to create the **mytab** table:

```
CREATE TABLE mytab (mycol INT)
   FRAGMENT BY EXPRESSION
      mycol < 5 in rootdbs.1
      mycol > 5 in rootdbs.2
```

Further, suppose that the dbspace named **rootdbs.1** resides on coserver 1, and the dbspace named **rootdbs.2** resides on coserver 2. Also suppose that you use the following statements to insert rows into the **mytab** table:

```
INSERT INTO mytab VALUES ('1');
INSERT INTO mytab VALUES ('6');
```

Finally, suppose that you are logged on to coserver 1 when you make the following query, which displays the values of all columns in the row where the value of the **mycol** column is 1. This query also displays the coserver ID of the coserver to which you are logged on when you enter the query.

```
SELECT *, DBINFO ('coserverid') AS cid
   FROM mytab
   WHERE mycol = 1
```

The following table shows the result of this query.

myco	**l**	**cid**
1		1

XPS

Using the 'coserverid' Option Followed by Table and Column Names

Use the **'coserverid'** option followed by the table name and column name and the **'currentrow'** string to find out the coserver ID where each row in a specified table is located. This option is especially useful when you fragment a table across multiple coservers.

In the following example, the user asks to see all columns and rows of the **mytab** table as well as the coserver ID of the coserver where each row resides. For a description of the **mytab** table, see "Using the 'coserverid' Option with No Other Arguments" on page 4-126.

```
SELECT *, DBINFO ('coserverid', mytab.mycol, 'currentrow')
   AS cid
   FROM mytab
```

The following table shows the result of this query.

mycol	cid
1	1
6	2

The column that you specify in the **DBINFO** function can be any column in the specified table.

XPS

Using the 'dbspace' Option Followed by Table and Column Names

Use the **'dbspace '** option followed by the table name and column name and the **'currentrow'** string to find out the name of the dbspace where each row in a specified table is located. This option is especially useful when you fragment a table across multiple dbspaces.

In the following example, the user asks to see all columns and rows of the **mytab** table as well as the name of the dbspace where each row resides. For a description of the **mytab** table, see "Using the 'coserverid' Option with No Other Arguments" on page 4-126.

```
SELECT *, DBINFO ('dbspace', mytab.mycol, 'currentrow')
   AS dbsp
   FROM mytab
```

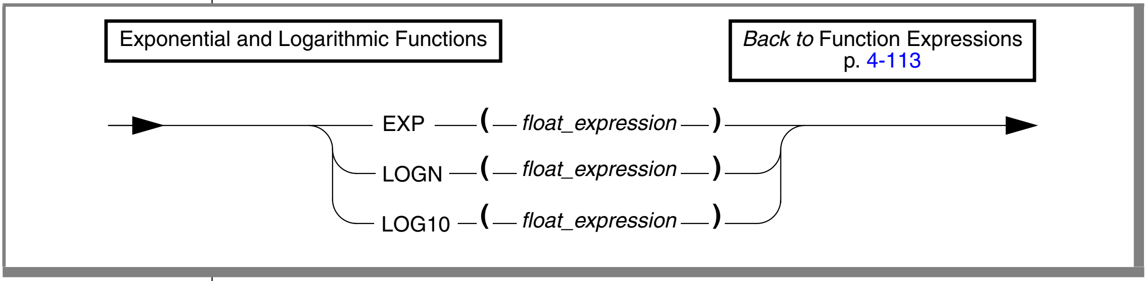
The following table shows the result of this query.

mycol	dbspace
1	rootdbs.1
6	rootdbs.2

The column that you specify in the **DBINFO** function can be any column in the specified table.

Exponential and Logarithmic Functions

Exponential and logarithmic functions take at least one argument and return a FLOAT data type.


Exponential and Logarithmic Functions
Back to Function Expressions p. 4-113
EXP (float_expression)
LOGN (float_expression)
LOG10 (float_expression)

Element	Purpose	Restrictions	Syntax
float_expression	An argument to the **EXP**, **LOGN**, or **LOG10** functions. For the meaning of *float_expression* in these functions, see the individual heading for each function on the pages that follow.	The domain is the set of real numbers, and the range is the set of positive real numbers.	Expression, p. 4-67

EXP Function

The **EXP** function returns the exponent of a numeric expression. The following example returns the exponent of 3 for each row of the angles table:

```
SELECT EXP(3) FROM angles
```

For this function, the base is always *e*, the base of natural logarithms, as the following example shows:

```
e=exp(1)=2.718281828459
```

When you want to use the base of natural logarithms as the base value, use the **EXP** function. If you want to specify a particular value to raise to a specific power, see the "POW Function" on page 4-115.

LOG10 Function

The **LOG10** function returns the log of a value to base 10. The following example returns the log base 10 of distance for each row of the travel table:

```
SELECT LOG10(distance) + 1 digits FROM travel
```

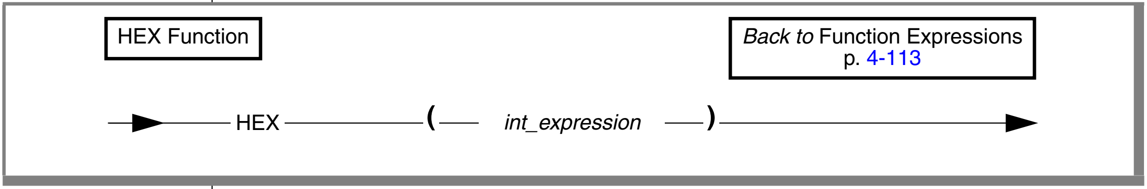
LOGN Function

The **LOGN** function returns the natural logarithm of a numeric argument. This value is the inverse of the exponential value. The following query returns the natural log of **population** for each row of the history table:

```
SELECT LOGN(population) FROM history WHERE country='US'
   ORDER BY date
```

HEX Function

The **HEX** function returns the hexadecimal encoding of an integer expression.



Element	Purpose	Restrictions	Syntax
int_expression	Expression for which you want the hexadecimal equivalent	Must be a literal integer or some other expression that returns an integer.	Expression, p. 4-67

The next example displays the data type and column length of the columns of the **orders** table in hexadecimal format. For MONEY and DECIMAL columns, you can then determine the precision and scale from the lowest and next-to-the-lowest bytes. For VARCHAR and NVARCHAR columns, you can determine the minimum space and maximum space from the lowest and next-to-the-lowest bytes. For more information about encoded information, see the *IBM Informix Guide to SQL: Reference*.

```
SELECT colname, coltype, HEX(collength)
   FROM syscolumns C, systables T
   WHERE C.tabid = T.tabid AND T.tabname = 'orders'
```

The following example lists the names of all the tables in the current database and their corresponding tblspace number in hexadecimal format.

```
SELECT tabname, HEX(partnum) FROM systables
```

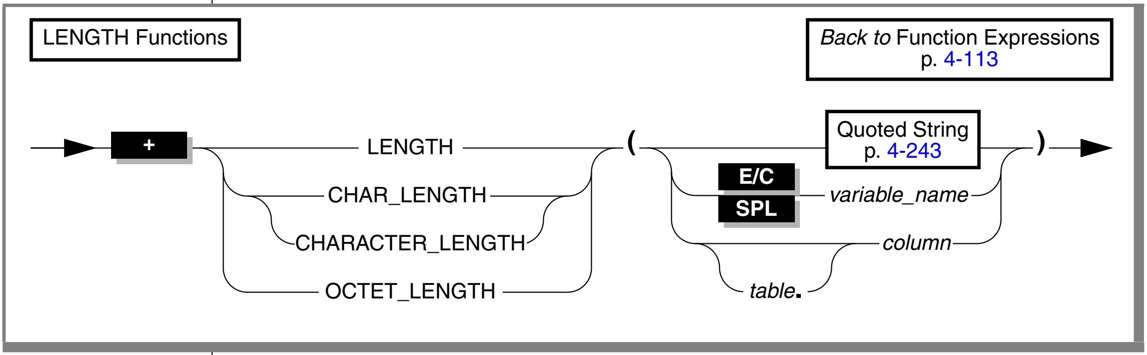
The two most significant bytes in the hexadecimal number constitute the dbspace number. They identify the table in **oncheck** output (in Dynamic Server) and in **onutilcheck** output (in Extended Parallel Server).

The **HEX** function can operate on an expression, as the next example shows:

```
SELECT HEX(order_num + 1) FROM orders
```

Length Functions

Use length functions to determine the length of a column, string, or variable.



Element	Purpose	Restrictions	Syntax
column	Name of a column in *table*	Must have a character data type.	Identifier, p. 4-189
table	Name of the table in which the specified column occurs	The table must exist.	Database- Object Name, p. 4-46
variable_name	Host variable or SPL variable that contains a character string	Variable must have a character data type.	See language-specific rules for names.

Each of these functions has a distinct purpose:

- **LENGTH**
- **OCTET_LENGTH**
- **CHAR_LENGTH** (also known as **CHARACTER_LENGTH**)

The LENGTH Function

The **LENGTH** function returns the number of bytes in a character column, not including any trailing blank spaces. For BYTE or TEXT columns, **LENGTH** returns the full number of bytes, including any trailing blank spaces.

E/C

In ESQL/C, **LENGTH** can also return the length of a character variable. ♦

The next example illustrates the use of the **LENGTH** function:

```
SELECT customer_num, LENGTH(fname) + LENGTH(lname),
   LENGTH('How many bytes is this?')
   FROM customer WHERE LENGTH(company) > 10
```

GLS

See also the discussion of **LENGTH** in the *IBM Informix GLS User's Guide*. ♦

The OCTET_LENGTH Function

OCTET_LENGTH returns the number of bytes in a character column, including any trailing spaces. See also the *IBM Informix GLS User's Guide*.

The CHAR_LENGTH Function

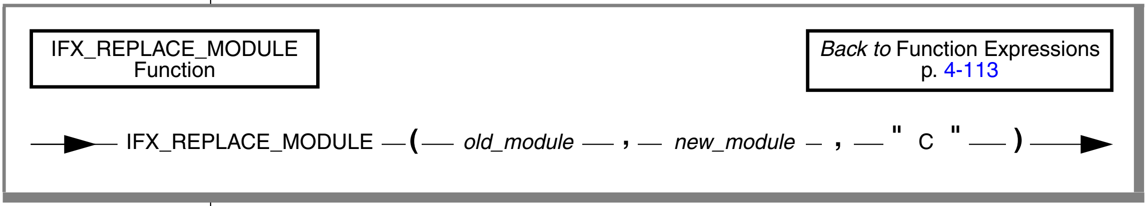
The **CHAR_LENGTH** function (also called **CHARACTER_LENGTH**) returns the number of logical characters (which can be distinct from the number of bytes in some East Asian locales) in a character column. For a discussion of this function, see the *IBM Informix GLS User's Guide*.

IDS

C

IFX_REPLACE_MODULE Function

The **IFX_REPLACE_MODULE** function replaces a loaded shared-object file with a new version that has a different name or location.


IFX_REPLACE_MODULE Function
Back to Function Expressions p. 4-113
IFX_REPLACE_MODULE
(
old_module
,
new_module
,
" C "
)

Element	Purpose	Restrictions	Syntax
new_module	Full pathname of the new shared-object file to replace the shared-object file that *old_module* specifies	The shared-object file must exist with the specified pathname, which can be no more than 255 bytes long.	Quoted String, p. 4-243
old_module	Full pathname of the shared-object file to replace with the shared-object file that *new_module* specifies	The shared-object file must exist with the specified pathname, which can be no more than 255 bytes long.	Quoted String, p. 4-243

The **IFX_REPLACE_MODULE** function returns an integer value to indicate the status of the shared-object-file replacement, as follows:

- Zero (`0`) to indicate success
- A negative integer to indicate an error

Important: *Do not use the* ***IFX_REPLACE_MODULE*** *function to reload a module of the same name. If the full names of the old and new modules that you send to* ***ifx_replace_module( )*** *are the same, then unpredictable results can occur.*

After **IFX_REPLACE_MODULE** completes execution, the database server "ages out" the *old_module* shared-object file; that is, all statements subsequent to the **IFX_REPLACE_MODULE** function will use UDRs in the *new_module* shared-object file, and the old module will be unloaded when any statements that were using it are complete. Thus, for a brief time, both the *old_module* and the *new_module* shared-object files could be resident in memory. If this "aging out" behavior is undesirable, use the **IFX_UNLOAD_MODULE** procedure to unload the shared-object file completely.

UNIX

For example, suppose you want to replace the **circle.so** shared library, which contains UDRs written in the C language. If the old version of this library resides in the **/usr/apps/opaque_types** directory and the new version in the **/usr/apps/shared_libs** directory, then the following EXECUTE FUNCTION statement executes the **IFX_REPLACE_MODULE** function:

```
EXECUTE FUNCTION ifx_replace_module(
   "/usr/apps/opaque_types/circle.so",
   "/usr/apps/shared_libs/circle.so", "C")
```

♦

Windows

For example, suppose you want to replace the **circle.dll** dynamic link library, which contains C UDRs. If the old version of this library resides in the C:**\usr\apps\opaque_types** directory and the new version in the C:**\usr\apps\DLLs** directory, then the following EXECUTE FUNCTION statement executes the **IFX_REPLACE_MODULE** function:

```
EXECUTE FUNCTION ifx_replace_module(
   "C:\usr\apps\opaque_types\circle.dll",
   "C:\usr\apps\DLLs\circle.dll", "C")
```

♦

E/C

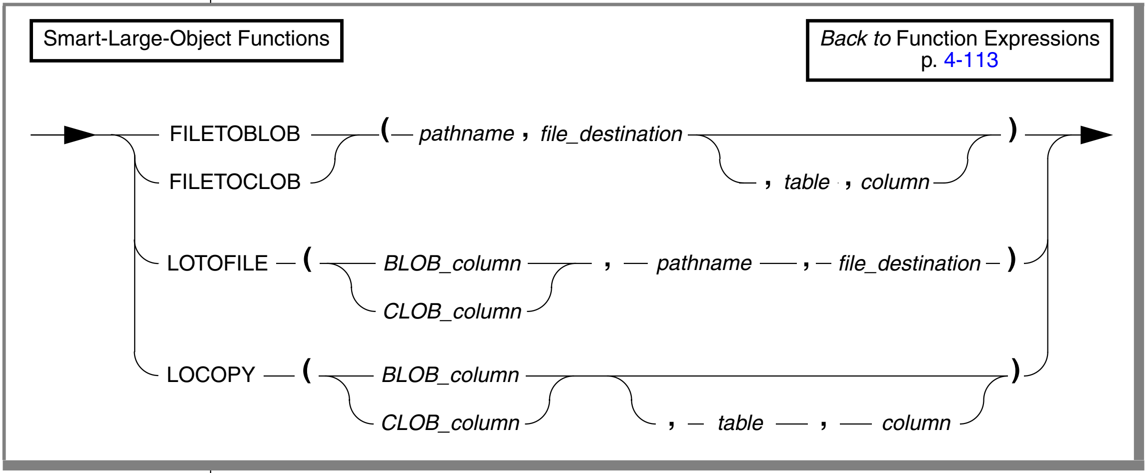
To execute the **IFX_REPLACE_MODULE** function in an IBM Informix ESQL/C application, you must associate the function with a cursor. ♦

For more information on how to use **IFX_REPLACE_MODULE** to replace a shared-object file, see the chapter on how to design a UDR in *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to use the **IFX_UNLOAD_MODULE** procedure, see "IFX_UNLOAD_MODULE Procedure" on page 2-416.

IDS

Smart-Large-Object Functions

The Smart-large-object functions support CLOB and BLOB data types:



Element	Purpose	Restrictions	Syntax
BLOB_column, *CLOB_column*	A column of type BLOB; a column of type CLOB	In *table***.***column*, the *column* must have BLOB or CLOB data type.	Identifier, p. 4-189
column	Column within *table* for the copy of the BLOB or CLOB value	Must have CLOB or BLOB as its data type.	Quoted String, p. 4-243
file_destination	Name of the system on which to put or get the smart large object	The only valid values are the strings `server` or `client`.	Quoted String, p. 4-243
pathname	Directory path and filename to locate the smart large object	Must exist on *file_destination* system. See also "Pathnames with Commas" on page 4-137.	Quoted String, p. 4-243
table	Name or synonym of a table that contains *column* whose storage characteristics are used for the copy of BLOB or CLOB value	Must exist in the database and must contain a CLOB or BLOB column.	Quoted String, p. 4-243

FILETOBLOB and FILETOCLOB Functions

The **FILETOBLOB** function creates a BLOB value for data that is stored in a specified operating-system file. Similarly, the **FILETOCLOB** function creates a CLOB value for data that is stored in an operating-system file.

These functions determine the operating-system file to use from the following parameters:

- The *pathname* parameter identifies the directory path and name of the source file.
- The *file destination* parameter identifies the computer, client or server, on which this file resides:
  - ❑ Set *file destination* to `client` to identify the client computer as the location of the source file. The *pathname* can be either a full pathname or relative to the current directory.
  - ❑ Set *file destination* to `server` to identify the server computer as the location of the source file. The *pathname* must be a full pathname.

The *table* and *column* parameters are optional:

- If you omit *table* and *column*, the **FILETOBLOB** function creates a BLOB value with the system-specified storage defaults, and the **FILETOCLOB** function creates a CLOB value with the system-specified storage defaults.

  These functions obtain the system-specific storage characteristics from either the ONCONFIG file or the sbspace. For more information on system-specified storage defaults, see the *Administrator's Guide*.
- If you specify *table* and *column*, the **FILETOBLOB** and **FILETOCLOB** functions use the storage characteristics from the specified column for the BLOB or CLOB value that they create.

The **FILETOBLOB** function returns a handle value (a pointer) to the new BLOB value. Similarly, **FILETOCLOB** returns a handle value to the new CLOB value. Neither function actually copies the smart-large-object value into a database column. You must assign the BLOB or CLOB value to the appropriate column.

GLS

The **FILETOCLOB** function performs any code-set conversion that might be required when it copies the file from the client or server computer to the database. ♦

The following INSERT statement uses the **FILETOCLOB** function to create a CLOB value from the value in the **smith.rsm** file:

```
INSERT INTO candidate (cand_num, cand_lname, resume)
   VALUES (2, 'Smith', FILETOCLOB('smith.rsm', 'client'))
```

In the preceding example, the **FILETOCLOB** function reads the **smith.rsm** file in the current directory on the client computer and returns a handle value to a CLOB value that contains the data in this file. Because the **FILETOCLOB** function does not specify a table and column name, this new CLOB value has the system-specified storage characteristics. The INSERT statement then assigns this CLOB value to the **resume** column in the **candidate** table.

The following INSERT statement uses the **FILETOBLOB** function to create a BLOB value in a remote table, **election2000**, from the value in the **photos.xxx** file on the local database server:

```
INSERT INTO rdb@rserv:election2000 (cand_pic)
   VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',
      'candidate', 'cand_photo'))
```

In the preceding example, the **FILETOBLOB** function reads the **photos.xxx** file in the specified directory on the local database server and returns a handle value to a BLOB value that contains the data in this file. The INSERT statement then assigns this BLOB value to the **cand_pic** column in the remote **election2000** table. This new BLOB value has the storage characteristics of the **cand_photo** column in the **candidate** table on the local database server.

In the following example, the new BLOB value has the storage characteristics of the **cand_pix** column in the **election96** table on a remote database server:

```
INSERT INTO rdb@rserv:election2000 (cand_pic)
   VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',
      'rdb2@rserv2:election96', 'cand_pix'))
```

When you qualify the **FILETOBLOB** or **FILETOCLOB** function with the name of a remote database and a remote database server, the *pathname* and the *file destination* become relative to the remote database server.

When you specify `server` as the file destination, as the following example shows, the **FILETOBLOB** function looks for the source file (in this case, **photos.xxx**) on the remote database server:

```
INSERT INTO rdb@rserv:election (cand_pic)
   VALUES (rdb@rserv:FILETOBLOB('C:\tmp\photos.xxx', 'server'))
```

When you specify `client` as the file destination, however, as in the following example, the **FILETOBLOB** function looks for the source file (in this case, **photos.xxx**) on the local client computer:

```
INSERT INTO rdb@rserv:election (cand_pic)
   VALUES (rdb@rserv:FILETOBLOB('photos.xxx', 'client'))
```

Pathnames with Commas

If a comma (**,**) symbol is within the *pathname* of the function, the database server expects the pathname to have the following format:

```
"offset, length, pathname"
```

For pathnames that contain a comma, you must also specify an offset and length, as in the following example:

```
FILETOBLOB("0,-1,/tmp/blob,x","server")
```

The first term in the quoted *pathname* string is an *offset* of `0`, which instructs the database server to begin reading at the start of the file.

The second term is a *length* of `-1`, which instructs the database server to continue reading until the end of the entire file.

The third term is the */tmp/blob,x pathname*, specifying which file to read. (Notice the comma symbol that precedes the *x*.)

Because the *pathname* includes a comma, the comma-separated *offset* and *length* specifications are necessary in this example to avoid an error when **FILETOBLOB** is called. You do not need to specify *offset* and *length* for pathnames that include no comma, but including `0,-1,` as the initial characters of the pathname string avoids this error for any valid pathname.

LOTOFILE Function

The **LOTOFILE** function copies a smart large object to an operating-system file. The first parameter specifies the BLOB or CLOB column to copy. The function determines what file to create from the following parameters:

- The *pathname* identifies the directory path and the source file name.
- The *file destination* identifies the computer, client or server, on which this file resides:
  - ❑ Set *file destination* to `client` to identify the client computer as the location of the source file. The *pathname* can be either a full pathname or a path relative to the current directory.
  - ❑ Set *file destination* to `server` to identify the server computer as the location of the source file. The full pathname is required.

By default, the **LOTOFILE** function generates a filename of the form:

```
file.hex_id
```

In this format, *file* is the filename you specify in *pathname* and *hex_id* is the unique hexadecimal smart-large-object identifier. The maximum number of digits for a smart-large-object identifier is 17; however most smart large objects would have an identifier with significantly fewer digits.

UNIX

For example, suppose you specify a *pathname* value as follows:

```
'/tmp/resume'
```

If the CLOB column has the identifier **203b2**, then **LOTOFILE** creates the file:

```
/tmp/resume.203b2
```

♦

Windows

For example, suppose you specify a *pathname* value as follows:

```
'C:\tmp\resume'
```

If the CLOB column has an identifier of **203b2**, the **LOTOFILE** function would create the file:

```
C:\tmp\resume.203b2
```

♦

To change the default filename, you can specify the following wildcards in the filename of the *pathname*:

- One or more contiguous question mark (?) characters in the filename can generate a unique filename.

  The **LOTOFILE** function replaces each question mark with a hexadecimal digit from the identifier of the BLOB or CLOB column.

  UNIX

  For example, suppose you specify a *pathname* value as follows:

  ```
  '/tmp/resume??.txt'
  ```

  The **LOTOFILE** function puts 2 digits of the hexadecimal identifier into the name. If the CLOB column has an identifier of **203b2**, the **LOTOFILE** function would create the file:

  ```
  /tmp/resume20.txt
  ```

  ♦

  If you specify more than 17 question marks, **LOTOFILE** ignores them.

- An exclamation (!) point at the end of the filename indicates that the filename does not need to be unique.

  Windows

  For example, suppose you specify a pathname value as follows:

  ```
  'C:\tmp\resume.txt!'
  ```

  The **LOTOFILE** function does not use the smart-large-object identifier in the filename, so it generates the following file:

  ```
  C:\tmp\resume.txt
  ```

  ♦

If the filename you specify already exists, **LOTOFILE** returns an error.

GLS

The **LOTOFILE** function performs any code-set conversion that might be required when it copies a CLOB value from the database to a file on the client or server computer. ♦

When you qualify **LOTOFILE** with the name of a remote database and a remote database server, the BLOB or CLOB column, the *pathname*, and the *file destination* become relative to the remote database server.

When you specify `server` as the file destination, as in the next example, the **LOTOFILE** function copies the smart large object from the remote database server to a source file in the specified directory on the remote database server:

```
rdb@rserv:LOTOFILE(blob_col, 'C:\tmp\photo.gif!', 'server')
```

If you specify `client` as the file destination, as in the following example, the **LOTOFILE** function copies the smart large object from the remote database server to a source file in the specified directory on the local client computer:

```
rdb@rserv:LOTOFILE(clob_col, 'C:\tmp\essay.txt!', 'client')
```

LOCOPY Function

The **LOCOPY** function creates a copy of a smart large object. The first parameter specifies the BLOB or CLOB column to copy. The *table* and *column* parameters are optional.

- If you omit *table* and *column*, the **LOCOPY** function creates a smart large object with system-specified storage defaults and copies the data in the BLOB or CLOB column into it.

  The **LOCOPY** function obtains the system-specific storage defaults from either the ONCONFIG file or the sbspace. For more information on system-specified storage defaults, see the *Administrator's Guide*.
- When you specify *table* and *column*, the **LOCOPY** function uses the storage characteristics from the specified *column* for the BLOB or CLOB value that it creates.

The **LOCOPY** function returns a handle value (a pointer) to the new BLOB or CLOB value. This function does *not* actually store the new smart-large-object value into a column in the database. You must assign the BLOB or CLOB value to the appropriate column.

The following ESQL/C code fragment copies the CLOB value in the **resume** column of the **candidate** table to the **resume** column of the **interview** table:

```
/* Insert a new row in the interviews table and get the
 * resulting SERIAL value (from sqlca.sqlerrd[1])
 */
EXEC SQL insert into interviews (intrv_num, intrv_time)
   values (0, '09:30');
intrv_num = sqlca.sqlerrd[1];

/* Update this interviews row with the candidate number
 * and resume from the candidate table. Use LOCOPY to
 * create a copy of the CLOB value in the resume column
 * of the candidate table.
 */
EXEC SQL update interviews
   SET (cand_num, resume) =
      (SELECT cand_num,
         LOCOPY(resume, 'candidate', 'resume')
      FROM candidate
      WHERE cand_lname = 'Haven')
   WHERE intrv_num = :intrv_num;
```

In the preceding example, the **LOCOPY** function returns a handle value for the copy of the CLOB **resume** column in the **candidate** table. Because the **LOCOPY** function specifies a table and column name, this new CLOB value has the storage characteristics of this **resume** column. If you omit the table (**candidate**) and column (**resume**) names, the **LOCOPY** function uses the system-defined storage defaults for the new CLOB value. The UPDATE statement then assigns this new CLOB value to the **resume** column in the **interviews** table.

In the following example, the **LOCOPY** function executes on the local database server and returns a handle value on the local server for the copy of the BLOB **cand_pic** column in the remote **election2000** table. The INSERT statement then assigns this new BLOB value to the **cand_photo** column in the local **candidate** table.

```
INSERT INTO candidate (cand_photo)
   SELECT LOCOPY(cand_pic) FROM rdb@rserv:election2000
```

When the **LOCOPY** function executes on the same database server as the original BLOB or CLOB column in a distributed query, it produces two copies of the BLOB or CLOB value, one on the remote database server and the other on the local database server, as the following two examples show.

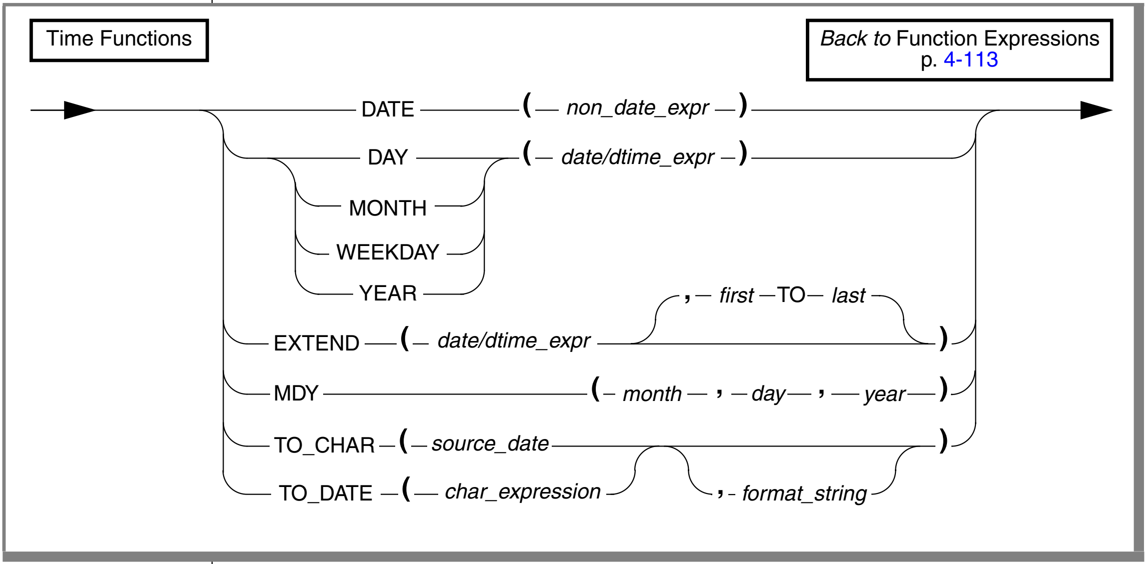
In the first example, the **LOCOPY** function executes on the remote database server and returns a handle value on the remote server for the copy of the BLOB **cand_pic** column in the remote **election2000** table. The INSERT statement then assigns this new BLOB value to the **cand_photo** column in the local **candidate** table:

```
INSERT INTO candidate (cand_photo)
   SELECT rdb@rserv:LOCOPY(cand_pic)
      FROM rdb@rserv:election2000
```

In the second example, the **LOCOPY** function executes on the local database server and returns a handle value on the local server for the copy of the BLOB **cand_photo** column in the local **candidate** table. The INSERT statement then assigns this new BLOB value to the **cand_pic** column in the remote **election2000** table:

```
INSERT INTO rdb@rserv:election2000 (cand_pic)
   SELECT LOCOPY(cand_photo) FROM candidate
```

Time Functions



Element	Purpose	Restrictions	Syntax
char_expression	Expression to be converted to a DATE or DATETIME value	Must be a literal, host variable, expression, or column of a character data type	Expression, p. 4-67
date/dtime_expr	Expression that returns a DATE or DATETIME value	Must return a DATE or DATETIME value.	Expression, p. 4-67
day	Expression that returns the number of a day of the month	Must evaluate to an integer not greater than the number of days in the specified month.	Expression, p. 4-67
first	Largest time unit in the result If you omit *first* and *last*, the default *first* is YEAR.	Must be a DATETIME qualifier that specifies a time unit no smaller than *last*.	DATETIME Qualifier, p. 4-65
format_string	String that contains a format mask for the DATE or DATETIME value	Must be a character data type and contain a valid date format. Can be a column, host variable, expression, or constant.	Quoted String, p. 4-243
last	Smallest time unit in the result	Must be a DATETIME qualifier that specifies a time unit no smaller than *first*.	DATETIME Qualifier, p. 4-65
month	Expression that represents the number of the month	Must evaluate to an integer between 1 and 12, inclusive.	Expression, p. 4-67

(1 of 2)

Element	Purpose	Restrictions	Syntax
non_date_expr	Expression that represents a value to be converted to a DATE data type	Typically an expression that returns a CHAR, DATETIME, or INTEGER value that can be converted to a DATE data type.	Expression, p. 4-67
source_date	Date to be converted to a character string	Type DATETIME or DATE. Can be host variable, expression, column, or constant.	Expression, p. 4-67
year	Expression that represents the year	Must evaluate to a four-digit integer. You cannot use a two-digit abbreviation.	Expression, p. 4-67

(2 of 2)

DATE Function

The **DATE** function converts a non-DATE expression to a DATE value. The argument can be any expression that can be converted to a DATE value, usually a CHAR, DATETIME, or INTEGER value. The following WHERE clause specifies a CHAR value for the nondate expression:

```
WHERE order_date < DATE('12/31/97')
```

When the **DATE** function interprets a CHAR nondate expression, it expects this expression to conform to any DATE format that the **DBDATE** environment specifies. For example, suppose **DBDATE** is set to `Y2MD/` when you execute the following query:

```
SELECT DISTINCT DATE('02/01/1998') FROM ship_info
```

This SELECT statement generates an error because the **DATE** function cannot convert this nondate expression. The **DATE** function interprets the first part of the date string (`02`) as the year and the second part (`01`) as the month.

For the third part (`1998`), the **DATE** function encounters four digits when it expects a two-digit day (valid day values must be between `01` and `31`). It therefore cannot convert the value. For the SELECT statement to execute successfully with the `Y2MD/` value for **DBDATE**, the nondate expression would need to be '98/02/01'. For information on the format of **DBDATE**, see the *IBM Informix Guide to SQL: Reference*.

When you specify a positive INTEGER value for the nondate expression, the **DATE** function interprets this as the number of days after December 31, 1899.

If the integer value is negative, the **DATE** function interprets the value as the number of days before December 31, 1899. The following WHERE clause specifies an INTEGER value for the nondate expression:

```
WHERE order_date  <  DATE(365)
```

The database server searches for rows with an **order_date** value less than December 31, 1900 (12/31/1899 plus 365 days).

DAY Function

The **DAY** function returns an integer that represents the day of the month. The following example uses the **DAY** function with the **CURRENT** function to compare column values to the current day of the month:

```
WHERE DAY(order_date) > DAY(CURRENT)
```

MONTH Function

The **MONTH** function returns an integer corresponding to the month portion of its type DATE or DATETIME argument. The following example returns a number from `1` through `12` to indicate the month when the order was placed:

```
SELECT order_num, MONTH(order_date) FROM orders
```

WEEKDAY Function

The **WEEKDAY** function returns an integer that represents the day of the week; zero (0) represents Sunday, one (1) represents Monday, and so on. The following example lists all the orders that were paid on the same day of the week, which is the current day:

```
SELECT * FROM orders
   WHERE WEEKDAY(paid_date) = WEEKDAY(CURRENT)
```

YEAR Function

The **YEAR** function returns a four-digit integer that represents the year.

The following example lists orders in which the **ship_date** is earlier than the beginning of the current year:

```
SELECT order_num, customer_num FROM orders
   WHERE year(ship_date) < YEAR(TODAY)
```

Similarly, because a DATE value is a simple calendar date, you cannot add or subtract a DATE value with an INTERVAL value whose *last* qualifier is smaller than DAY. In this case, convert the DATE value to a DATETIME value.

EXTEND Function

The **EXTEND** function adjusts the precision of a DATETIME or DATE value. The expression cannot be a quoted string representation of a DATE value.

If you do not specify *first* and *last* qualifiers, the default qualifiers are YEAR TO FRACTION(3).

If the expression contains fields that are not specified by the qualifiers, the unwanted fields are discarded.

If the *first* qualifier specifies a larger (that is, more significant) field than what exists in the expression, the new fields are filled in with values returned by the **CURRENT** function. If the *last* qualifier specifies a smaller field (that is, less significant) than what exists in the expression, the new fields are filled in with constant values. A missing MONTH or DAY field is filled in with `1`, and the missing HOUR to FRACTION fields are filled in with `0`.

In the following example, the first EXTEND call evaluates to the **call_dtime** column value of YEAR TO SECOND. The second statement expands a literal DATETIME so that an interval can be subtracted from it. You must use the **EXTEND** function with a DATETIME value if you want to add it to or subtract it from an INTERVAL value that does not have all the same qualifiers. The third example updates only a portion of the datetime value, the hour position. The **EXTEND** function yields just the *hh:mm* part of the datetime. Subtracting 11:00 from the hours and minutes of the datetime yields an INTERVAL value of the difference, plus or minus, and subtracting that from the original value forces the value to 11:00.

```
EXTEND (call_dtime, YEAR TO SECOND)

EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE (3) TO MINUTE

UPDATE cust_calls SET call_dtime = call_dtime -
(EXTEND(call_dtime, HOUR TO MINUTE) - DATETIME (11:00)
HOUR TO MINUTE) WHERE customer_num = 106
```

MDY Function

The **MDY** function returns a type DATE value with three expressions that evaluate to integers that represent the month, day, and year. The first expression must evaluate to an integer that represents the number of the month (1 to 12).

The second expression must evaluate to an integer that represents the number of the day of the month (1 to 28, 29, 30, or 31, as appropriate for the month.)

The third expression must evaluate to a four-digit integer that represents the year. You cannot use a two-digit abbreviation for the third expression. The following example sets the **paid_date** associated with the order number 8052 equal to the first day of the present month:

```
UPDATE orders SET paid_date = MDY(MONTH(TODAY), 1, YEAR(TODAY))
   WHERE po_num = '8052'
```

TO_CHAR Function

The **TO_CHAR** function converts a DATE or DATETIME value to a character string. The character string contains the date that was specified in the *source_date* parameter and represents this date in the format that was specified in the *format_string* parameter.

IDS

You can use this function only with built-in data types. ♦

If the value of the *source_date* parameter is NULL, the function returns a NULL value.

If you omit the *format_string* parameter, the **TO_CHAR** function uses the default date format to format the character string. The default date format is specified by environment variables such as **GL_DATETIME** and **GL_DATE**.

The *format_string* parameter does not have to imply the same qualifiers as the *source_date* parameter. When the implied formatting mask qualifier in *format_ string* is different from the qualifier in *source_date*, the **TO_CHAR** function extends the DATETIME value as if it had called the **EXTEND** function.

In the following example, the user wants to convert the **begin_date** column of the **tab1** table to a character string. The **begin_date** column is defined as a DATETIME YEAR TO SECOND data type. The user uses a SELECT statement with the **TO_CHAR** function to perform this conversion.

```
SELECT TO_CHAR(begin_date, '%A %B %d, %Y %R') FROM tab1
```

The symbols in the *format_string* parameter in this example have the following meanings. For a complete list of format symbols and their meanings, see the **GL_DATE** and **GL_DATETIME** environment variables in the *IBM Informix GLS User's Guide*.

Symbol	Meaning
%A	Full weekday name as defined in the locale
%B	Full month name as defined in the locale
%d	Day of the month as a decimal number
%Y	Year as a 4-digit decimal number
%R	Time in 24-hour notation

The result of applying the specified *format_string* to the **begin_date** column is as follows:

```
Wednesday July 23, 1997 18:45
```

TO_DATE Function

The **TO_DATE** function converts a character string to a DATETIME value. The function evaluates the *char_expression* parameter as a date according to the date format you specify in the *format_string* parameter and returns the equivalent date. If *char_expression* is NULL, then a NULL value is returned.

IDS

The argument of the **TO_DATE** function must be of a built-in data type. ♦

If you omit the *format_string* parameter, the **TO_DATE** function applies the default DATETIME format to the DATETIME value. The default DATETIME format is specified by the **GL_DATETIME** environment variable.

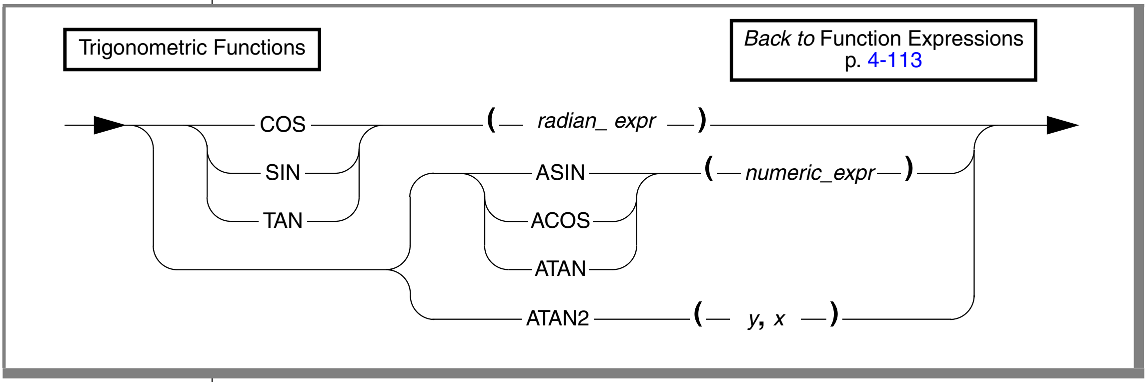
In the following example, the user wants to convert a character string to a DATETIME value in order to update the **begin_date** column of the **tab1** table with the converted value. The **begin_date** column is defined as a DATETIME YEAR TO SECOND data type. The user uses an UPDATE statement that contains a **TO_DATE** function to accomplish this result.

```
UPDATE tab1
   SET begin_date = TO_DATE('Wednesday July 23, 1997 18:45',
   '%A %B %d, %Y %R');
```

The *format_string* parameter in this example tells the **TO_DATE** function how to format the converted character string in the **begin_date** column. For a table that shows the meaning of each format symbol in this format string, see "TO_CHAR Function" on page 4-147.

Trigonometric Functions

The built-in trigonometric functions have the following syntax.



Element	Purpose	Restrictions	Syntax
numeric_expr	Expression that serves as an argument to the ASIN, ACOS, or ATAN functions	Must return a value between -1 and 1, inclusive.	Expression, p. 4-67
radian_expr	Expression that represents the number of radians	Must return a numeric value.	Expression, p. 4-67
x	Expression that represents the x coordinate of the rectangular coordinate pair (*x*, *y*)	Must return a numeric value.	Expression, p. 4-67
y	Expression that represents the y coordinate of the rectangular coordinate pair (*x*, *y*)	Must return a numeric value.	Expression, p. 4-67

Formulas for Radian Expressions

The **COS**, **SIN**, and **TAN** functions take the number of radians (*radian_expr*) as an argument. If you are using degrees and want to convert degrees to radians, use the following formula:

```
# degrees * p/180= # radians
```

To convert radians to degrees, use the following formula:

```
# radians * 180/p = # degrees
```

COS Function

The **COS** function returns the cosine of a radian expression. The following example returns the cosine of the values of the degrees column in the **anglestbl** table. The expression passed to the **COS** function in this example converts degrees to radians.

```
SELECT COS(degrees*180/3.1416) FROM anglestbl
```

SIN Function

The **SIN** function returns the sine of a radian expression. This example returns the sine of the values in the **radians** column of the **anglestbl** table:

```
SELECT SIN(radians) FROM anglestbl
```

TAN Function

The **TAN** function returns the tangent of a radian expression. This example returns the tangent of the values in the **radians** column of the **anglestbl** table:

```
SELECT TAN(radians) FROM anglestbl
```

ACOS Function

The **ACOS** function returns the arc cosine of a numeric expression. The following example returns the arc cosine of the value (-0.73) in radians:

```
SELECT ACOS(-0.73) FROM anglestbl
```

ASIN Function

The **ASIN** function returns the arc sine of a numeric expression. The following example returns the arc sine of the value (-0.73) in radians:

```
SELECT ASIN(-0.73) FROM anglestbl
```

ATAN Function

The **ATAN** function returns the arc tangent of a numeric expression. The following example returns the arc tangent of the value (-0.73) in radians:

```
SELECT ATAN(-0.73) FROM anglestbl
```

ATAN2 Function

The **ATAN2** function computes the angular component of the polar coordinates (*r*, q) associated with (*x, y*). The following example compares *angles* to q for the rectangular coordinates (4, 5):

```
WHERE angles > ATAN2(4,5)     --determines q for (4,5) and
                                compares to angles
```

You can determine the length of the radial coordinate *r* using the expression that the following example shows:

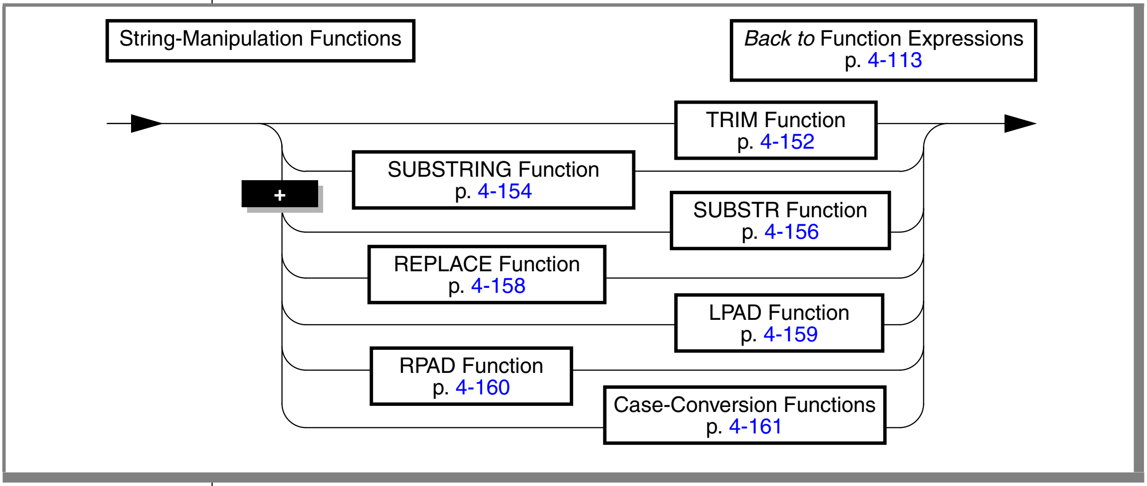
```
SQRT(POW(x,2) + POW(y,2))     --determines r for (x,y)
```

You can determine the length of the radial coordinate *r* for the rectangular coordinates (4,5) using the expression that the following example shows:

```
SQRT(POW(4,2) + POW(5,2))     --determines r for (4,5)
```

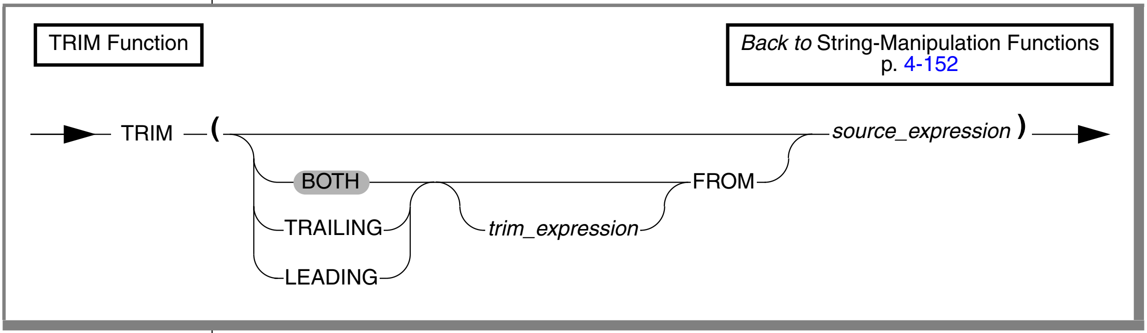
String-Manipulation Functions

String-manipulation functions perform various operations on strings of characters. The syntax for string-manipulation functions is as follows.


String-Manipulation Functions
Back to Function Expressions p. 4-113
TRIM Function p. 4-152
SUBSTRING Function p. 4-154
+
SUBSTR Function p. 4-156
REPLACE Function p. 4-158
LPAD Function p. 4-159
RPAD Function p. 4-160
Case-Conversion Functions p. 4-161

TRIM Function

The **TRIM** function removes leading or trailing pad characters from a string.



Element	Purpose	Restrictions	Syntax
trim _expression	Expression that evaluates to a single character or NULL Default is a blank space (**=** ASCII 32)	Must be a character expression	Quoted String, p. 4-243
source _expression	Character expression, including a character column name, or a call to another TRIM function	Cannot be LVARCHAR nor a host variable	Quoted String, p. 4-243

The **TRIM** function returns a VARCHAR value identical to its character string argument, except that any leading or trailing whitespace characters, if specified, are deleted. If no trim qualifier (LEADING, TRAILING, or BOTH) is specified, BOTH is the default. If no *trim_expression* is used, a single blank space is assumed. If either the *trim_expression* or the *source_expression* evaluates to NULL, the result of the **TRIM** function is NULL. The maximum length of the returned string must be 255 bytes or fewer, because the VARCHAR data type supports no more than 255 bytes.

The following example shows some generic uses for the **TRIM** function:

```
SELECT TRIM (c1) FROM tab;
SELECT TRIM (TRAILING '#' FROM c1) FROM tab;
SELECT TRIM (LEADING FROM c1) FROM tab;
UPDATE c1='xyz' FROM tab WHERE LENGTH(TRIM(c1))=5;
SELECT c1, TRIM(LEADING '#' FROM TRIM(TRAILING '%' FROM
   '###abc%%%')) FROM tab;
```

GLS

When you use the DESCRIBE statement with a SELECT statement that uses the **TRIM** function in the select list, the described character type of the trimmed column depends on the database server that you are using and on the data type of the *source_expression*. For further information on the GLS aspects of the **TRIM** function in ESQL/C, see the *IBM Informix GLS User's Guide*. ♦

Fixed Character Columns

The **TRIM** function can be specified on fixed-length character columns. If the length of the string is not completely filled, the unused characters are padded with blank space. Figure 4-3 shows this concept for the column entry '##A2T##', where the column is defined as CHAR(10).

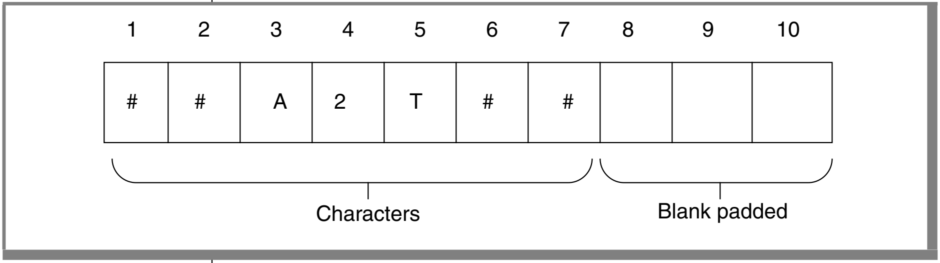


Figure 4-3
Column Entry in a Fixed-Length Character Column

If you want to trim the pound sign (#) *trim_expression* from the column, you need to consider the blank padded spaces as well as the actual characters.

For example, if you specify the trim specification BOTH, the result from the trim operation is A2T##, because the **TRIM** function does not match the blank padded space that follows the string. In this case, the only pound signs (#) trimmed are those that precede the other characters. The SELECT statement is shown, followed by Figure 4-4, which presents the result.

```
SELECT TRIM(LEADING '#' FROM col1) FROM taba
```

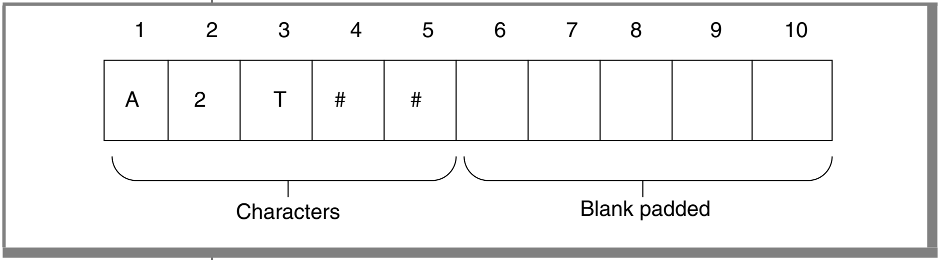


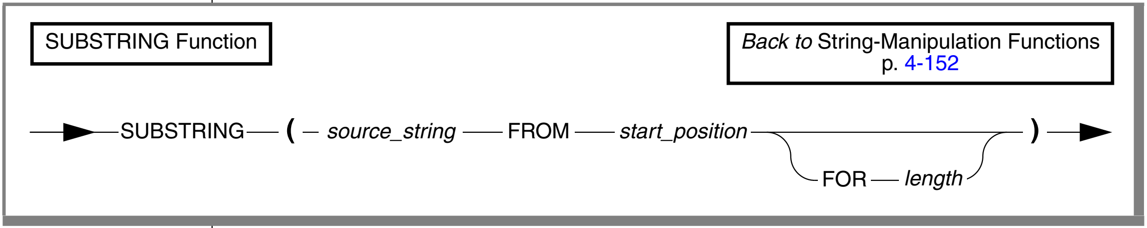
Figure 4-4
Result of TRIM Operation

This SELECT statement removes all occurrences of the pounds (#) sign:

```
SELECT TRIM(BOTH '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba
```

SUBSTRING Function

The **SUBSTRING** function returns a subset of a character string.



Element	**Purpose**	**Restrictions**	**Syntax**
length	Number of characters to return from *source_string*	Must be an expression, constant, column, or host variable that returns an integer.	Literal Number, p. 4-216
source_string	String argument to the **SUBSTRING** function	Must be an expression, constant, column, or host variable whose value can be converted to a character data type.	Expression, p. 4-67
start_position	Position in *source_string* of first returned character	Must be an expression, constant, column, or host variable that returns an integer.	Literal Number, p. 4-216

IDS

You can use the **SUBSTRING** function only with built-in data types. ♦

The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position.*

Value of *Start_Position*	**How the Database Server Determines the Starting Position of the Return Subset**
Positive	Counts forward from the first character in *source_string* For example, if *start_position* = `1`, the first character in the *source_string* is the first character in the return subset.
Zero (`0`)	Counts from one position before (that is, to the left of) the first character in *source_string* For example, if *start_position* = `0` and *length* = `1`, the database server returns NULL, whereas if *length* = `2`, the database server returns the first character in *source_string*.
Negative	Counts backward from one position before (that is, left of) the first character in *source_string* For example, if *start_position* = `-1`, the starting position of the return subset is two positions (`0` and `-1`) before the first character in *source_string*.

GLS

In locales for languages with a right-to-left writing direction, such as Arabic, Farsi, or Hebrew, *right* should replace *left* in the preceding table. ♦

The size of the subset is specified by *length*. The *length* parameter refers to the number of logical characters rather than to the number of bytes. If you omit the *length* parameter, the **SUBSTRING** function returns the entire portion of *source_ string* that begins at *start_position.* The following example specifies that the subset of the source string that begins in column position `3` and is two characters long should be returned:

```
SELECT SUBSTRING('ABCDEFG' FROM 3 FOR 2) FROM mytable
```

The following table shows the output of this SELECT statement.

(constant)
CD

In the following example, the user specifies a negative *start_position* for the return subset:

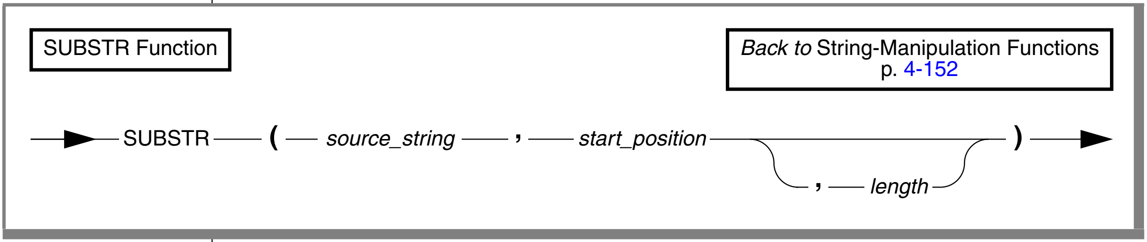
```
SELECT SUBSTRING('ABCDEFG' FROM -3 FOR 7)
   FROM mytable
```

The database server starts at the `-3` position (four positions before the first character) and counts forward for `7` characters. The following table shows the output of this SELECT statement.

(constant)
ABC

SUBSTR Function

The **SUBSTR** function has the same purpose as the **SUBSTRING** function (to return a subset of a source string), but it uses different syntax.



Element	**Purpose**	**Restrictions**	**Syntax**
length	Number of characters to be returned from source_string	Must be an expression, constant, column, or host variable that returns an integer.	Literal Number, p. 4-216
source_string	String that serves as input to the **SUBSTR** function	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67
start_position	Column position in *source_string* where the **SUBSTR** function starts to return characters	Must be an integer expression, constant, column, or host variable. Can have a plus sign (+), a minus sign (-), or no sign.	Literal Number, p. 4-216

IDS

Any argument to the **SUBSTR** function must be of a built-in data type. ♦

The **SUBSTR** function returns a subset of *source_string*. The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position.*

Value of *Start_Position*	How the Database Server Determines the Starting Position of the Return Subset
Positive	Counts forward from the first character in *source_string*
Zero (`0`)	Counts forward from the first character in *source_string* (that is, treats a *start_position* of `0` as equivalent to `1`)
Negative	Counts backward from the last character in *source_string* A value of `-1` returns the last character in *source_string*.

The *length* parameter specifies the number of logical characters (not bytes) in the subset. If you omit the *length* parameter, the **SUBSTR** function returns the entire portion of *source_ string* that begins at *start_position.*

In the following example, the user specifies that the subset of the source string to be returned begins at a starting position 3 characters back from the end of the string. Because the source string is 7 characters long, the starting position is the fifth column of *source_string*. Because the user does not specify a value for *length*, the database server returns the entire portion of the source string that begins in column position 5.

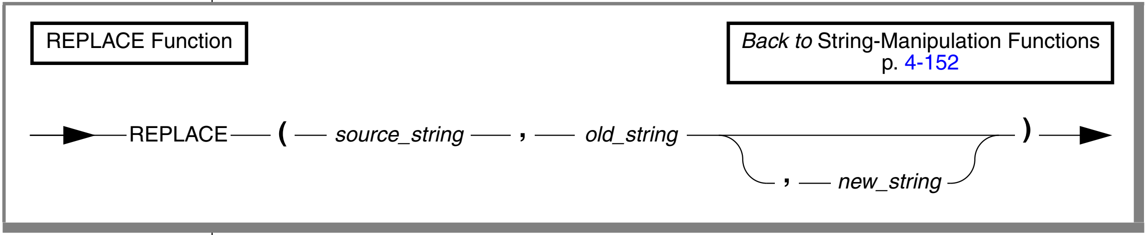
```
SELECT SUBSTR('ABCDEFG', -3)
   FROM mytable
```

The following table shows the output of this SELECT statement.

(constant)
EFG

REPLACE Function

The **REPLACE** function replaces specified characters within a source string with different characters.



Element	Purpose	Restrictions	Syntax
new_string	Character or characters that replace *old_string* in the return string	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67
old_string	Character or characters in *source_string* that are to be replaced by *new_string*	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67
source_string	String of characters argument to the **REPLACE** function	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67

IDS

Any argument to the **REPLACE** function must be of a built-in data type. ♦

The **REPLACE** function returns a copy of *source_string* in which every occurrence of *old_string* is replaced by *new_string*. If you omit the *new_string* option, every occurrence of *old_string* is omitted from the return string.

In the following example, the user replaces every occurrence of `xz` in the source string with `t`:

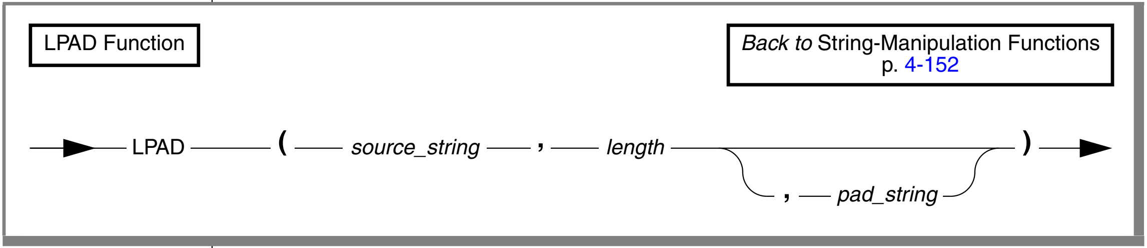
```
SELECT REPLACE('Mighxzy xzime', 'xz', 't')
   FROM mytable
```

The following table shows the output of this SELECT statement.

(constant)
Mighty time

LPAD Function

The **LPAD** function returns a copy of *source_string* that is left-padded to the total number of characters specified by *length*.



Element	Purpose	Restrictions	Syntax
length	Integer value that specifies total number of characters in the returned string	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Literal Number, p. 4-216
pad_string	String that specifies the pad character or characters	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67
source_string	String that serves as input to the LPAD function	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type.	Expression, p. 4-67

IDS

Any argument to the **LPAD** function must be of a built-in data type. ♦

The *pad_string* parameter specifies the character or characters to be used for padding the source string. The sequence of pad characters occurs as many times as necessary to make the return string the length specified by *length*.

The series of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you specify no *pad_string*, the default value is a single blank.

In the following example, the user specifies that the source string is to be left-padded to a total length of 16 characters. The user also specifies that the pad characters are a series consisting of a hyphen and an underscore (-_).

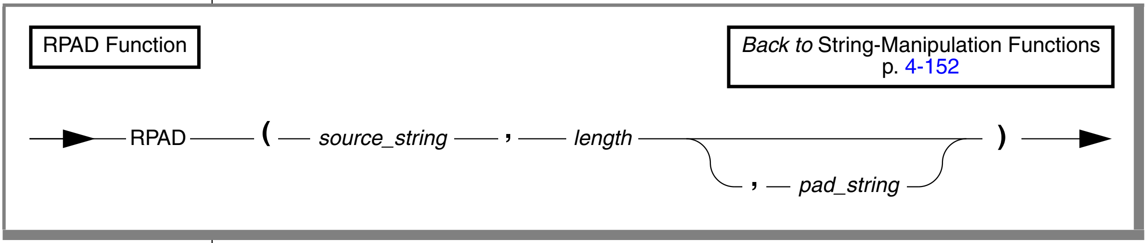
```
SELECT LPAD('Here we are', 16, '-_') FROM mytable
```

The following table shows the output of this SELECT statement.

```
(constant)
-_-_-Here we are
```

RPAD Function

The **RPAD** function returns a copy of *source_string* that is right-padded to the total number of characters that *length* specifies.



Element	Purpose	Restrictions	Syntax
length	Integer value that indicates the total number of characters in the return string	Must be an expression, constant, column, or host variable	Literal Number, p. 4-216
pad_string	String that specifies the pad character or characters	Must be an expression, column, constant, or host variable of a data type that can be converted to a character data type	Expression, p. 4-67
source_string	String that serves as input to the RPAD function	Same as for *pad_stringe*	Expression, p. 4-67

IDS

Any argument to the **RPAD** function must be of a built-in data type. ♦

The *pad_string* parameter specifies the pad character or characters to be used to pad the source string.

The series of pad characters occurs as many times as necessary to make the return string reach the length that *length* specifies. The series of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you omit the *pad_string* parameter, the default value is a single blank space.

In the following example, the user specifies that the source string is to be right-padded to a total length of 18 characters. The user also specifies that the pad characters to be used are a sequence consisting of a question mark and an exclamation point (?!).

```
SELECT RPAD('Where are you', 18, '?!')
   FROM mytable
```

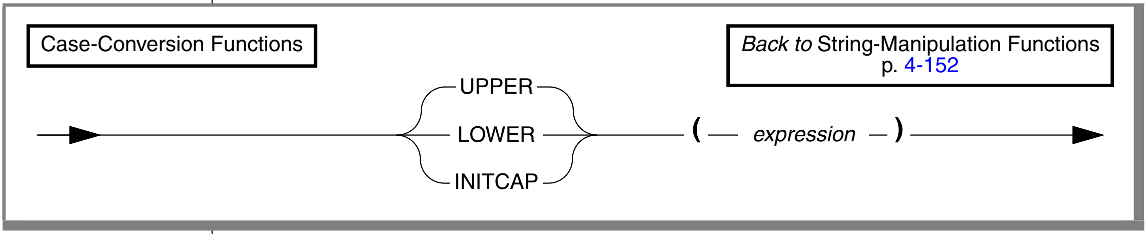
The following table shows the output of this SELECT statement.

(constant)
`Where are you?!?!?`

Case-Conversion Functions

The case-conversion functions perform lettercase conversion on alphabetic characters. In the default locale, only the ASCII characters `A - Z` and `a - z` can be modified by these functions, which enable you to perform case-insensitive searches in your queries and to specify the format of the output.

The case-conversion functions are **UPPER**, **LOWER**, and **INITCAP**. The following diagram shows the syntax of these case-conversion functions.


Case-Conversion Functions
Back to String-Manipulation Functions p. 4-152
UPPER
LOWER
INITCAP
(
expression
)

Element	Purpose	Restrictions	Syntax
expression	Expression returning a character string	Must be a character type. If a host variable, its declared length must be long enough to store the converted string.	Expression, p. 4-67

The *expression* must return a character data type. When the column is described, the data type returned by the database server is that of *expression*. For example, if the input type is CHAR, the output type is also CHAR.

IDS

Arguments to these functions must be of the built-in data types. ♦

GLS

The byte length returned from the description of a column with a case-conversion function is the input byte length of the source string. If you use a case-conversion function with a multibyt *expression* argument, the conversion might increase or decrease the length of the string. If the byte length of the result string exceeds the byte length *expression*, the database server truncates the result string to fit into the byte length of *expression*.

Only characters designated as ALPHA class in the locale file are converted, and this occurs only if the locale recognizes the construct of lettercase. ♦

If *expression* is NULL, the result of a case-conversion function is also NULL.

The database server treats a case-conversion function as an SPL routine in the following instances:

- If it has no argument
- If it has one argument, and that argument is a named argument
- If it has more than one argument
- If it appears in a SELECT list with a host variable as an argument

If none of the conditions in the preceding list are met, the database server treats a case-conversion function as a system function.

The following example uses all the case-conversion functions in the same query to specify multiple output formats for the same value:

```
Input value:

SAN Jose

Query:

SELECT City, LOWER(City), Lower("City"),
   UPPER (City), INITCAP(City)
      FROM Weather;

Query output:

SAN Jose   san jose   city   SAN JOSE   San Jose
```

UPPER Function

The **UPPER** function returns a copy of the *expression* argument in which every lowercase alphabetical character in the *expression* is replaced by a corresponding uppercase alphabetic character.

The following example shows how to use the **UPPER** function to perform a case-insensitive search on the **lname** column for all employees with the last name of **curran**:

```
SELECT title, INITCAP(fname), INITCAP(lname) FROM employees
WHERE UPPER (lname) = "CURRAN"
```

Because the **INITCAP** function is specified in the select list, the database server returns the results in a mixed-case format. For example, the output of one matching row might read: `accountant James Curran`.

LOWER Function

The **LOWER** function returns a copy of the *expression* in which every uppercase alphabetic character in the *expression* is replaced by a corresponding lowercase alphabetic character.

The following example shows how to use the **LOWER** function to perform a case-insensitive search on the **City** column. This statement directs the database server to replace all instances (that is, any variation) of the words `san jose`, with the mixed-case format, `San Jose`.

```
UPDATE Weather SET City = "San Jose"
WHERE LOWER (City) = "san jose";
```

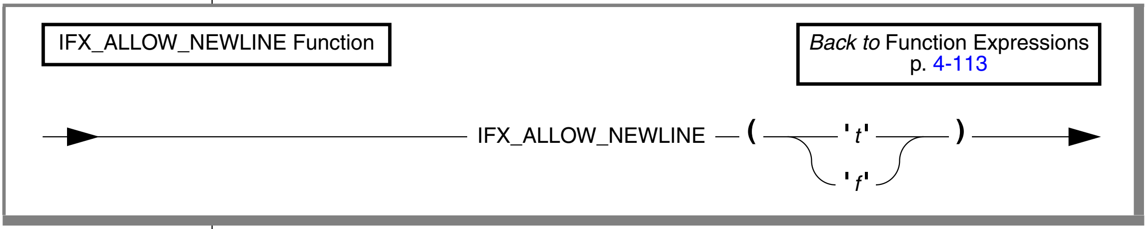
INITCAP Function

The **INITCAP** function returns a copy of the *expression* in which every word in the *expression* begins with an uppercase letter. With this function, a *word* begins after any character other than a letter. Thus, in addition to a blank space, symbols such as commas, periods, colons, and so on, introduce a new word.

For an example of the **INITCAP** function, see "UPPER Function" on page 4-162.

IFX_ALLOW_NEWLINE Function

The **IFX_ALLOW_NEWLINE** function sets a newline mode that allows newline characters in quoted strings or disallows newline characters in quoted strings within the current session.


IFX_ALLOW_NEWLINE Function
Back to Function Expressions p. 4-113
IFX_ALLOW_NEWLINE
(
'*t*'
'*f*'
)

If you enter `'t'` as the argument of this function, you enable newline characters in quoted strings in the session. If you enter `'f'` as the argument, you disallow newline characters in quoted strings in the session.

You can set the newline mode for all sessions by setting the ALLOW_NEWLINE parameter in the **ONCONFIG** file to a value of `0` (newline characters not allowed) or to a value of `1` (newline characters allowed). If you do not set this configuration parameter, the default value is `0`. Each time you start a session, the new session inherits the newline mode set in the **ONCONFIG** file. To change the newline mode for the session, execute the **IFX_ALLOW_NEWLINE** function. Once you have set the newline mode for a session, the mode remains in effect until the end of the session or until you execute the **IFX_ALLOW_NEWLINE** function again within the session.

In the following example, assume that you did not specify any value for the ALLOW_NEWLINE parameter in the **ONCONFIG** file, so by default newline characters are not allowed in quoted strings in any session. After you start a new session, you can enable newline characters in quoted strings in that session by executing the **IFX_ALLOW_NEWLINE** function:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('t')
```

E/C

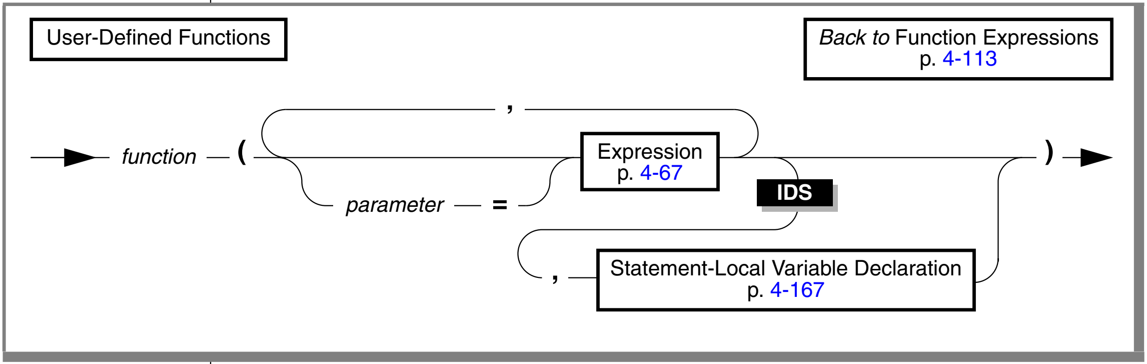
The newline mode that is set by the ALLOW_NEWLINE parameter in the **ONCONFIG** file or by the execution of the **IFX_ALLOW_NEWLINE** function in a session applies only to quoted-string literals in SQL statements. The newline mode does not apply to quoted strings contained in host variables in SQL statements. Host variables can contain newline characters within string data regardless of the newline mode currently in effect.

For example, you can use a host variable to insert data that contains newline characters into a column even if the ALLOW_NEWLINE parameter in the **ONCONFIG** file is set to 0. ♦

For further information on how the **IFX_ALLOW_NEWLINE** function affects quoted strings, see "Quoted String" on page 4-243. For further information on the ALLOW_NEWLINE parameter in the **ONCONFIG** file, see the *Administrator's Reference*.

User-Defined Functions

A user-defined function is a function that you write in SPL or in a language external to the database, such as C or Java.


User-Defined Functions
Back to Function Expressions p. 4-113
function
(
,
parameter
=
Expression p. 4-67
IDS
,
Statement-Local Variable Declaration p. 4-167
)

Element	Purpose	Restrictions	Syntax
function	Name of the function	Function must exist	Database Object Name, p. 4-46
parameter	Name of an argument that was declared in a CREATE FUNCTION statement	If you use the *parameter* = option for any argument in the called function, you must use it for all arguments	Identifier, p. 4-189

You can call user-defined functions within SQL statements. Unlike built-in functions, user-defined functions can only be used by the creator of the function, the DBA, and the users who have been granted the Execute privilege on the function. For more information, see "GRANT" on page 2-459.

The following examples show some user-defined function expressions. The first example omits the *parameter* option when it lists the function argument:

```
read_address('Miller')
```

This second example uses the *parameter* option to specify the argument value:

```
read_address(lastname = 'Miller')
```

When you use the *parameter* option, the *parameter* name must match the name of the corresponding parameter in the function registration. For example, the preceding example assumes that the **read_address( )** function had been registered as follows:

```
CREATE FUNCTION read_address(lastname CHAR(20))
RETURNING address_t ...
```

IDS

A statement-local variable (SLV) enables you to transmit a value from a user-defined function call to another part of the SQL statement.

To use SLVs with a call to a user-defined function

1. Write one or more OUT parameters for the user-defined function.

   For more information on how to write a user-defined function with OUT parameters, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

2. When you register the user-defined function, specify the OUT keyword in front of each OUT parameter.

   For more information, see "Specifying OUT Parameters for a User-Defined Routine" on page 4-269.

3. Declare each SLV in a function expression that calls the user-defined function with each OUT parameter.

   The call to the user-defined function must be made within a WHERE clause. For information about the syntax to declare a SLV, see "Statement-Local Variable Declaration" on page 4-167.

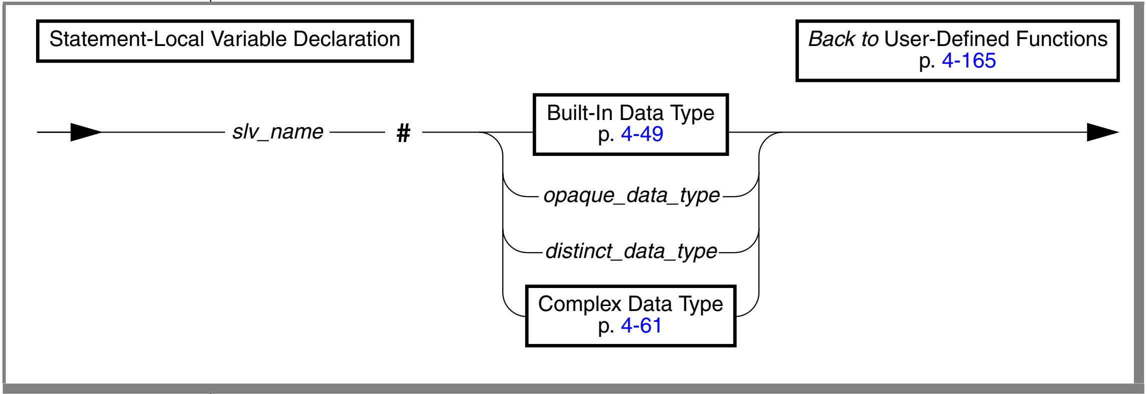
4. Use the SLVs that the user-defined function has initialized within the SQL statement.

   Once the call to the user-defined function has initialized the SLVs, you can use their values in other parts of the SQL statement. For information about the use of SLVs within an SQL statement, see "Statement-Local Variable Expressions" on page 4-169.

IDS

Statement-Local Variable Declaration

The Statement-Local Variable Declaration declares a statement-local variable (SLV) in a call to a user-defined function that defines one or more OUT parameters.


Statement-Local Variable Declaration
Back to User-Defined Functions p. 4-165
slv_name
#
Built-In Data Type p. 4-49
opaque_data_type
distinct_data_type
Complex Data Type p. 4-61

Element	Purpose	Restrictions	Syntax
distinct_data_type	Name of a distinct data type	The distinct data type must already exist in the database	Identifier, p. 4-189
opaque_data_type	Name of an opaque data type	The opaque data type must already exist in the database	Identifier, p. 4-189
slv_name	Name of a statement local variable you are defining	The *slv_name* exists only for the life of the statement. The *slv_name* must be unique within the statement	Identifier, p. 4-189

You declare an SLV in a user-defined function call so that a user-defined function can assign the value of its OUT parameter to the SLV. The user-defined function *must* be invoked in the WHERE clause of the SQL statement. For example, if you register a function with the following CREATE FUNCTION statement, you can use its **y** parameter as an SLV in a WHERE clause:

```
CREATE FUNCTION find_location(a FLOAT, b FLOAT, OUT y INTEGER)
RETURNING VARCHAR(20)
EXTERNAL NAME "/usr/lib/local/find.so"
LANGUAGE C
```

In this example, **find_location( )** accepts two FLOAT values that represent a latitude and a longitude and returns the name of the nearest city with an extra value of type INTEGER that represents the population rank of the city.

You can now call **find_location( )** in a WHERE clause:

```
SELECT zip_code_t FROM address
   WHERE address.city = find_location(32.1, 35.7, rank # INT)
   AND rank < 101;
```

The function expression passes two FLOAT values to **find_location(** and declares an SLV named **rank** of type INT. In this case, **find_location( )** will return the name of the city nearest latitude 32.1 and longitude 35.7 (which might be a heavily populated area) whose population rank is between 1 and 100. The statement then returns the zip code that corresponds to that city.

The WHERE clause of the SQL statement *must* produce an SLV that is used within other parts of the statement. The following SELECT statement is *invalid* because the select list of the Projection clause produces the SLV:

```
-- illegal SELECT statement
SELECT title, contains(body, 'dog and cat', rank # INT), rank
   FROM documents
```

The data type you use when you declare a SLV in a statement must be the same as the data type of the corresponding OUT parameter in the CREATE FUNCTION statement. If you use different but compatible data types, such as INTEGER and FLOAT, the database server automatically performs the cast between the data types.

SLVs share the name space with UDR variables and the column names of the table involved in the SQL statement. Therefore, the database uses the following precedence to resolve ambiguous situations:

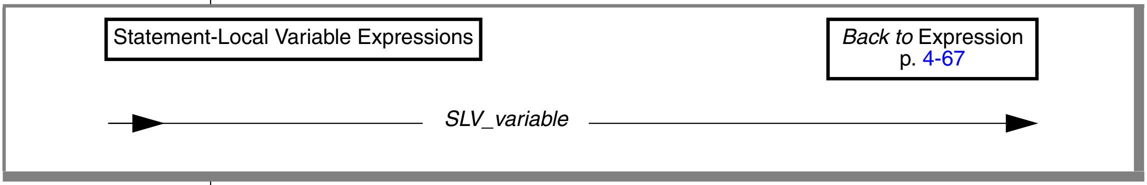
- UDR variables
- Column names
- SLVs

Once the user-defined function assigns its OUT parameters to the SLVs, you can use this SLV value in other parts of the SQL statement. For more information, see "Statement-Local Variable Expressions" on page 4-169.

IDS

Statement-Local Variable Expressions

The Statement-Local Variable Expression specifies a statement-local variable (SLV) that you can use elsewhere in the same SQL statement.



Element	**Purpose**	**Restrictions**	**Syntax**
SLV_variable	Statement-local variable (SLV) assigned in a call to a user-defined function in the same SQL statement	The *SLV_variable* exists only for the life of the statement. Its name must be unique within the statement.	Identifier, p. 4-189

You define an SLV in the call to a user-defined function in the WHERE clause of the SQL statement. This user-defined function must be defined with one or more OUT parameters. The call to the user-defined function assigns the value of the OUT parameters to the SLVs. For more information, see "Statement-Local Variable Declaration" on page 4-167.

Once the user-defined function assigns its OUT parameters to the SLVs, you can use these values in other parts of the SQL statement, subject to the following scop-of-reference rules:

- The SLV is *read-only* throughout the query (or subquery) in which it is defined.
- The scope of an SLV extends from the query in which the SLV is defined down into all nested subqueries.

  In other words, if a query contains a subquery, an SLV that is visible in the query is also visible to all subqueries of that query.
- In nested queries, the scope of an SLV does *not* extend upwards.

  In other words, if a query contains a subquery and the SLV is defined in the subquery, it is *not* visible to the parent query.

- In queries that involve UNION, the SLV is only visible in the query in which it is defined.

  The SLV is not visible to all other queries involved in the UNION.

- For INSERT, DELETE, and UPDATE statements, an SLV is not visible outside the SELECT portion of the statement.

  Within this SELECT portion, all the above scoping rules apply.

Important: *A statement-local variable is valid only for the life of a single SQL statement.*

The following SELECT statement calls the **find_location( )** function in a WHERE clause and defines the **rank** SLV. Here **find_location( )** accepts two values that represent a latitude and a longitude and return the name of the nearest city with an extra value of type INTEGER that represents the population rank of the city.

```
SELECT zip_code_t FROM address
   WHERE address.city = find_location(32.1, 35.7, rank # INT)
   AND rank < 101;
```

When execution of the **find_location()** function completes successfully, the function has initialized the **rank** SLV. The SELECT then uses this **rank** value in a second WHERE clause condition. In this example, the Statement-Local Variable Expression is the variable **rank** in the second WHERE clause condition:

```
rank < 101
```

The number of OUT parameters and SLVs that a user-defined function can have is not restricted. (Releases of Dynamic Server earlier than Version 9.4 restricted user-defined functions to a single OUT parameter, and thereby restricted the number of SLVs to no more than one.)

If the user-defined function that initializes the SLVs is *not* executed in an iteration of the statement, the SLVs each have a value of NULL. SLV values do *not* persist across iterations of the statement. At the start of each iteration, the database server sets the SLV values to NULL.

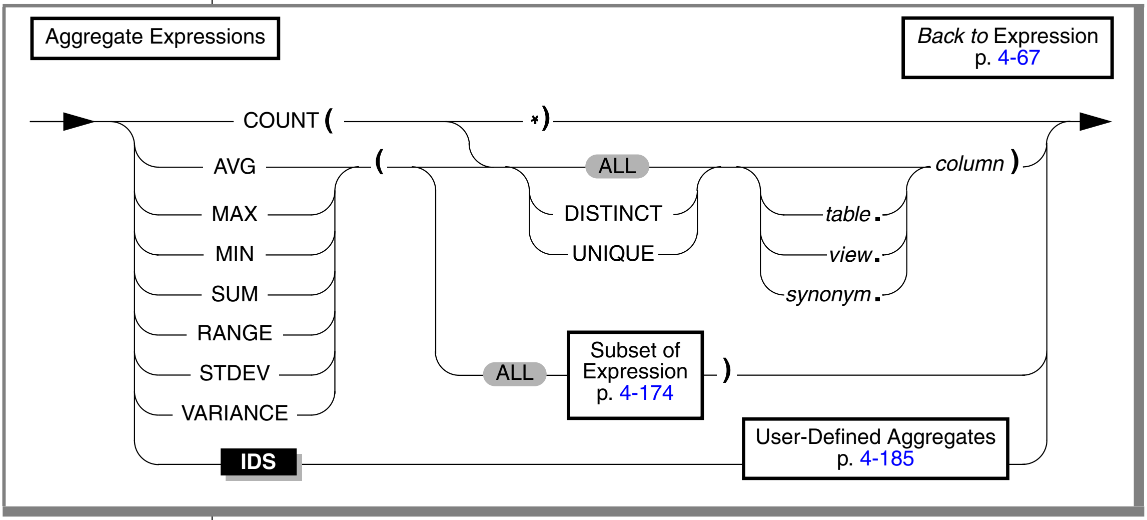
The following partial statement calls two user-defined functions with OUT parameters, whose values are referenced with the SLV names **out1** and **out2**:

```
SELECT...
   WHERE func_2(x, out1 # INTEGER) < 100
   AND (out1 = 12 OR out1 = 13)
   AND func_3(a, out2 # FLOAT) = "SAN FRANCISCO"
   AND out2 = 3.1416;
```

For more information on how to write a user-defined function with OUT parameters, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data. The built-in aggregate functions have the following syntax.



Element	Purpose	Restrictions	Syntax
column	Column to which aggregate function is applied	See headings for individual keywords on pages that follow	Identifier, p. 4-189
synonym, table, view	Synonym, table, or view that contains *column*	*Synonym* and the *table* or *view* to which it points must exist	Database Object Name, p. 4-46

You cannot use an aggregate expression in a condition that is part of a WHERE clause unless you use the aggregate expression within a subquery. You cannot apply an aggregate function to a BYTE or TEXT column. For other general restrictions, see "Subset of Expressions Valid in an Aggregate Expression" on page 4-174.

An aggregate function returns one value for a set of queried rows. The following examples show aggregate functions in SELECT statements:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013

SELECT COUNT(*) FROM orders WHERE order_num = 1001

SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer
```

If you use an aggregate function and one or more columns in the select list of the Projection clause, you must put all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause.

Types of Aggregate Expressions

SQL statements can include *built-in* aggregates and *user-defined* aggregates. The built-in aggregates include all the aggregates shown in the syntax diagram in "Aggregate Expressions" on page 4-171 except for the "User-Defined Aggregates" category. User-defined aggregates are any new aggregates that the user creates with the CREATE AGGREGATE statement.

Built-in Aggregates

Built-in aggregates are aggregate functions that are defined by the database server, such as AVG, SUM, and COUNT. These aggregates work only with built-in data types, such as INTEGER and FLOAT. You can extend these built-in aggregates to work with extended data types. To extend built-in aggregates, you must create UDRs that overload several binary operators.

After you overload the binary operators for a built-in aggregate, you can use that aggregate with an extended data type in an SQL statement. For example, if you have overloaded the **plus** operator for the SUM aggregate to work with a specified row type and assigned this row type to the **complex** column of the **complex_tab** table, you can apply the SUM aggregate to the **complex** column:

```
SELECT SUM(complex) FROM complex_tab
```

For more information on how to extend built-in aggregates, see *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to invoke built-in aggregates, see the descriptions of individual built-in aggregates in the following pages.

User-Defined Aggregates

A user-defined aggregate is an aggregate that you define to perform an aggregate computation that the database server does not provide. For example, you can create a user-defined aggregate named SUMSQ that returns the sum of the squared values of a specified column. User-defined aggregates can work with built-in data types or extended data types or both, depending on how you define the support functions for the user-defined aggregate.

To create a user-defined aggregate, use the CREATE AGGREGATE statement. In this statement you name the new aggregate and specify the support functions for the aggregate. Once you create the new aggregate and its support functions, you can use the aggregate in SQL statements. For example, if you created the SUMSQ aggregate and specified that it works with the FLOAT data type, you can apply the SUMSQ aggregate to a FLOAT column named **digits** in the **test** table:

```
SELECT SUMSQ(digits) FROM test
```

For more information on how to create user-defined aggregates, see "CREATE AGGREGATE" on page 2-104 and the discussion of user-defined aggregates in *IBM Informix User-Defined Routines and Data Types Developer's Guide*. For information on how to invoke user-defined aggregates, see "User-Defined Aggregates" on page 4-185.

Subset of Expressions Valid in an Aggregate Expression

As indicated in the diagrams for "Aggregate Expressions" on page 4-171 and "User-Defined Aggregates" on page 4-185, not all expressions are available when you use an aggregate expression. The argument of an aggregate function, for example, cannot itself contain an aggregate function. You cannot use aggregate functions in the following contexts:

- In a WHERE clause, unless it is contained in a subquery, or unless the aggregate is on a correlated column from a parent query and the WHERE clause is in a subquery within a HAVING clause
- As an argument to an aggregate function

  The following nested aggregate expression is invalid:

  ```
  MAX (AVG (order_num))
  ```
- On a BYTE or TEXT column

You cannot use a column that is a collection data type as an argument to the following aggregate functions:

- **AVG**
- **SUM**
- **MIN**
- **MAX**

Expression or *column* arguments to built-in aggregates (except for **COUNT**, **MAX**, **MIN**, and **PERCENT**) must return numeric or INTERVAL data types, but **RANGE** also accepts DATE and DATETIME arguments.

For **SUM** and **AVG**, you cannot use the difference between two DATE values directly as the argument to an aggregate, but you can use DATE differences as operands within arithmetic expression arguments. For example,

```
SELECT . . .  AVG(ship_date - order_date)
```

returns error -1201, but the following equivalent expression is valid:

```
SELECT . . .  AVG((ship_date - order_date)*1)
```

Including or Excluding Duplicates in the Row Set

The DISTINCT keyword restricts the argument to unique values from the specified column. The UNIQUE and DISTINCT keywords are synonyms.

The ALL keyword specifies that all values selected from the column or expression, including any duplicate values, are used in the calculation.

AVG Function

The **AVG** function returns the average of all values in the specified column or expression. You can apply the **AVG** function only to number columns. If you use the DISTINCT keyword, the average (meaning the *mean*) is calculated from only the distinct values in the specified column or expression. The query in the following example finds the average price of a helmet:

```
SELECT AVG(unit_price) FROM stock WHERE stock_num = 110
```

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **AVG** function returns a NULL for that column.

Overview of COUNT Functions

The **COUNT** function is actually a set of functions that enable you to count column values in different ways, according to arguments after the COUNT keyword. Each form of the **COUNT** function is explained in the following subsections. For a comparison of the different forms of the **COUNT** function, see “Comparison of the Different COUNT Functions” on page 4-177.

COUNT(*) Function

The **COUNT (*)** function returns the number of rows that satisfy the WHERE clause of a SELECT statement. The following example finds how many rows in the **stock** table have the value HRO in the **manu_code** column:

```
SELECT COUNT(*) FROM stock WHERE manu_code = 'HRO'
```

If the SELECT statement does not have a WHERE clause, the **COUNT (*)** function returns the total number of rows in the table. The following example finds how many rows are in the **stock** table:

```
SELECT COUNT(*) FROM stock
```

If the SELECT statement contains a GROUP BY clause, the **COUNT (*)** function reflects the number of values in each group. The following example is grouped by the first name; the rows are selected if the database server finds more than one occurrence of the same name:

```
SELECT fname, COUNT(*) FROM customer GROUP BY fname
   HAVING COUNT(*) > 1
```

If the value of one or more rows is NULL, the **COUNT (*)** function includes the NULL columns in the count unless the WHERE clause explicitly omits them.

COUNT DISTINCT and COUNT UNIQUE Functions

The **COUNT DISTINCT** function returns the number of unique values in the column or expression, as the following example shows. If the **COUNT DISTINCT** function encounters NULLs, it ignores them.

```
SELECT COUNT (DISTINCT item_num) FROM items
```

NULLs are ignored unless every value in the specified column is NULL. If every column value is NULL, the **COUNT DISTINCT** function returns zero (0).

The UNIQUE keyword has the same meaning as the DISTINCT keyword in **COUNT** functions. The UNIQUE keyword instructs the database server to return the number of unique non-NULL values in the column or expression. The following example calls the **COUNT UNIQUE** function, but it is equivalent to the preceding example that calls the **COUNT DISTINCT** function:

```
SELECT COUNT (UNIQUE item_num) FROM items
```

COUNT column Function

The **COUNT** *column* function returns the total number of non-NULL values in the column or expression, as the following example shows:

```
SELECT COUNT (item_num) FROM items
```

The ALL keyword can precede the specified column name for clarity, but the query result is the same whether you include the ALL keyword or omit it.

The following example shows how to include the ALL keyword in the **COUNT** *column* function:

```
SELECT COUNT (ALL item_num) FROM items
```

Comparison of the Different COUNT Functions

You can use the different forms of the **COUNT** function to retrieve different types of information about a table. The following table summarizes the meaning of each form of the **COUNT** function.

COUNT Function	Description
COUNT (*)	Returns the number of rows that satisfy the query If you do not specify a WHERE clause, this function returns the total number of rows in the table.
COUNT DISTINCT or COUNT UNIQUE	Returns the number of unique non-NULL values in the specified column
COUNT (*column*) or COUNT (ALL *column*)	Returns the total number of non-NULL values in the specified column

Some examples can help to show the differences among the different forms of the **COUNT** function. Most of the following examples query against the **ship_instruct** column of the **orders** table in the demonstration database. For information on the structure of the **orders** table and the data in the **ship_instruct** column, see the description of the demonstration database in the *IBM Informix Guide to SQL: Reference*.

Examples of the Count(*) Function

In the following example, the user wants to know the total number of rows in the **orders** table. So the user calls the **COUNT(*)** function in a SELECT statement without a WHERE clause.

```
SELECT COUNT(*) AS total_rows FROM orders
```

The following table shows the result of this query.

total_rows
23

In the following example, the user wants to know how many rows in the **orders** table have a NULL value in the **ship_instruct** column. The user calls the **COUNT(*)** function in a SELECT statement with a WHERE clause, and specifies the IS NULL condition in the WHERE clause.

```
SELECT COUNT (*) AS no_ship_instruct FROM orders
   WHERE ship_instruct IS NULL
```

The following table shows the result of this query.

no_ship_instruct
2

In the following example, the user wants to know how many rows in the **orders** table have the value `express` in the **ship_instruct** column. So the user calls the **COUNT(*)** function in the select list and specifies the equals (=) relational operator in the WHERE clause.

```
SELECT COUNT (*) AS ship_express FROM ORDERS
   WHERE ship_instruct = 'express'
```

The following table shows the result of this query.

ship_express
6

Examples of the COUNT DISTINCT Function

In the next example, the user wants to know how many unique non-NULL values are in the **ship_instruct** column of the **orders** table. The user calls the **COUNT DISTINCT** function in the select list of the SELECT statement.

```
SELECT COUNT(DISTINCT ship_instruct) AS unique_notnulls
   FROM orders
```

The following table shows the result of this query.

unique_notnulls
16

Examples of the COUNT column Function

In the following example the user wants to know how many non-NULL values are in the **ship_instruct** column of the **orders** table. The user invokes the **COUNT(*column*)** function in the select list of the SELECT statement.

```
SELECT COUNT(ship_instruct) AS total_notnullsFROM orders
```

The following table shows the result of this query.

total_notnulls
21

A similar query for non-NULL values in the **ship_instruct** column can include the ALL keyword in the parentheses that follow the COUNT keyword.

```
SELECT COUNT(ALL ship_instruct) AS all_notnulls FROM orders
```

The following table shows that the query result is the same whether you include or omit the ALL keyword (because ALL is the default).

all_notnulls
21

MAX Function

The **MAX** function returns the largest value in the specified column or expression. Using the DISTINCT keyword does not change the results. The query in the following example finds the most expensive item that is in stock but has not been ordered:

```
SELECT MAX(unit_price) FROM stock
   WHERE NOT EXISTS (SELECT * FROM items
      WHERE stock.stock_num = items.stock_num AND
      stock.manu_code = items.manu_code)
```

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **MAX** function returns a NULL for that column.

MIN Function

The **MIN** function returns the lowest value in the column or expression. Using the DISTINCT keyword does not change the results. The following example finds the least expensive item in the **stock** table:

```
SELECT MIN(unit_price) FROM stock
```

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **MIN** function returns a NULL for that column.

SUM Function

The **SUM** function returns the sum of all the values in the specified column or expression, as the following example shows. If you use the DISTINCT keyword, the sum is for only distinct values in the column or expression.

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013
```

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **SUM** function returns a NULL for that column. You cannot use the **SUM** function with a non-numeric column.

RANGE Function

The **RANGE** function computes the range of returned values. It calculates the difference between the maximum and the minimum values, as follows:

```
range(expr) = max(expr) - min(expr)
```

You can apply the **RANGE** function only to numeric columns. The following query finds the range of ages for a population:

```
SELECT RANGE(age) FROM u_pop
```

As with other aggregates, the **RANGE** function applies to the rows of a group when the query includes a GROUP BY clause, as the next example shows:

```
SELECT RANGE(age) FROM u_pop GROUP BY birth
```

Because DATE values are stored internally as integers, you can use the **RANGE** function on DATE columns. With a DATE column, the return value is the number of days between the earliest and latest dates in the column.

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **RANGE** function returns a NULL for that column.

Important: *All computations for the **RANGE** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

STDEV Function

The **STDEV** function computes the standard deviation of a data set, which is the square root of the **VARIANCE** function. You can apply the **STDEV** function only to numeric columns. The next query finds the standard deviation:

```
SELECT STDEV(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the **STDEV** function applies to the rows of a group when the query includes a GROUP BY clause, as this example shows:

```
SELECT STDEV(age) FROM u_pop GROUP BY birth WHERE STDEV(age) > 0
```

NULL values are ignored unless every value in the specified column is NULL. If every column value is NULL, **STDEV** returns a NULL for that column.

Important: *All computations for the **STDEV** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

You cannot use this function on columns of type DATE.

Within a SELECT Statement with GROUP BY clause, **STDEV** returns a zero variance for a count of 1. You can omit this special case through appropriate query construction (for example, "having count(*) > 1"). Otherwise, a data set that has only a few cases might block the rest of the query result.

VARIANCE Function

The **VARIANCE** function returns an estimate of the population variance, as the standard deviation squared. **VARIANCE** calculates the following value:

$$(\mathrm{SUM}(X_i^2) - (\mathrm{SUM}(X_i)^2)/N)/N$$

In this formula, X_i is each value in the column and N is the total number of non-NULL values in the column (unless all values are NULL, in which case the variance is logically undefined, and the **VARIANCE** function returns NULL).

You can apply the **VARIANCE** function only to numeric columns.

The following query estimates the variance of **age** values for a population:

```
SELECT VARIANCE(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the **VARIANCE** function applies to the rows of a group when the query includes a GROUP BY clause, as in this example:

```
SELECT VARIANCE(age) FROM u_pop GROUP BY birth
   WHERE VARIANCE(age) > 0
```

As previously noted, **VARIANCE** ignores NULL values unless every qualified row is NULL for a specified column. If every value is NULL, then **VARIANCE** returns a NULL result for that column. (This typically indicates missing data, and is not necessarily a good estimate of underlying population variance.)

If N, the total number of qualified non-NULL column values, equals one, then the **VARIANCE** function returns zero (another implausible estimate of the true population variance). To omit this special case, you can modify the query. For example, you might include a HAVING COUNT(*) > 1 clause.

Important: *All calculations for the **VARIANCE** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

Although DATE values are stored internally as an integer, you cannot use the **VARIANCE** function on columns of data type DATE.

E/C

Error Checking in ESQL/C

Aggregate functions always return one row. If no rows are selected, the function returns a NULL. You can use the **COUNT (*)** function to determine whether any rows were selected, and you can use an indicator variable to determine whether any selected rows were empty. Fetching a row with a cursor that is associated with an aggregate function always returns one row; hence, `100` for end of data is never returned into the **sqlcode** variable for a first FETCH attempt.

You can also use the GET DIAGNOSTICS statement for error checking.

Summary of Aggregate Function Behavior

An example can help to summarize the behavior of the aggregate functions. Assume that the **testtable** table has a single INTEGER column that is named **num**. The contents of this table are as follows.

num
2
2
2
3
3
4
(NULL)

You can use aggregate functions to obtain information about the **num** column and the **testtable** table. The following query uses the **AVG** function to obtain the average of all the non-NULL values in the **num** column:

```
SELECT AVG(num) AS average_number FROM testtable
```

The following table shows the result of this query.

average_number
2.66666666666667

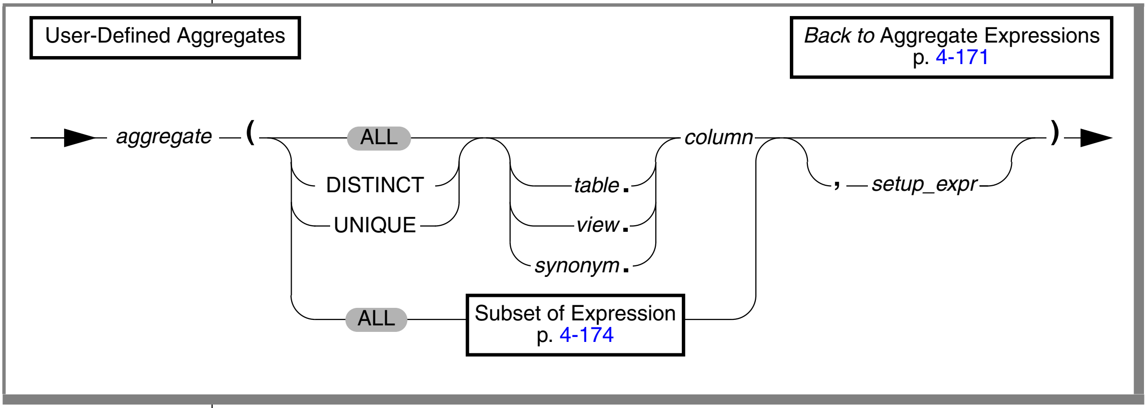
You can use the other aggregate functions in SELECT statements that are similar to the preceding example. If you enter a series of SELECT statements that have different aggregate functions in the select list and do not include a WHERE clause, you receive the results that the following table shows.

Function	Results	Function	Results
COUNT (*)	7	MAX	4
COUNT (DISTINCT)	3	MAX(DISTINCT)	4
COUNT (ALL *num*)	6	MIN	2
COUNT (*num*)	6	MIN(DISTINCT)	2
AVG	2.66666666666667	RANGE	2
AVG (DISTINCT)	3.00000000000000	SUM	16
STDEV	0.74535599249993	SUM(DISTINCT)	9
VARIANCE	0.55555555555556		

IDS

User-Defined Aggregates

You can create your own aggregate expressions with the CREATE AGGREGATE statement and then invoke these aggregates wherever you can invoke the built-in aggregates. The following diagram shows the syntax for invoking a user-defined aggregate.



Element	Purpose	Restrictions	Syntax
aggregate	Name of the user-defined aggregate to invoke	The *aggregate* and the support functions defined for *aggregate* must exist	Identifier, p. 4-189
column	Name of a column within *table*	Must exist and have a numeric data type	Quoted String, p. 4-243
setup_expr	Set-up expression that customizes *aggregate* for a specific invocation	Cannot be a lone host variable. Any columns referenced in *setup_expr* must be in the GROUP BY clause of the query	Expression, p. 4-67
synonym, table, view	Synonym, table, or view in which *column* occurs	The *synonym* and the *table* or *view* to which it points must exist	Database Object Name, p. 4-46

Use the DISTINCT or UNIQUE keywords to specify that the user-defined aggregate is to be applied only to unique values in the named column or expression. Use the ALL keyword to specify that the aggregate is to be applied to all values in the named column or expression.

If you omit the DISTINCT, UNIQUE, and ALL keywords, ALL is the default. For further information on the DISTINCT, UNIQUE, and ALL keywords, see "Including or Excluding Duplicates in the Row Set" on page 4-175.

When you specify a set-up expression, this value is passed to the **INIT** support function that was defined for the user-defined aggregate in the CREATE AGGREGATE statement.

In the following example, you apply the user-defined aggregate named **my_avg** to all values of the **quantity** column in the **items** table:

```
SELECT my_avg(quantity) FROM items
```

In the following example, you apply the user-defined aggregate named **my_sum** to unique values of the **quantity** column in the **items** table. You also supply the value `5` as a set-up expression. This value might specify that the initial value of the sum that **my_avg** will compute is `5`.

```
SELECT my_sum(DISTINCT quantity, 5) FROM items
```

In the following example, you apply the user-defined aggregate named **my_max** to all values of the **quantity** column in the remote **items** table:

```
SELECT my_max(remote.quantity) FROM rdb@rserv:items remote
```

If the **my_max** aggregate is defined as EXECUTEANYWHERE, then the distributed query can be pushed to the remote database server, **rserv**, for execution. If the **my_max** aggregate is not defined as EXECUTEANYWHERE, then the distributed query scans the remote **items** table and computes the **my_max** aggregate on the local database server.

You cannot qualify a user-defined aggregate with the name of a remote database server, as the following example shows. In this case, the database server returns an error:

```
SELECT rdb@rserv:my_max(remote.quantity)
FROM rdb@rserv:items remote
```

For further information on user-defined aggregates, see "CREATE AGGREGATE" on page 2-104 and the discussion of user-defined aggregates in *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

Related Information

For a discussion of expressions in the context of the SELECT statement, see the *IBM Informix Guide to SQL: Tutorial*.

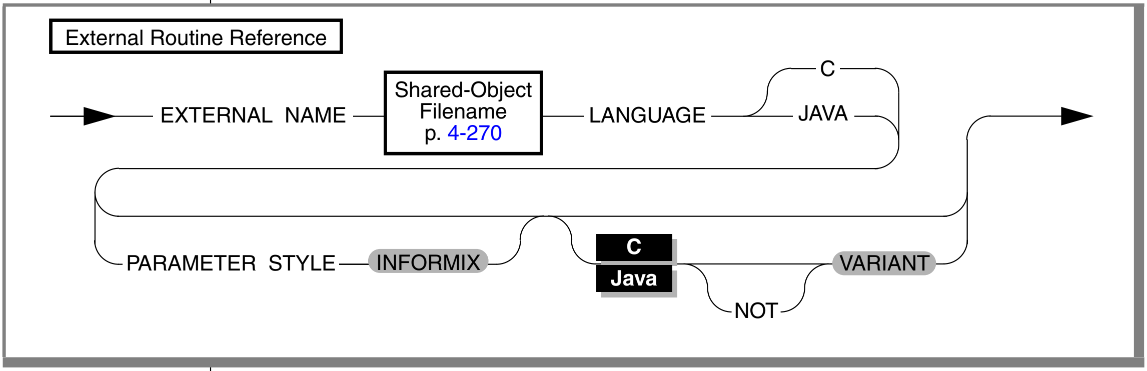
For discussions of column expressions, length functions, and the **TRIM** function, see the *IBM Informix GLS User's Guide*.

IDS

External Routine Reference

Use an External Routine Reference when you write an external routine.

Syntax



Usage

The External Routine Reference segment specifies the following information about an external routine:

- Pathname to the executable object code, stored in a shared-object file

  For C routines, this file is either a DLL or a shared library, depending on your operating system.

  For Java routines, this file is a jar file. Before you can create a UDR written in the Java language, you must assign a jar identifier to the external jar file with the **sqlj.install_jar** procedure. For more information, see "sqlj.install_jar" on page 2-418.
- The name of the language in which the UDR is written
- The parameter style of the UDR

  By default, the parameter style is INFORMIX. (This implies that if you specify an OUT parameter, the OUT argument is passed by reference.)
- The VARIANT or NOT VARIANT option, if you specify one

  SPL

  This option is not available for SPL routines. ♦

VARIANT or NOT VARIANT Option

A function is *variant* if it can return different results when it is invoked with the same arguments or if it modifies a database or variable state. For example, a function that returns the current date or time is a variant function.

By default, user-defined functions are variant. If you specify NOT VARIANT when you create or modify a function, it cannot contain any SQL statements.

If the function is nonvariant, the database server might cache the return variant functions. For more information on functional indexes, see "CREATE INDEX" on page 2-144.

To register a nonvariant function, add the NOT VARIANT option in this clause or in the Routine Modifier clause that is discussed in "Routine Modifier" on page 4-257. If you specify the modifier in both contexts, however, you must use the same modifier (either VARIANT or NOT VARIANT) in both clauses.

C

Example of a C User-Defined Function

The next example registers an external function named **equal( )** that takes two **point** data type values as arguments. In this example, **point** is an opaque data type that specifies the **x** and **y** coordinates of a two-dimensional point.

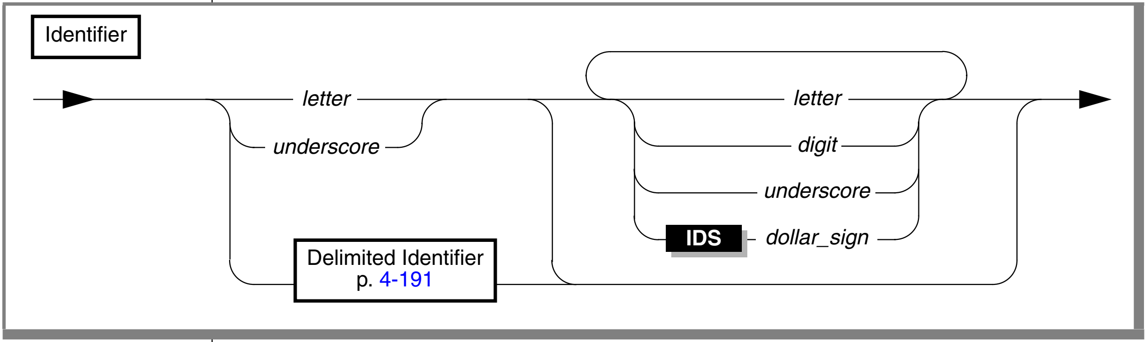
```
CREATE FUNCTION equal( a point, b point ) RETURNING BOOLEAN;
   EXTERNAL NAME "/usr/lib/point/lib/libbtype1.so(point1_equal)"
   LANGUAGE C
END FUNCTION
```

The function returns a single value of type BOOLEAN. The external name specifies the path to the C shared-object file where the object code of the function is stored. The external name indicates that the library contains another function, **point1_equal( )**, which is invoked while **equal( )** executes.

Identifier

An *identifier* specifies the simple name of a database object, such as a column, table, index, or view. Use the Identifier segment whenever you see a reference to an identifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	Integer in range 0 to 9	Cannot be the first character.	Literal number, p. 4-216
dollar_sign	Dollar-sign symbol ($)	Cannot be the first character.	Literal symbol entered from the keyboard.
letter	Upper- or lowercase letter of the alphabet	In the default locale, must be an ASCII character in the range of A to Z or a to z.	Literal symbol entered from the keyboard.
underscore	Underscore (_) character	Cannot substitute a blank space, hyphen, or other nonalphanumeric character.	Literal symbol entered from the keyboard.

Usage

This is a logical subset of "Database Object Name" on page 4-46, a segment that can specify the *owner*, *database*, and *database server* of external objects.

To include a blank space (ASCII 32) in an identifier, you must use a delimited identifier. It is recommended that you do not use the dollar sign ($) in identifiers, because this symbol is a special character whose inclusion in an identifier might cause conflicts with other syntax elements. For more information, see "Delimited Identifiers" on page 4-191.

An identifier can contain up to 128 bytes, inclusive. For example, the following table name is valid: **employee_information**.

GLS

If you are using a multibyte code set, keep in mind that the maximum length of an identifier refers to the number of bytes, not the number of characters.

For letter characters in nondefault locales, see "Support for Non-ASCII Characters in Identifiers" on page 4-191. For further information on the GLS aspects of identifiers, see the *IBM Informix GLS User's Guide*. ♦

IDS

E/C

The database server checks the internal version number of the client application and the setting of the **IFX_LONGID** environment variable to determine whether a client application supports long identifiers (up to 128 bytes in length). For more information, see the *IBM Informix Guide to SQL: Reference*. ♦

Use of Uppercase Characters

You can specify the name of a database object with uppercase characters, but the database server shifts the name to lowercase characters unless the **DELIMIDENT** environment variable is set and the name of the database object is enclosed in double quotes. In this case, the database server treats the name of the database object as a delimited identifier and preserves the uppercase characters in the name. For more information, see "Delimited Identifiers" on page 4-191.

Use of Keywords as Identifiers

Although you can use almost any word as an identifier, syntactic ambiguities can result from using keywords as identifiers in SQL statements. The statement might fail or might not produce the expected results. For a discussion of the syntactic ambiguities that can result from using keywords as identifiers and an explanation of workarounds for these problems, see "Potential Ambiguities and Syntax Errors" on page 4-194.

Delimited identifiers provide the easiest and safest way to use a keyword as an identifier without syntactic ambiguities. No workarounds are necessary for a keyword as a delimited identifier. For the syntax and usage of delimited identifiers, see "Delimited Identifiers" on page 4-191. Delimited identifiers require, however, that your code always use single (') quotes, rather than double (") quotes, to delimit character-string literals.

IDS

The keywords of the Informix implementation of SQL in Dynamic Server are listed in Appendix A, "Reserved Words for IBM Informix Dynamic Server." ♦

XPS

The keywords of SQL in Extended Parallel Server are listed in Appendix B, "Reserved Words for IBM Informix Extended Parallel Server." ♦

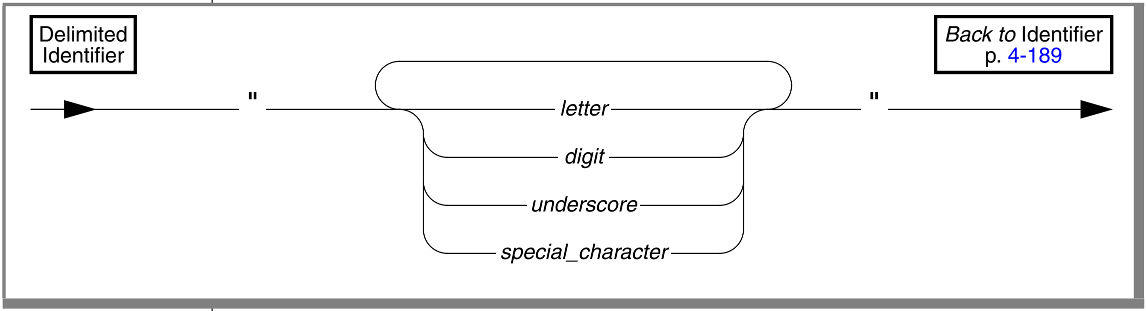
Tip: *If an error message seems unrelated to the statement that caused the error, check to see whether the statement uses a keyword as an undelimited identifier.*

GLS

Support for Non-ASCII Characters in Identifiers

In a nondefault locale, you can use any alphabetic character that your locale recognizes as a *letter* in an SQL identifier. This feature enables you to use non-ASCII characters in the names of some database objects. For objects that support non-ASCII characters, see the *IBM Informix GLS User's Guide*.

Delimited Identifiers



Element	Purpose	Restrictions	Syntax
digit	Integer in the range 0 to 9	Cannot be the first character.	Literal number, p. 4-216
letter	Letter that forms part of the delimited identifier	Letters in delimited identifiers are case-sensitive.	Literal value entered from the keyboard.
special_character	Nonalphanumeric character, such as #, $, or blank space	Must be an element in the code set of the locale.	Literal value entered from the keyboard.
underscore	Underscore (_) that forms part of the delimited identifier	None.	Literal value entered from the keyboard.

Delimited identifiers allow you to specify names for database objects that are otherwise identical to SQL keywords, such as TABLE, WHERE, DECLARE, and so on. The only database object for which you cannot use delimited identifiers is a database name.

Letters in delimited identifiers are case sensitive. If you are using the default locale, *letter* must be an upper- or lowercase character in the range a to z or A to Z (in the ASCII code set). If you are using a nondefault locale, *letter* must be an alphabetic character that the locale supports. For more information, see "Support for Non-ASCII Characters in Delimited Identifiers" on page 4-192.

Delimited identifiers are compliant with the ANSI/ISO standard for SQL.

When you create a database object, avoid including leading blank spaces or other whitespace characters between the first delimiting quotation mark and the first nonblank character of the delimited identifier. (Otherwise, you might not be able to reference the object in some contexts.) Delimited identifiers also require that your code always use single (') quotes, rather than double (") quotes, to delimit character-string literals.

Support for Nonalphanumeric Characters

You can use delimited identifiers to specify nonalphanumeric characters in the names of database objects. You cannot use delimited identifiers, however, to specify non-alphanumeric characters in the names of storage objects such as dbspaces and blobspaces.

GLS

Support for Non-ASCII Characters in Delimited Identifiers

When you are using a nondefault locale whose code set supports non-ASCII characters, you can specify non-ASCII characters in most delimited identifiers. The rule is that if you can specify non-ASCII characters in the undelimited form of the identifier, you can also specify non-ASCII characters in the delimited form of the same identifier. For a list of identifiers that support non-ASCII characters and for information on non-ASCII characters in delimited identifiers, see the *IBM Informix GLS User's Guide*.

Effect of DELIMIDENT Environment Variable

To use delimited identifiers, you must set the **DELIMIDENT** environment variable. If you set **DELIMIDENT**, strings enclosed in double quotes (**"**) are treated as identifiers and database objects enclosed in single quotes (**'**) are treated as strings. If the **DELIMIDENT** environment variable is not set, values enclosed in double quotes are also treated as strings.

If **DELIMIDENT** is set, the SELECT statement in the following example must be in single quotes in order to be treated as a quoted string:

```
PREPARE ... FROM 'SELECT * FROM customer'
```

If a delimited identifier is used in the SELECT statement that defines a view, then the **DELIMIDENT** environment variable must be set in order for the view to be accessed, even if the view name itself contains no special characters.

Examples of Delimited Identifiers

The next example shows how to create a table with a case-sensitive name:

```
CREATE TABLE "Power_Ranger" (...)
```

The following example creates a table whose name includes a whitespace character. If the table name were not enclosed by double (**"**) quotes, and if **DELIMIDENT** were not set, you could not use a blank space in the identifier.

```
CREATE TABLE "My Customers" (...)
```

The next example creates a table that has a keyword as the table name:

```
CREATE TABLE "TABLE" (...)
```

IDS

The following example shows how to delete all the rows from a table that is named FROM when you omit the keyword FROM in the DELETE statement:

```
DELETE “FROM”; ♦
```

Using Double Quotes Within a Delimited Identifier

To include a double quote (**"**) in a delimited identifier, you must precede the double quote (**"**) with another double quote ("), as this example shows:

```
CREATE TABLE "My""Good""Data" (...)
```

Potential Ambiguities and Syntax Errors

You can use almost any word as an SQL identifier, but syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. The following sections outline some potential pitfalls and workarounds.

Using the Names of Built-In Functions as Column Names

The following two examples show a workaround for using a built-in function as a column name in a SELECT statement. This workaround applies to the aggregate functions (AVG, COUNT, MAX, MIN, SUM) as well as the function expressions (algebraic, exponential and logarithmic, time, hex, length, dbinfo, trigonometric, and trim functions).

Using **avg** as a column name causes the next example to fail because the database server interprets **avg** as an aggregate function rather than as a column name:

```
SELECT avg FROM mytab -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **avg** as a column name as the following example shows:

```
SELECT "avg" from mytab -- successful
```

The workaround in the following example removes ambiguity by including a table name with the column name:

```
SELECT mytab.avg FROM mytab
```

If you use the keyword TODAY, CURRENT, or USER as a column name, ambiguity can occur, as the following example shows:

```
CREATE TABLE mytab (user char(10),
   CURRENT DATETIME HOUR TO SECOND,TODAY DATE)

INSERT INTO mytab VALUES('josh','11:30:30','1/22/1998')

SELECT user,current,today FROM mytab
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the built-in functions USER, CURRENT, and TODAY. Thus, instead of returning `josh`, `11:30:30`,`1/22/1998`, the SELECT statement returns the current user name, the current time, and the current date.

If you want to select the actual columns of the table, you must write the SELECT statement in one of the following ways:

```
SELECT mytab.user, mytab.current, mytab.today FROM mytab;
```

```
EXEC SQL select * from mytab;
```

Using Keywords as Column Names

Specific workarounds exist for using a keyword as a column name in a SELECT statement or other SQL statement. In some cases, more than one suitable workaround might be available.

Using ALL, DISTINCT, or UNIQUE as a Column Name

If you want to use the ALL, DISTINCT, or UNIQUE keywords as column names in a SELECT statement, you can take advantage of a workaround.

First, consider what happens when you try to use one of these keywords without a workaround. In the following example, using **all** as a column name causes the SELECT statement to fail because the database server interprets **all** as a keyword rather than as a column name:

```
SELECT all FROM mytab -- fails
```

You need to use a workaround to make this SELECT statement execute successfully. If the **DELIMIDENT** environment variable is set, you can use **all** as a column name by enclosing **all** in double quotes. In the following example, the SELECT statement executes successfully because the database server interprets **all** as a column name:

```
SELECT "all" from mytab -- successful
```

The workaround in the following example uses the keyword ALL with the column name **all**:

```
SELECT ALL all FROM mytab
```

The examples that follow show workarounds for using the keywords UNIQUE or DISTINCT as a column name in a CREATE TABLE statement.

This example fails to declare a column named **unique** because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER) -- fails
```

The workaround in the following example uses two SQL statements. The first statement creates the column **mycol**; the second statement renames the column **mycol** to **unique**.

```
CREATE TABLE mytab (mycol INTEGER)

RENAME COLUMN mytab.mycol TO unique
```

The workaround in the following example also uses two SQL statements. The first statement creates the column **mycol**; the second alters the table, adds the column **unique**, and drops the column **mycol**.

```
CREATE TABLE mytab (mycol INTEGER)

ALTER TABLE mytab
   ADD (unique INTEGER),
   DROP (mycol)
```

Using INTERVAL or DATETIME as a Column Name

The examples in this section show workarounds for using the keyword INTERVAL (or DATETIME) as a column name in a SELECT statement.

Using **interval** as a column name causes the following example to fail because the database server interprets **interval** as a keyword and expects it to be followed by an INTERVAL qualifier:

```
SELECT interval FROM mytab -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **interval** as a column name, as the following example shows:

```
SELECT "interval" from mytab -- successful
```

The workaround in the following example removes ambiguity by specifying a table name with the column name:

```
SELECT mytab.interval FROM mytab;
```

The workaround in the following example includes an owner name with the table name:

```
SELECT josh.mytab.interval FROM josh.mytab;
```

IDS

Using rowid as a Column Name

Every nonfragmented table has a virtual column named **rowid**. To avoid ambiguity, you cannot use **rowid** as a column name. Performing the following actions causes an error:

- Creating a table or view with a column named **rowid**
- Altering a table by adding a column named rowid
- Renaming a column to rowid

You can, however, use the term **rowid** as a table name.

```
CREATE TABLE rowid (column INTEGER, date DATE, char CHAR(20))
```

Important: *It is recommended that you use primary keys as an access method, rather than exploiting the **rowid** column.*

Using Keywords as Table Names

Examples in this section show workarounds that involve owner naming when the keyword STATISTICS or OUTER is a table name. (This workaround also applies to STATISTICS or OUTER as a view name or synonym.)

Using **statistics** as a table name causes the following example to fail because the database server interprets it as part of the UPDATE STATISTICS syntax rather than as a table name in an UPDATE statement:

```
UPDATE statistics SET mycol = 10
```

The workaround in the following example specifies an owner name with the table name, to avoid ambiguity:

```
UPDATE josh.statistics SET mycol = 10
```

Using **outer** as a table name causes the following example to fail because the database server interprets **outer** as a keyword for performing an outer join:

```
SELECT mycol FROM outer -- fails
```

The following successful example uses owner naming to avoid ambiguity:

```
SELECT mycol FROM josh.outer
```

Workarounds That Use the Keyword AS

In some cases, although a statement is not ambiguous and the syntax is correct, the database server returns a syntax error. The preceding pages show existing syntactic workarounds for several situations. You can use the AS keyword to provide a workaround for the exceptions.

You can use the AS keyword in front of column labels or table aliases.

The following example uses the AS keyword with a column label:

```
SELECT column_name AS display_label FROM table_name
```

The following example uses the AS keyword with a table alias:

```
SELECT select_list FROM table_name AS table_alias
```

Using AS with Column Labels

The examples in this section show workarounds that use the AS keyword with a column label. The first two examples show how you can use the keyword UNITS (or YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION) as a column label.

Using **units** as a column label causes this example to fail because the database server interprets it as a DATETIME qualifier for the column named **mycol**:

```
SELECT mycol units FROM mytab
```

The workaround in the following example includes the AS keyword:

```
SELECT mycol AS units FROM mytab;
```

The following examples use w the AS or FROM keyword as a column label.

Using **as** as a column label causes the following example to fail because the database server interprets **as** as identifying **from** as a column label and thus finds no required FROM clause:

```
SELECT mycol as from mytab -- fails
```

The following successful example repeats the AS keyword:

```
SELECT mycol AS as from mytab
```

Using **from** as a column label causes the following example to fail because the database server expects a table name to follow the first **from**:

```
SELECT mycol from FROM mytab -- fails
```

This example uses the AS keyword to identify the first **from** as a column label:

```
SELECT mycol AS from FROM mytab
```

Using AS with Table Aliases

Examples in this section show workarounds that use the AS keyword with a table alias. The first pair shows how to use the ORDER, FOR, GROUP, HAVING, INTO, UNION, WITH, CREATE, GRANT, or WHERE keyword as a table alias.

Using **order** as a table alias causes the following example to fail because the database server interprets **order** as part of an ORDER BY clause:

```
SELECT * FROM mytab order -- fails
```

The workaround in the following example uses the keyword AS to identify **order** as a table alias:

```
SELECT * FROM mytab AS order;
```

The next two examples show how to use the keyword WITH as a table alias.

Using **with** as a table alias causes the next example to fail because the database server interprets **with** as part of the WITH CHECK OPTION syntax:

```
EXEC SQL select * from mytab with; -- fails
```

The workaround in the following example uses the keyword AS to identify **with** as a table alias:

```
EXEC SQL select * from mytab as with;
```

The following two examples show how to use the keyword CREATE (or GRANT) as a table alias.

Using **create** as a table alias causes the following example to fail because the database server interprets the keyword as part of the syntax to create a new database object, such as a table, synonym, or view:

```
EXEC SQL select * from mytab create; -- fails
```

The workaround in the following example uses the keyword AS to identify **create** as a table alias:

```
EXEC SQL select * from mytab as create;
```

Fetching Keywords as Cursor Names

In a few situations, no workaround exists for the syntactic ambiguity that occurs when a keyword is used as an identifier in an SQL program.

In the following example, the FETCH statement specifies a cursor named **next**. The FETCH statement generates a syntax error because the preprocessor interprets **next** as a keyword, signifying the next row in the active set and expects a cursor name to follow **next**. This occurs whenever the keyword NEXT, PREVIOUS, PRIOR, FIRST, LAST, CURRENT, RELATIVE, or ABSOLUTE is used as a cursor name.

```
/* This code fragment fails */
EXEC SQL declare next cursor for
   select customer_num, lname from customer;

EXEC SQL open next;
EXEC SQL fetch next into :cnum, :lname;
```

Using Keywords as Variable Names in UDRs

If you use any of the following keywords as identifiers for variables in a user-defined routine (UDR), you can create ambiguous syntax:

CURRENT
DATETIME
GLOBAL
INTERVAL
NULL
OFF
ON
OUT
PROCEDURE
SELECT

Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT

A UDR cannot insert a variable that was declared using the CURRENT, DATETIME, INTERVAL, or NULL keywords as the name. For example, if you declare a variable called **null**, when you try to insert the value **null** into a column, you receive a syntax error, as the following example shows:

```
CREATE PROCEDURE problem()
. . .
DEFINE null INT;
LET null = 3;
INSERT INTO tab VALUES (null); -- error, inserts NULL, not 3
```

Using NULL and SELECT in a Condition

If you define a variable with the name **null** or **select**, using it in a condition that uses the IN keyword is ambiguous. The following example shows three conditions that cause problems: in an IF statement, in a WHERE clause of a SELECT statement, and in a WHILE condition:

```
CREATE PROCEDURE problem()
. . .
DEFINE x,y,select, null, INT;
DEFINE pfname CHAR[15];
LET x = 3; LET select = 300;
LET null = 1;
IF x IN (select, 10, 12) THEN LET y = 1; -- problem if

IF x IN (1, 2, 4) THEN
SELECT customer_num, fname INTO y, pfname FROM customer
   WHERE customer IN (select , 301 , 302, 303); -- problem in

WHILE x IN (null, 2) -- problem while
. . .
END WHILE;
```

You can use the variable **select** in an IN list if you ensure it is not the first element in the list. The workaround in the following example corrects the IF statement that the preceding example shows:

```
 IF x IN (10, select, 12) THEN LET y = 1; -- problem if
```

No workaround exists to using **null** as a variable name and attempting to use that variable in an IN condition.

Using ON, OFF, or PROCEDURE with TRACE

If you define an SPL variable called **on**, **off**, or **procedure**, and you attempt to use it in a TRACE statement, the value of the variable is not traced. Instead, the TRACE ON, TRACE OFF, or TRACE PROCEDURE statements execute. You can trace the value of the variable by making the variable into a more complex expression.

The following example shows the ambiguous syntax and the workaround:

```
DEFINE on, off, procedure INT;

TRACE on;   --ambiguous
TRACE 0+ on;--ok
TRACE off;  --ambiguous
TRACE ''||off;--ok

TRACE procedure;--ambiguous
TRACE 0+procedure;--ok
```

Using GLOBAL as a Variable Name

If you attempt to define a variable with the name **global**, the DEFINE operation fails. The syntax that the following example shows conflicts with the syntax for defining global variables:

```
DEFINE global INT; -- fails;
```

If the **DELIMIDENT** environment variable is set, you could use **global** as a variable name, as the following example shows:

```
DEFINE "global" INT; -- successful
```

Important: *Although workarounds that the preceding sections show can avoid compilation or runtime syntax conflicts from keywords used as identifiers, keep in mind that such identifiers tend to make code more difficult understand and maintain.*

Using EXECUTE, SELECT, or WITH as Cursor Names

Do not use an EXECUTE, SELECT, or WITH keyword as the name of a cursor. If you try to use one of these keywords as the name of a cursor in a FOREACH statement, the cursor name is interpreted as a keyword in the FOREACH statement. No workaround exists.

The following example does not work:

```
DEFINE execute INT;
FOREACH execute FOR SELECT col1 -- error, looks like
                                -- FOREACH EXECUTE PROCEDURE
    INTO var1 FROM tab1;
```

SELECT Statements in WHILE and FOR Statements

If you use a SELECT statement in a WHILE or FOR loop, and if you need to enclose it in parentheses, enclose the entire SELECT statement in a BEGIN…END statement block. The SELECT statement in the first WHILE statement in the following example is interpreted as a call to the procedure **var1**; the second WHILE statement is interpreted correctly:

```
DEFINE var1, var2 INT;
WHILE var2 = var1
   SELECT col1 INTO var3 FROM TAB -- error, interpreted as call var1()
   UNION
   SELECT co2 FROM tab2;
END WHILE

WHILE var2 = var1
   BEGIN
      SELECT col1 INTO var3 FROM TAB -- ok syntax
      UNION
      SELECT co2 FROM tab2;
   END
END WHILE
```

SET Keyword in the ON EXCEPTION Statement

If you use a statement that begins with the keyword SET in ON EXCEPTION, you must enclose it in a BEGIN … END statement block. The following list shows some of the SQL statements that begin with the keyword SET:

SET AUTOFREE
SET CONNECTION
SET CONSTRAINTS
SET DATASKIP
SET DEBUG FILE
SET DEFERRED PREPARE
SET DESCRIPTOR
SET EXPLAIN
SET INDEX
SET INDEXES
SET ISOLATION
SET LOCK MODE
SET LOG
SET OPTIMIZATION
SET PDQPRIORITY
SET PLOAD FILE
SET ROLE
SET SCHEDULE LEVEL
SET SESSION AUTHORIZATION
SET STATEMENT CACHE
SET TABLE
SET TRANSACTION

The following examples show the incorrect and correct use of a SET LOCK MODE statement inside an ON EXCEPTION statement.

The following ON EXCEPTION statement returns an error because the SET LOCK MODE statement is not enclosed in a BEGIN … END statement block:

```
ON EXCEPTION IN (-107)
   SET LOCK MODE TO WAIT; -- error, value expected, not 'lock'
END EXCEPTION
```

The following ON EXCEPTION statement executes successfully because the SET LOCK MODE statement is enclosed in a BEGIN … END statement block:

```
ON EXCEPTION IN (-107)
   BEGIN
   SET LOCK MODE TO WAIT; -- ok
   END
END EXCEPTION
```

Related Information

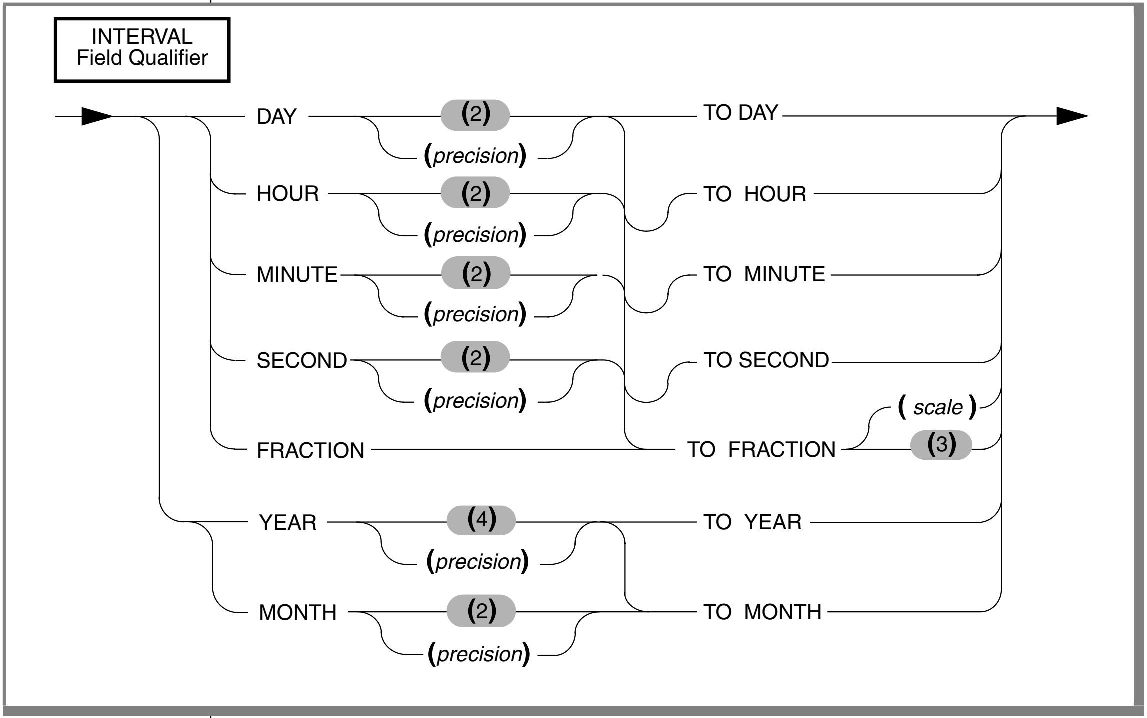
For a discussion of owner naming, see your *Performance Guide*.

For a discussion of identifiers that support non-ASCII characters and a discussion of non-ASCII characters in delimited identifiers, see the *IBM Informix GLS User's Guide*.

INTERVAL Field Qualifier

The INTERVAL field qualifier specifies the units for an INTERVAL value. Use the INTERVAL Field Qualifier segment whenever you see a reference to an INTERVAL field qualifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
scale	Integer number of digits in FRACTION field. Default is 5.	Must be in the range from 1 to 5.	Literal number, p. 4-216
precision	Integer number of digits in the largest time unit that the INTERVAL includes. For YEAR, the default is 4. For all other time units, the default is 2.	Must be in the range from 1 to 9.	Literal number, p. 4-216

Usage

This segment specifies the precision and scale of an INTERVAL data type.

A keyword specifying the *largest* time unit must be the first keyword, and a keyword specifying the *smallest* time unit must follow the TO keyword. These can be the same keyword. This segment resembles the syntax of a "DATETIME Field Qualifier" on page 4-65, but with these exceptions:

- If the largest time unit keyword is YEAR or MONTH, the smallest time unit keyword cannot specify a time unit smaller than MONTH.
- You can specify up to 9-digit precision after the first time unit, unless FRACTION is the first time unit (in which case the limit is 5 digits).

The next two examples show INTERVAL data types with YEAR TO MONTH qualifiers. The first example can hold an interval of up to 999 years and 11 months, because it gives `3` as the precision of the YEAR field. The second example uses the default precision on the YEAR field, so it can hold an interval of up to 9,999 years and 11 months.

```
YEAR (3) TO MONTH

YEAR TO MONTH
```

When you want a value to specify only one kind of time unit, the first and last qualifiers are the same. For example, an interval of whole years is qualified as YEAR TO YEAR or YEAR (5) TO YEAR, for an interval of up to 99,999 years.

The following examples show several forms of INTERVAL field qualifiers:

```
YEAR(5) TO MONTH

DAY (5) TO FRACTION(2)

DAY TO DAY

FRACTION TO FRACTION (4)
```

Related Information

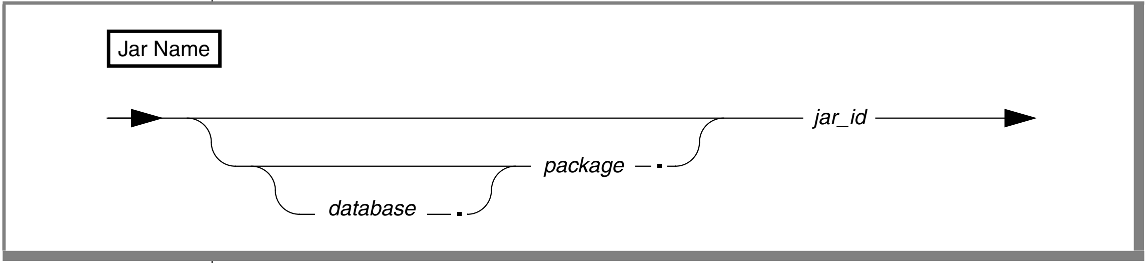
For information about how to specify INTERVAL field qualifiers and use INTERVAL data in arithmetic and relational operations, see the discussion of the INTERVAL data type in the *IBM Informix Guide to SQL: Reference*.

IDS

Jar Name

Use the Jar Name segment to specify the name of a jar ID. Use this segment whenever you see a reference to Jar Name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	Database in which to install or access *jar_id* Default is the current database.	Fully qualified *database***.***package***.***jar_id* identifier must not exceed 255 bytes.	Database Name, p. 4-189
jar_id	The **.jar** file that contains the Java class to be accessed	File must exist in *database***.***package.*	Identifier, p. 4-189
package	Name of the package	Package must exist in *database*.	Identifier, p. 4-189

If a Jar name is specified as a character string value to the **sqlj.install_jar**, **sqlj.replace_jar**, or **sqlj.remove_jar** procedures, then any identifiers in the jar name that are delimited identifiers will include the surrounding double quote characters.

Before you can access a *jar_id* in any way (including its use in a CREATE FUNCTION or CREATE PROCEDURE statement), it must be defined in the current database with the **install_jar( )** procedure. For more information, see "EXECUTE PROCEDURE" on page 2-414.

Related Information

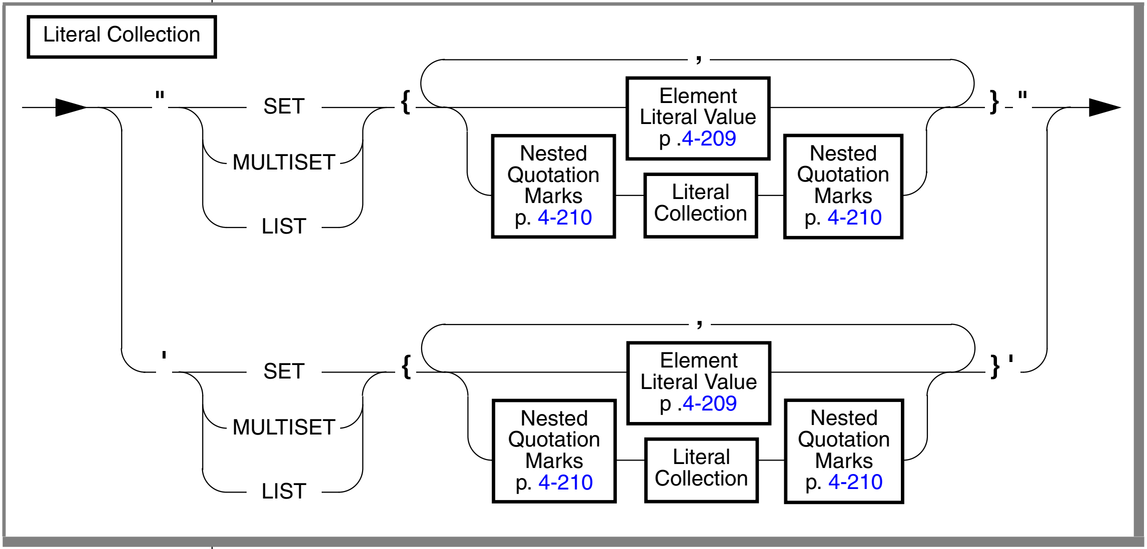
For information on how to update the three-part names of JAR files after you rename the database, see the *J/Foundation Developer's Guide*.

IDS

Literal Collection

Use the Literal Collection segment to specify values for a collection data type. For the syntax of expressions that return values of individual elements within a collection, see "Collection Constructors" on page 4-108.

Syntax


Literal Collection
"
SET
MULTISET
LIST
{
,
Element Literal Value p .4-209
Nested Quotation Marks p. 4-210
Literal Collection
Nested Quotation Marks p. 4-210
}
"
'
SET
MULTISET
LIST
{
,
Element Literal Value p .4-209
Nested Quotation Marks p. 4-210
Literal Collection
Nested Quotation Marks p. 4-210
}
'

Usage

You can specify literal collection values for SET, MULTISET, or LIST data types.

To specify a single literal-collection value, specify the collection type and the literal values. The following SQL statement inserts four integer values into a column called **set_col** that was declared as `SET(INT NOT NULL)`:

```
INSERT INTO table1 (set_col) VALUES ("SET{6, 9, 9, 4}")
```

Specify an empty collection with an empty pair of braces ({ }) symbols. This example inserts an empty list into n column **list_col** that was declared as LIST(INT NOT NULL):

```
INSERT INTO table2 (list_col) VALUES ("LIST{}")
```

A pair of single (') or double (") quotes must delimit the collection.

SQLE

If you are passing a literal collection as an argument to an SPL routine, make sure that there is a blank space between the parentheses that surround the arguments and the quotation marks that indicate the beginning and end of the literal collection. ♦

If you specify a collection as a literal value in a row-string literal, you can omit the quotation marks around the collection itself. Only the outermost quotation marks that delimit the row-string literal are necessary. No quotation marks need surround the nested collection type. For an example, see "Literals for Nested Rows" on page 4-221.

Element Literal Value

The diagram for "Literal Collection" on page 4-208 refers to this section.

Elements of a collection can be literal values for the following data types.

For a Collection of Type	Literal Value Syntax
BOOLEAN	`t` or `f`, representing TRUE or FALSE as a quoted string.
CHAR, VARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, DATE	Quoted String, p. 4-243
DATETIME	Literal DATETIME, p. 4-212
DECIMAL, MONEY, FLOAT, INTEGER, INT8, SMALLFLOAT, SMALLINT	Literal Number, p.4-216
INTERVAL	Literal INTERVAL, p. 4-214
Opaque data types	Quoted String, p. 4-243. The string literal must be recognized by the input support function for the associated opaque type.
ROW Type	"Literal Row" on page 4-218. When the collection element type is a named ROW type, you do not need to cast the inserted values to the named ROW type.

***Important:** You cannot specify the simple-large-object data types (BYTE and TEXT) as the element type for a collection.*

Quoted strings must be specified with a different type of quotation mark than the quotation marks that encompass the collection, so that the database server can parse the quoted strings. Thus, if you use double (") quotation marks to specify the collection, use single (') quotation marks to specify individual, quoted-string elements.

Nested Quotation Marks

The diagram for "Literal Collection" on page 4-208 refers to this section.

A *nested collection* is a collection that is the element type for another collection.

Whenever you nest collection literals, use nested quotation marks. In these cases, you must follow the rule for nesting quotation marks. Otherwise, the database server cannot correctly parse the strings.

The general rule is that you must double the number of quotation marks for each new level of nesting. For example, if you use double (") quotation marks for the first level, you must use two double quotation marks for the second level, four double quotation marks for the third level, eight for the fourth level, sixteen for the fifth level, and so on.

Likewise, if you use single (') quotes for the first level, you must use two single quotation marks for the second level and four single quotation marks for the third level. There is no limit to the number of levels you can nest, as long as you follow this rule.

The following examples illustrate the case for two levels of nested collection literals, using double (") quotation marks. Here table **tab5** is a single-column table whose only column, **set_col**, is a nested collection type.

The following statement creates the **tab5** table:

```
CREATE TABLE tab5 (set_col SET(SET(INT NOT NULL) NOT NULL));
```

The following statement inserts values into the table **tab5**:

```
INSERT INTO tab5 VALUES ( "SET{""SET{34, 56, 23, 33}""}" )
```

For each literal value, the opening quotation mark and the closing quotation mark must match. Thus, if you open a literal with two double quotes, you must close that literal with two double quotes (`""a literal value""`).

E/C

To specify nested quotation marks within an SQL statement in an ESQL/C program, use the C escape character for every double quote inside a single-quote string. Otherwise, the ESQL/C preprocessor cannot correctly interpret the literal collection value. For example, the preceding INSERT statement on the **tab5** table would appear in an ESQL/C program as follows:

```
EXEC SQL insert into tab5
   values ('set{\"set{34, 56, 23, 33}\"}');
```

For more information, see the chapter on complex data types in the *IBM Informix ESQL/C Programmer's Manual*. ♦

If the collection is a nested collection, you must include the collection-constructor syntax for each level of collection type. Suppose you define the following column:

```
nest_col SET(MULTISET (INT NOT NULL) NOT NULL)
```

The following statement inserts three elements into the **nest_col** column:

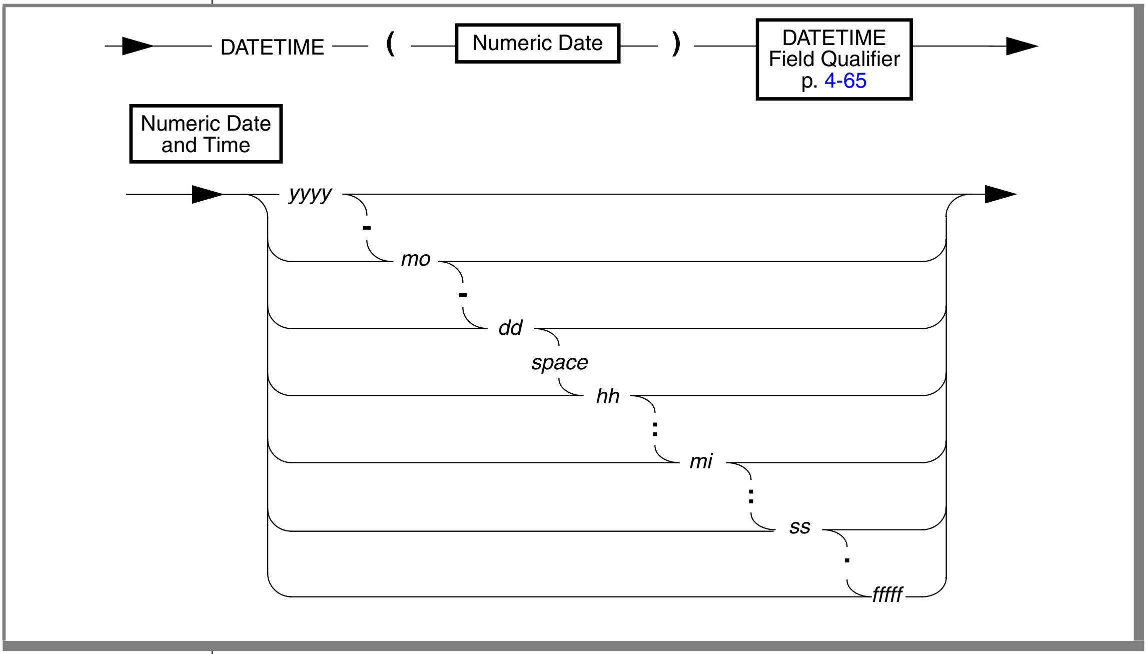
```
INSERT INTO tabx (nest_col)
   VALUES ("SET{'MULTISET{1, 2, 3}'}")
```

To learn how to use quotation marks in INSERT statements, see "Nested Quotation Marks" on page 4-210.

Literal DATETIME

The Literal DATETIME segment specifies a DATETIME value. Use this segment when you see a reference to a literal DATETIME in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	Day of month, expressed in digits	0 ≤ *dd* ≤ 28, 29, 30, or 31	Literal Number, p. 4-216
fffff	Fraction of a second, in digits	0 ≤ *fffff* ≤ 9999	Literal Number, p. 4-216
hh	Hour of day, expressed in digits	0 ≤ *hh* ≤ 23	Literal Number, p. 4-216
mi	Minute of hour, expressed in digits	0 ≤ *mi* ≤ 59	Literal Number, p. 4-216
mo	Month of year, expressed in digits	1 ≤ *mo* ≤ 12	Literal Number, p. 4-216
space	Blank space (ASCII 32)	Only 1 blank character allowed	Literal blank space
ss	Second of minute, in digits	0 ≤ *ss* ≤ 59	Literal Number, p. 4-216
yyyy	Year, expressed in digits	You can specify up to 4 digits	Literal Number, p. 4-216

Usage

You must specify both a numeric date and a DATETIME field qualifier for this date in the Literal DATETIME segment. The DATETIME field qualifier must correspond to the numeric date you specify. For example, if you specify a numeric date that includes a year as the largest unit and a minute as the smallest unit, you must specify YEAR TO MINUTE as the DATETIME field qualifier.

If you specify 2 digits for the year, the database server uses the setting of the **DBCENTURY** environment variable to expand the abbreviated year value to four digits. If the **DBCENTURY** is not set, the first two digits of the current year are used to expand the abbreviated year value.

The following examples show literal DATETIME values:

```
DATETIME (97-3-6) YEAR TO DAY

DATETIME (09:55:30.825) HOUR TO FRACTION

DATETIME (97-5) YEAR TO MONTH
```

The following example shows a literal DATETIME value used with the EXTEND function:

```
EXTEND (DATETIME (1997-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE (3) TO MINUTE
```

Related Information

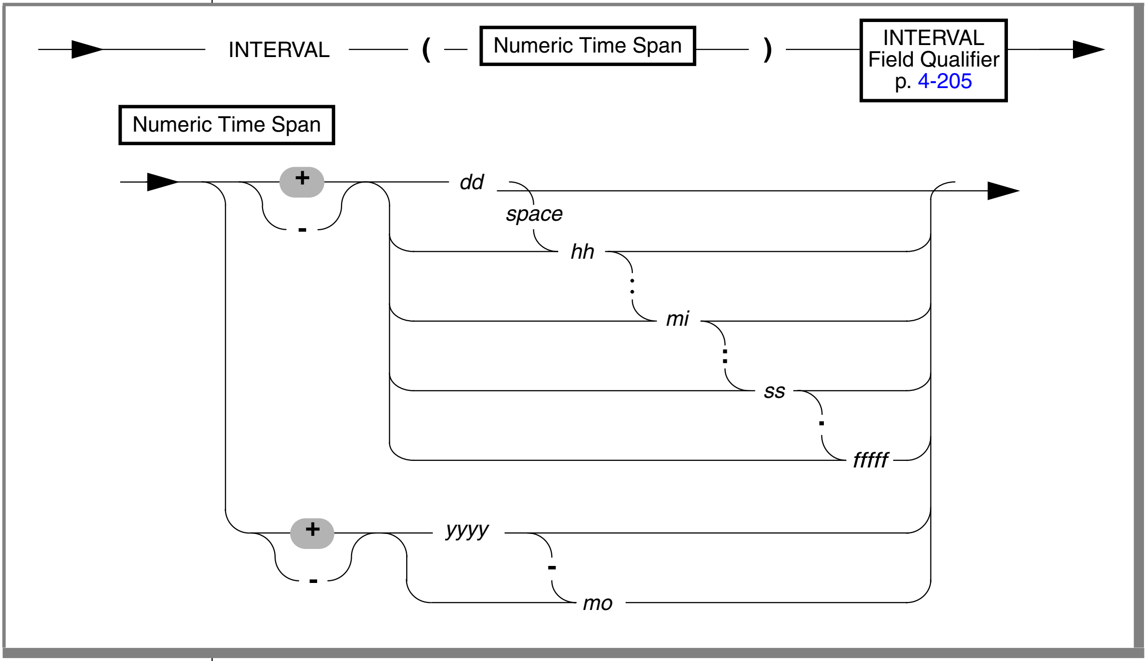
For discussions of the DATETIME data type and the **DBCENTURY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

For a discussion of how to customize DATETIME values for a locale, see the *IBM Informix GLS User's Guide*.

Literal INTERVAL

The Literal INTERVAL segment specifies a literal INTERVAL value. Use this whenever you see a reference to a literal INTERVAL in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	Number of days	-10**10 < *dd* < 10**10	Literal Number, p. 4-216
fffff	Fractions of a second	0 ≤ *fffff* ≤ 9999	Literal Number, p. 4-216
hh	Number of hours	If not first, 0 ≤ *hh* ≤ 23	Literal Number, p. 4-216
mi	Number of minutes	If not first, 0 ≤ *mi* ≤ 59	Literal Number, p. 4-216
mo	Number of months	If not first, 0 ≤ *mo* ≤ 11	Literal Number, p. 4-216
space	Blank space (ASCII 32)	Only 1 blank character allowed	Literal blank space
ss	Number of seconds	If not first, 0 ≤ *ss* ≤ 59	Literal Number, p. 4-216
yyyy	Number of years	-10**10 < *yyyy* < 10**10	Literal Number, p. 4-216

Usage

Unlike DATETIME literals, INTERVAL literals can include the unary plus (**+**) or unary minus (**-**) sign. If you specify no sign, the default is plus.

The precision of the first time unit can be specified by the INTERVAL qualifier. Except for FRACTION, which can have no more than 5 digits of precision, the first time unit can have up to 9 digits of precision, if you specified a nondefault precision in the declaration of the INTERVAL column or variable.

The following examples show literal INTERVAL values:

```
INTERVAL (3-6) YEAR TO MONTH
INTERVAL (09:55:30.825) HOUR TO FRACTION
INTERVAL (40 5) DAY TO HOUR
INTERVAL (299995.2567) SECOND(6) TO FRACTION(4)
```

Only the last of these examples has nondefault precision. For the syntax of declaring the precision of INTERVAL data types and the default values for each time unit, refer to "INTERVAL Field Qualifier" on page 4-205.

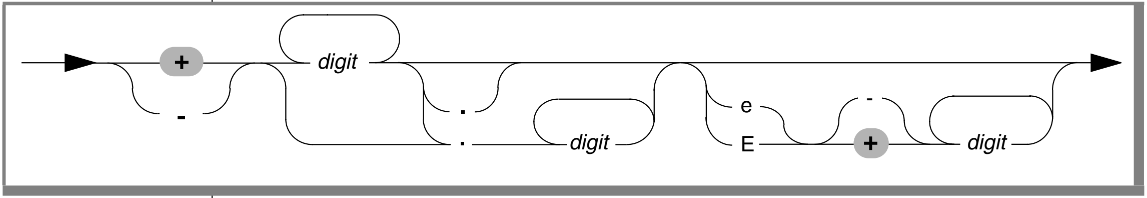
Related Information

For information on how to use INTERVAL data in arithmetic and relational operations, see the discussion of the INTERVAL data type in the *IBM Informix Guide to SQL: Reference*.

Literal Number

A *literal number* is the base-10 representation of a real number as an integer, as a fixed-point decimal number, or in exponential notation. Use the Literal Number segment whenever you see a reference to a literal number in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	Integer in the range 0 through 9	Must be ASCII digit.	Literal entered from the keyboard.

Usage

You cannot include comma (**,**) or blank (ASCII 32). The unary plus (**+**) or minus (**-**) sign can precede a literal number, mantissa, or exponent.

GLS

You cannot include non-ASCII digits in literal numbers, such as the Hindi numbers that some nondefault locales support. ♦

Integer Literals

In many contexts, a literal number is restricted to an integer literal. An integer has no fractional part and cannot include a decimal point. Built-in data types of SQL that can be exactly represented as literal integers include INT8, INT, SMALLINT, SERIAL, SERIAL8, and DECIMAL(*p*, 0).

If you use the representation of a number in a base other than 10 (such as a binary, octal, or hexadecimal number) in any context where a literal integer is valid, the database server will attempt to interpret the value as a base-10 literal integer. For most data values, the result will be incorrect.

The following examples show some valid literal integers:

```
10              -27              +25567
```

Thousands separators (such as comma symbols) are not valid in literal integers, nor in any other literal number.

Fixed-Point Decimal Literals

Fixed-point decimal literals can exactly represent DECIMAL(*p*,*s*) and MONEY values. These can include a decimal point:

```
-123.456        00123456        +123456.0
```

The digits to the right of the decimal point in these examples are the fractional portions of the numbers.

Floating-Point Decimal Literals

Floating-point literals can exactly represent FLOAT, SMALLFLOAT, and DECIMAL(*p*) values, using a decimal point or exponential notation, or both. They can approximately represent real numbers in exponential notation. The next examples show floating point numbers:

```
-123.45E6       1.23456E2       123456.0E-3
```

The `E` in the previous examples is the symbol for exponential notation. The digit that follows `E` is the value of the exponent. For example, the number 3E5 (or 3E+5) means 3 multiplied by 10 to the fifth power, and the number 3E-5 means 3 multiplied by the reciprocal of 10 to the fifth power.

Literal Numbers and the MONEY Data Type

When you use a literal number as a MONEY value, do not include a currency symbol or include commas. The **DBMONEY** environment variable or the locale file can format how MONEY values are displayed in output.

Related Information

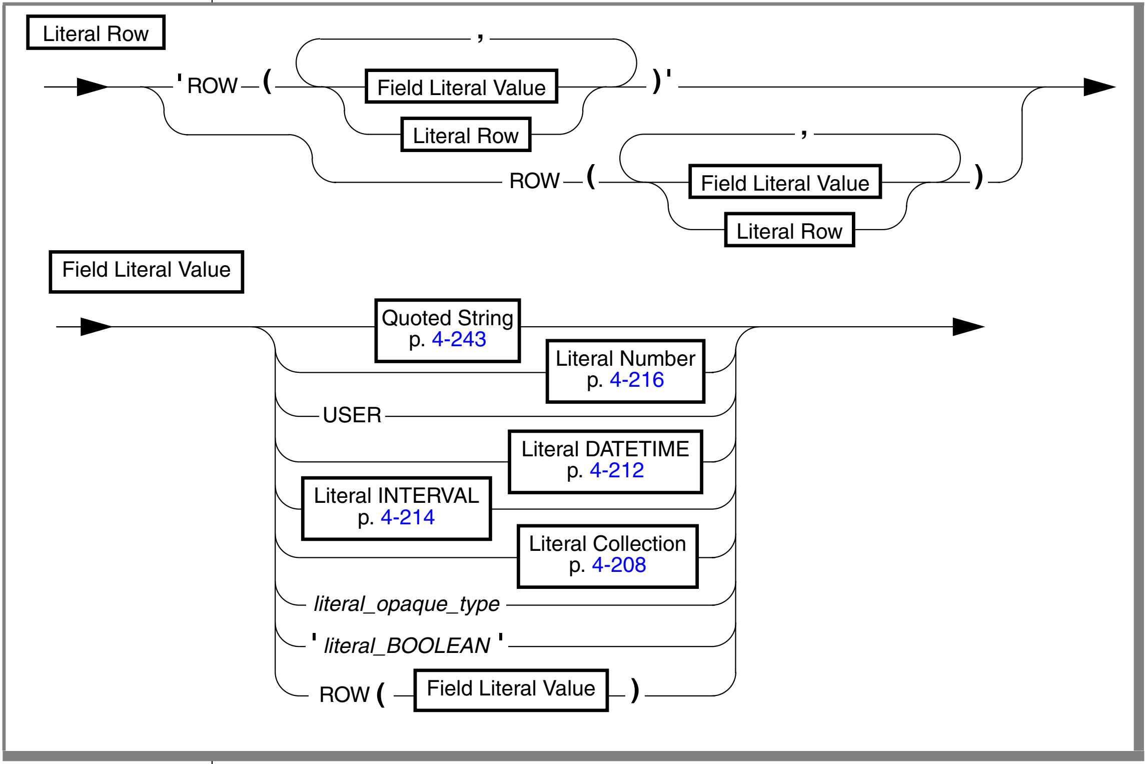
For discussions of numeric data types, such as DECIMAL, FLOAT, INTEGER, and MONEY, see the *IBM Informix Guide to SQL: Reference*.

IDS

Literal Row

The Literal Row segment specifies the syntax for literal values of named and unnamed ROW data types. For syntax that allows you to use expressions that evaluate to field values, see "ROW Constructors" on page 4-106.

Syntax.



Element	Purpose	Restrictions	Syntax
literal_opaque_type	Literal representation for an opaque data type	Must be a literal that is recognized by the input support function for the associated opaque type.	Defined by the developer of the opaque data type.
literal_BOOLEAN	Literal representation of a BOOLEAN value	Must be either `'t'` (= TRUE) or `'f'` (= FALSE) specified as a quoted string.	Quoted String, p. 4-243

Usage

You can specify literal values for named ROW and unnamed ROW data types. A ROW constructor introduces a literal row value, which can optionally be enclosed between quotation marks.

The format of the value for each field of the ROW type must be compatible with the data type of the corresponding field.

Important: *You cannot specify simple-large-object data types (BYTE or TEXT) as the field type for a row.*

Fields of a row can be literal values for the data types in the following table.

For a Field of Type	Literal Value Syntax
BOOLEAN	`t` or `f`, representing TRUE or FALSE
CHAR, VARCHAR, LVARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, DATE	Quoted String, p. 4-243
DATETIME	Literal DATETIME, p. 4-212
DECIMAL, MONEY, FLOAT, INTEGER, INT8, SMALLFLOAT, SMALLINT	Literal Number, p.4-216
INTERVAL	Literal INTERVAL, p. 4-214
Opaque data types	Quoted String, p. 4-243 The string must be a literal that is recognized by the input support function for the associated opaque type.
Collection type (SET, MULTISET, LIST)	"Literal Collection" on page 4-208 For information on literal collection values as variable or column values, see "Nested Quotation Marks" on page 4-210. For information on literal collection values for a ROW type, see "Literals for Nested Rows" on page 4-221.
Another ROW type (named or unnamed)	For information on ROW type values, see "Literals for Nested Rows" on page 4-221.

Literals of an Unnamed Row Type

To specify a literal value for an unnamed ROW type, introduce the literal row with the ROW constructor; you must enclose the values between parentheses. For example, suppose that you define the **rectangles** table as follows:

```
CREATE TABLE rectangles
(
   area FLOAT,
   rect ROW(x INTEGER, y INTEGER, length FLOAT, width FLOAT),
)
```

The following INSERT statement inserts values into the **rect** column of the **rectangles** table:

```
INSERT INTO rectangles (rect)
   VALUES ("ROW(7, 3, 6.0, 2.0)")
```

Literals of a Named Row Type

To specify a literal value for a named ROW, type, introduce the literal row with the ROW type constructor and enclose the literal values for each field in parentheses. In addition, you can cast the row literal to the appropriate named ROW type to ensure that the row value is generated as a named ROW type. The following statements create the named ROW type **address_t** and the **employee** table:

```
CREATE ROW TYPE address_t
(
street CHAR(20),
city CHAR(15),
state CHAR(2),
zipcode CHAR(9)
);

CREATE TABLE employee
(
   name CHAR(30),
   address address_t
);
```

The following INSERT statement inserts values into the **address** column of the **employee** table:

```
INSERT INTO employee (address)
VALUES (
"ROW('103 Baker St', 'Tracy','CA', 94060)"::address_t)
```

Literals for Nested Rows

If the literal value is for a nested row, specify the ROW type constructor for each row level. If you include quotation marks as delimiters, they should enclose the outermost row. For example, suppose that you create the following **emp_tab** table:

```
CREATE TABLE emp_tab
(
   emp_name CHAR(10),
   emp_info ROW( stats ROW(x INT, y INT, z FLOAT))
);
```

The following INSERT statement adds a row to the **emp_tab** table:

```
INSERT INTO emp_tab
VALUES ('joe boyd', "ROW(ROW(8,1,12.0))" )
```

Similarly, if the row-string literal contains a nested collection, only the outermost literal row can be enclosed between quotation marks. Do not put quotation marks around an inner, nested collection type.

Related Information

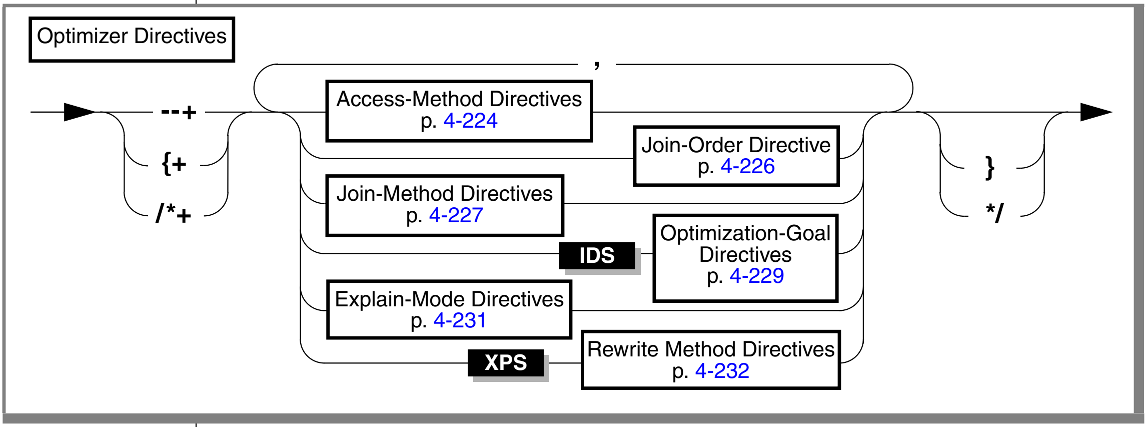
Related statements: CREATE ROW TYPE, INSERT, UPDATE, and SELECT

For information on ROW constructors, see the Expression segment. See also the Collection Literal segment.

Optimizer Directives

The Optimizer Directives segment specifies keywords that you can use to partially or fully specify the query plan of the optimizer. Use this segment whenever you see a reference to Optimizer Directives in a syntax diagram.

Syntax


Optimizer Directives
--+
{+
/*+
,
Access-Method Directives p. 4-224
Join-Order Directive p. 4-226
Join-Method Directives p. 4-227
IDS
Optimization-Goal Directives p. 4-229
Explain-Mode Directives p. 4-231
XPS
Rewrite Method Directives p. 4-232
}
*/

Usage

Optimizer directives are valid in any query of a DELETE, SELECT, or UPDATE statement. Use one or more directives to partially or fully specify the query plan of the optimizer. The scope of the directive is the current query only.

Directives are enabled by default. To obtain information about how specified directives are processed, view the output of the SET EXPLAIN statement. To disable directives, set the **IFX_DIRECTIVES** environment variable to `0` or `OFF`, or set the DIRECTIVES parameter in the ONCONFIG file to `0`.

Optimizer Directives as Comments

Optimizer directives require the SQL or C comment indicators as delimiters. If **{+** are the opening delimiters, you must use **}** as the closing delimiter. If **/*** are the opening delimiters, then you must use ***/** as the closing delimiters. If **--+** are the opening delimiters, then no closing delimiter is needed.

An optimizer directive or a string of optimizer directives immediately follows the DELETE, SELECT, or UPDATE keyword in the form of a comment. After the comment symbol, the first character in an optimizer directive is always a plus (+) sign. No blank space or other whitespace character is allowed between the comment indicator and the plus sign.

You can use any of the following comment indicators:

- A double hyphen (--) delimiter

  The double hyphen needs no closing symbol because it specifies only the remainder of the current line as comment. When you use this style, include the optimizer directive on only the current line.

- Braces ({   }) delimiters

  The comment extends from the left brace ({) until the next right (}) brace; this can be in the same line or in some subsequent line.

- C-language style slash and asterisk (/*   */) paired delimiters

  The comment extends from the initial slash-asterisk (/*) pair until the next asterisk-slash (*/) characters in the same line or in some subsequent line.

  E/C

  In ESQL/C, the **-keep** command option to the **esql** compiler must be specified when you use C-style comments. ♦

For more information on SQL comment indicators, see "How to Enter SQL Comments" on page 1-6.

If you specify multiple directives in the same query, you must separate them with a blank space, a comma, or any character that you choose. It is recommended that you separate successive directives with a comma.

If an alias exists for a table, use the alias (rather than the actual table name) in the optimizer directive specification. Because system-generated index names begin with a blank character, use quotation marks to delimit such names.

Syntax errors in an optimizer directive do not cause a valid query to fail. Use the SET EXPLAIN statement to obtain information related to such errors.

E/C

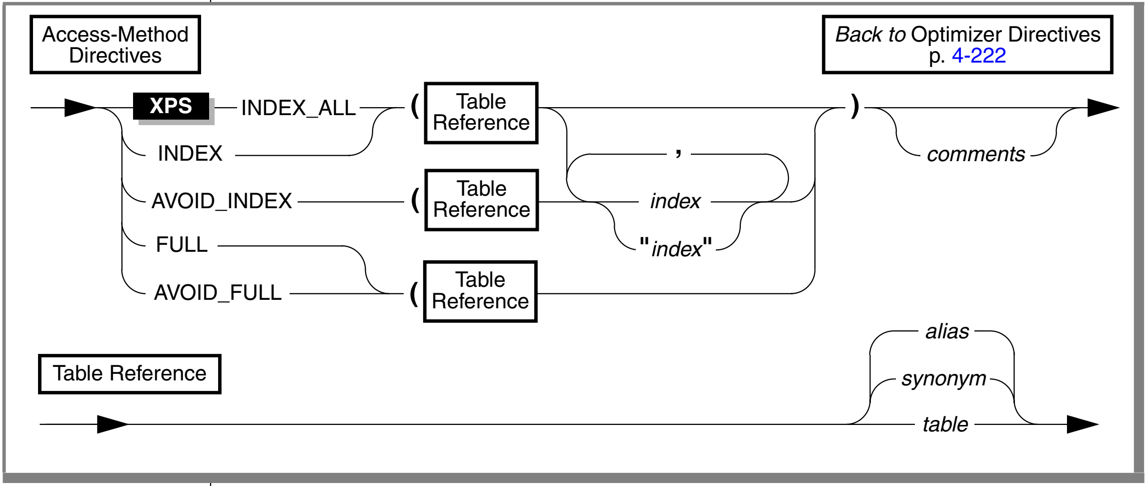
Restrictions on Optimizer Directives

You can specify optimizer directives for any query in a DELETE, SELECT, or UPDATE statement, unless it includes any of the following syntax elements:

- A query that accesses a outside the current database
- In ESQL/C, a statement with the WHERE CURRENT OF *cursor* clause

Access-Method Directives

Use the access-method directive to specify the manner in which the optimizer should search the tables.



Element	Purpose	Restrictions	Syntax
alias	Temporary alternative name declared for a table in the FROM clause	If an *alias* is declared, it must be used in the optimizer directive	Identifier, p. 4-189
comments	Text that documents the optimizer directive	Must be outside the parenthesis but inside the comment symbols	Character string
index	Index for which to specify a query plan directive	Must exist. With AVOID_INDEX, at least one *index* is required	Database Object Name, p. 4-46
synonym, *table*	Synonym or table in a query for which to specify a directive	Synonym and the table to which it points must exist	Database Object Name, p. 4-46

Use commas or blank spaces to separate elements within the parentheses.

The following table describes each of the access-method directives and indicates how it affects the query plan of the optimize.

Keywords	Effect	Optimizer Action
AVOID_FULL	No full-table scan on the listed table	The optimizer considers the various indexes it can scan. If no index exists, the optimizer performs a full-table scan.
AVOID_INDEX	Does not use any of the indexes listed	The optimizer considers the remaining indexes and a full- table scan.If all indexes for a table are specified, optimizer uses a full-table scan to access the table.
FULL	Performs a full-table scan	Even if an index exists on a column, the optimizer uses a full-table scan to access the table.
INDEX	Uses the index specified to access the table	If more than one index is specified, the optimizer chooses the index that yields the least cost. If no indexes are specified, then all the available indexes are considered.
INDEX_ALL	Access the table using all listed indexes (Multi-index scan)	If more than one index is specified, all are used. If only one is specified, optimizer follows an INDEX SKIP SCAN using that index. If no index is specified, then all available indexes are considered. ♦

XPS

Both the AVOID_FULL and INDEX keywords specify that the optimizer should avoid a full scan of a table. It is recommended, however, that you use the AVOID_FULL keyword to specify the intent to avoid a full scan on the table. In addition to specifying that the optimizer not use a full-table scan, the negative directive allows the optimizer to use indexes that are created after the access-method directive is specified.

In general, you can specify only one access-method directive per table. You can, however, specify both AVOID_FULL and AVOID_INDEX for the same table. When you specify both of these access-method directives, the optimizer avoids performing a full scan of the table and it avoids using the specified index or indexes.

This combination of negative directives allows the optimizer to use indexes that are created after the access-method directives are specified.

Suppose that you have a table named **emp** that contains the following indexes: **loc_no**, **dept_no**, and **job_no**. When you perform a SELECT that uses the table in the FROM clause, you might direct the optimizer to access the table in one of the following ways:

- Example using a positive directive:

  ```
  SELECT {+INDEX(emp dept_no)}
  ```

  In this example the access-method directive forces the optimizer to scan the index on the **dept_no** column.

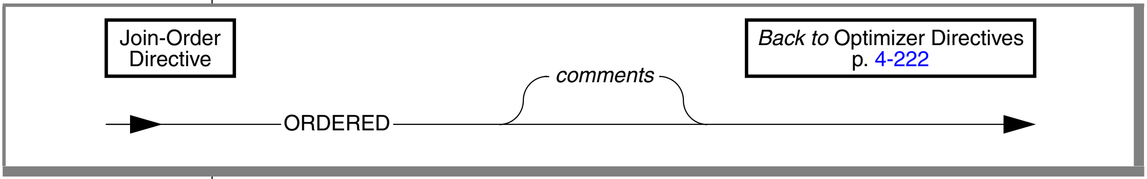
- Example using a negative directive:

  ```
  SELECT {+AVOID_INDEX(emp loc_no, job_no), AVOID_FULL(emp)}
  ```

  This example includes multiple access-method directives. These access-method directives also force the optimizer to scan the index on the **dept_no** column. If a new index, **emp_no** is created for table **emp**, however, the optimizer can consider it.

Join-Order Directive

Use the ORDERED join-order directive to force the optimizer to join tables or views in the order in which they appear in the FROM clause of the query.



Element	Purpose	Restrictions	Syntax
comments	Text that documents the directive	Must appear between comment symbols.	Character string

For example, the following query forces the database server to join the **dept** and **job** tables and then join the result with the **emp** table:

```
SELECT --+ ORDERED
   name, title, salary, dname
FROM dept, job, emp WHERE title = 'clerk' AND loc = 'Palo Alto'
   AND emp.dno = dept.dno
   AND emp.job= job.job
```

Because no predicates occur between the **dept** table and the **job** table, this query forces the database server to construct a Cartesian product.

When your query involves a view, the placement of the ORDERED join-order directive determines whether you are specifying a partial- or total-join order.

- Specifying partial-join order when you create a view

  If you use the ORDERED directive when you create a view, the base tables are joined contiguously in the order of the view definition.

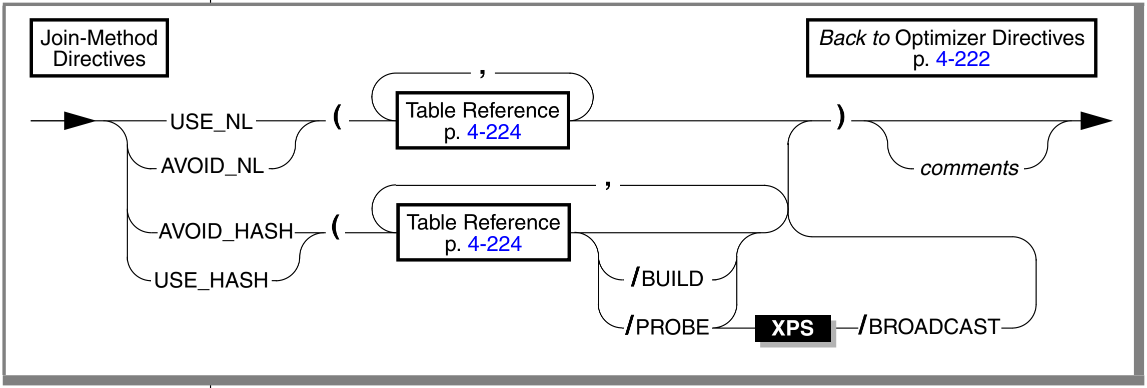
  For all subsequent queries on the view, the database server joins the base tables contiguously in the order specified in the view definition. When used in a view, the ORDERED directive does not affect the join order of other tables named in the FROM clause in a query.
- Specifying total-join order when you query a view

  When you specify the ORDERED join-order directive in a query that uses a view, all tables are joined in the order specified, even those tables that form views. If a view is included in the query, the base tables are joined contiguously in the order of the view definition.

For examples of ORDERED with views, refer to your *Performance Guide*.

Join-Method Directives

Use join-method directives to influence how tables are joined in a query.



Element	Purpose	Restrictions	Syntax
comments	Text that documents the directive	Must appear between comment symbols.	Character string

XPS

This diagram is simplified: /BROADCAST is not valid with AVOID_HASH. ♦

Use commas or blank spaces to separate the elements within the parentheses.

The following table describes each of the join-method directives.

Keyword	Purpose
USE_NL	Uses the specified tables as the inner table in a nested-loop join If *n* tables are specified in the FROM clause, then at most *n-1* tables can be specified in the USE_NL join-method directive.
USE_HASH	Uses a hash join to access the specified table You can also choose whether the table will be used to create the hash table or to probe the hash table.
AVOID_NL	Does not use the specified table as inner table in a nested loop join A table listed with this directive can still participate in a nested loop join as the outer table.
AVOID_HASH	Does not access the specified table using a hash join You can optionally use a hash join, but impose restrictions on the role of the table within the hash join.

A join-method directive takes precedence over the join method forced by the OPTCOMPIND configuration parameter.

When you specify the USE_HASH or AVOID_HASH directives (to use or avoid a hash join, respectively), you can also specify the role of each table:

- /BUILD

  With the USE_HASH directive, this keyword indicates that the specified table be used to construct a hash table. With the AVOID_HASH directive, this keyword indicates that the specified table *not* be used to construct a hash table.

- /PROBE

  With the USE_HASH directive, this keyword indicates that the specified table be used to probe the hash table. With the AVOID_HASH directive, this keyword indicates that the specified table *not* be used to probe the hash table. You can specify multiple probe tables as long as there is at least one table for which you do not specify PROBE.

IDS

- /BROADCAST

  This BROADCAST directive forces a plan which broadcasts the base table or derived table of a specific size. The directive must also include either BUILD or PROBE. If the BUILD option is included, the USE_HASH table is broadcast. If the PROBE option is included, the join in which the USE_HASH table participates is broadcast. ♦

If neither the /BUILD nor the /PROBE keyword is specified, the optimizer uses cost estimates to determine the role of the table.

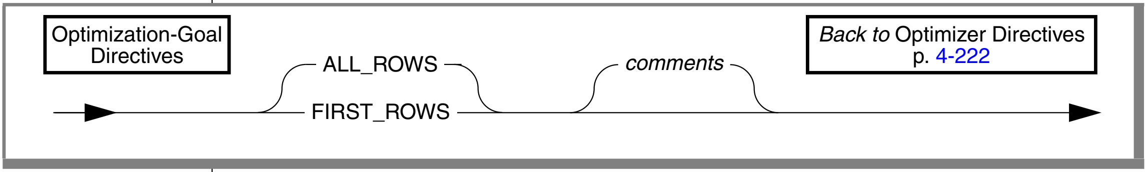
In this example, the USE_HASH directive forces the optimizer to construct a hash table on the **dept** table and consider only the hash table to join **dept** with the other tables. Because no other directives are specified, the optimizer can choose the least expensive join methods for the other joins in the query.

```
SELECT /*+ USE_HASH (dept /BUILD)
   The optimizer must use dept to construct a hash table */
   name, title, salary, dname
   FROM emp, dept, job WHERE loc = 'Phoenix'
      AND emp.dno = dept.dno  AND emp.job = job.job
```

IDS

Optimization-Goal Directives

Use optimization-goal directives to specify the measure that is used to determine the performance of a query result.


Optimization-Goal Directives
ALL_ROWS
FIRST_ROWS
comments
Back to Optimizer Directives p. 4-222

Element	Purpose	Restrictions	Syntax
comments	Text that documents the directive	Must appear between comment symbols.	Character string

The two optimization-goal directives are:

- FIRST_ROWS

  This tells the optimizer to choose a plan that optimizes the process of finding only the first screenful of rows that satisfies the query.
  Use this option to decrease initial response time for queries that use an interactive mode or that require the return of only a few rows.

- ALL_ROWS

  This directive tells the optimizer to choose a plan that optimizes the process of finding all rows that satisfy the query.

  This form of optimization is the default.

An optimization-goal directive takes precedence over the **OPT_GOAL** environment variable and the OPT_GOAL configuration parameter.

For information about how to set the optimization goal for an entire session, see the SET OPTIMIZATION statement.

You cannot use an optimization-goal directive in the following contexts:

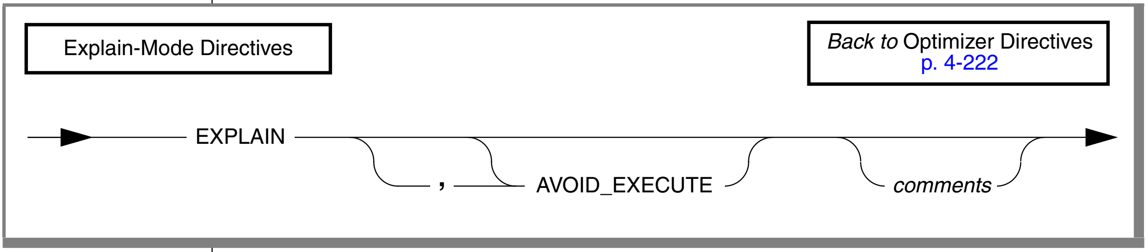
- In a view definition
- In a subquery

The following query returns the names of the employees who earned the top fifty bonuses. The optimization-goal directive directs the optimizer to return the first screenful of rows as fast as possible.

```
SELECT {+FIRST_ROWS
   Return the first screenful of rows as fast as possible}
```

Explain-Mode Directives

Use the explain-mode directives to test and debug query plans and to print information about the query plan to the **sqexplain.out** file.



Element	Purpose	Restrictions	Syntax
comments	Text that documents the directive	Must appear between comment symbols.	Character string

The following table lists the effect of each explain-mode directive.

Keyword	Effect
EXPLAIN	Turns SET EXPLAIN ON for the specified query
AVOID_EXECUTE	Prevents the data manipulation statement from executing; instead, the query plan is printed to the **sqexplain.out** file

The EXPLAIN directive is primarily useful for testing and debugging query plans. It is redundant when SET EXPLAIN ON is already in effect. It is not valid in a view definition or in a subquery.

The next query executes and prints the query plan to the **sqexplain.out** file:

```
SELECT {+EXPLAIN}
   c.customer_num, c.lname, o.order_date
   FROM customer c, orders o WHERE c.customer_num = o.customer_num
```

The AVOID_EXECUTE directive prevents execution of a query on either the local or remote site, if a remote table is part of the query. This directive does not prevent nonvariant functions in a query from being evaluated.

The following query does not return data but writes the query plan to the **sqexplain.out** file:

```
SELECT {+EXPLAIN, AVOID_EXECUTE}
   c.customer_num, c.lname, o.order_date
FROM customer c, orders o WHERE c.customer_num = o.customer_num
```

You must use both the EXPLAIN and AVOID_EXECUTE directives to see the query plan of the optimizer (in the **sqexplain.out** file) without executing the query. The comma (**,**) separating these two directives is optional.

If you omit the EXPLAIN directive when you specify the AVOID_EXECUTE directive, no error is issued, but no query plan is written to the **sqexplain.out** file and no DML statement is executed.

You cannot use the explain-mode directives in the following contexts:

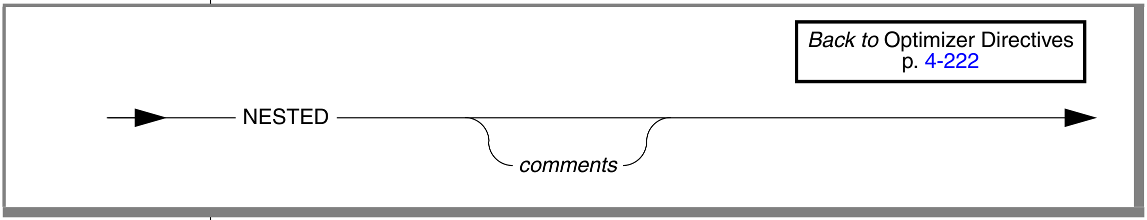
- In a view definition
- In a trigger
- In a subquery

They are valid, however, in a SELECT statement within an INSERT statement.

XPS

Rewrite Method Directive

By default, the database server attempts to unnest correlated subqueries. You can use the NESTED rewrite directive in the outermost query to prevent the query plan from being rewritten with unnested subqueries.



Back to Optimizer Directives
p. 4-222

Element	**Purpose**	**Restrictions**	**Syntax**
comments	Text that documents the directive	Must appear between comment symbols.	Character string

Related Information

For information about the **sqexplain.out** file, see SET EXPLAIN.

For information about how to set optimization settings for an entire session, see SET OPTIMIZATION.

For a discussion about optimizer directives and performance, see your *Performance Guide*.

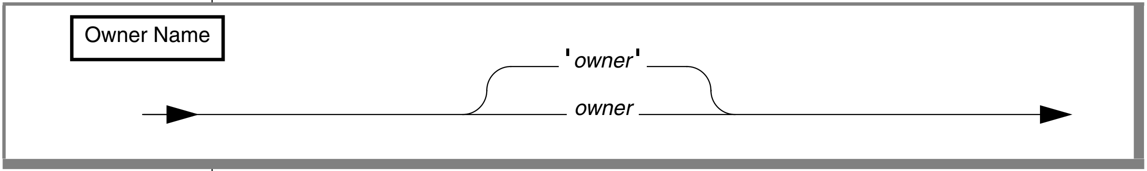
For information on the **IFX_DIRECTIVES** environment variable, see the *IBM Informix Guide to SQL: Reference*.

For information on the DIRECTIVES parameter in the ONCONFIG file, see your *Administrator's Reference*.

Owner Name

The owner name specifies the owner of a database object. Use this segment whenever you see a reference to Owner Name in a syntax diagram. A synonym for owner name is *authorization identifier*.

Syntax



Element	Purpose	Restrictions	Syntax
owner	User name of the owner of a database object in a database	In Dynamic Server, the maximum length of *owner* is 32 bytes. In Extended Parallel Server, the maximum length of *owner* is 8 bytes.	Must conform to the rules of your operating system.

Usage

ANSI

In an ANSI-compliant database, you must specify the *owner* of any database object that you do not own. The ANSI term for *owner name* is *schema name*. ♦

In databases that are not ANSI-compliant, the *owner* name is optional. You do not need to specify *owner* when you create database objects or use data access statements. If you do not specify *owner* when you create a database object, the database server assigns your login name as the owner of the object, in most cases. For exceptions to this rule, see "Ownership of Created Database Objects" on page 2-140 in CREATE FUNCTION and "Ownership of Created Database Objects" on page 2-189 in CREATE PROCEDURE.

If you specify *owner* in data-access statements, the database server checks it for correctness. Without quotation marks, *owner* is case insensitive. The following four queries all can access data from the table **kaths.tab1**:

```
SELECT * FROM tab1
SELECT * FROM kaths.tab1
SELECT * FROM KATHS.tab1
SELECT * FROM Kaths.tab1
```

Using Quotation Marks

When you use quotation marks, *owner* is case sensitive. In other words, quotation marks instruct the database server to read or store the name exactly as typed when you create or access a database object. For example, suppose that you have a table whose owner is `Sam`. You can use either one of the following two statements to access data in the table:

```
SELECT * FROM table1
SELECT * FROM 'Sam'.table1
```

The first query succeeds because the owner name is not required. The second query succeeds because the specified owner name matches the owner name as it is stored in the database.

IDS

In a distributed query, if the owner name is not between quotation marks, the remote database recognizes the case convention of the local database. For example, if the local database is ANSI-compliant, then the remote database processes the owner name in *uppercase*. If the local database is not ANSI-compliant, the remote database processes the owner name in *lowercase*. ♦

Accessing Information from the System Catalog Tables

If you use the owner name as one of the selection criteria to access database object information from one of the system catalog tables, the owner name is case sensitive. Because this type of query requires that you use quotation marks, you must type the owner name exactly as it is stored in the system catalog table. Of the following two examples, only the second successfully accesses information on the table **Kaths.table1**.

```
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'kaths'
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'Kaths'
```

User **informix** is the owner of the system catalog tables.

Tip: *The USER keyword returns the login name exactly as it is stored on the system. If the owner name is stored differently from the login name (for example, a mixed-case owner name and an all lowercase login name), the* `owner = USER` *syntax fails.*

ANSI-Compliant Database Restrictions and Case Sensitivity

ANSI

If you specify the owner name when you create or rename a database object in an ANSI-compliant database, you must include the owner name in data access statements. You must include the owner name when you access a database object that you do not own. ♦

The following table describes how the database server reads and stores *owner* when you create, rename, or access a database object.

Owner Name Specification	What the Database Server Does
Do not specify	Reads or stores owner exactly as the login name is stored in the system. Users must specify *owner* for a database object or database they do not own.
Specify without quotation marks	Reads or stores *owner* in uppercase letters
Enclose within quotation marks	Reads or stores *owner* exactly as entered. For more information on how the database server handles this notation, see "Using Quotation Marks" on page 4-235 and "Accessing Information from the System Catalog Tables" on page 4-235.

Because the database server automatically shifts *owner* to uppercase letters if not between quotation marks, case-sensitive errors can cause queries to fail. For example, if you are user **nancy** and you use the following statement, the resulting view has the name **nancy.njcust**:

```
CREATE VIEW 'nancy'.njcust AS
   SELECT fname, lname FROM customer WHERE state = 'NJ'
```

The following SELECT statement fails because it tries to match the name **NANCY.njcust** to the actual owner and table name of **nancy.njcust**:

```
SELECT * FROM nancy.njcust
```

Tip: *When you use the owner name as one of the selection criteria in a query (for example,* `WHERE owner = 'kaths'`*), make sure that the quoted string matches the owner name as it is stored in the database. If the database server cannot find the database object or database, you might need to modify the query so that the quoted string uses uppercase letters (for example,* `WHERE owner = 'KATHS'`*).*

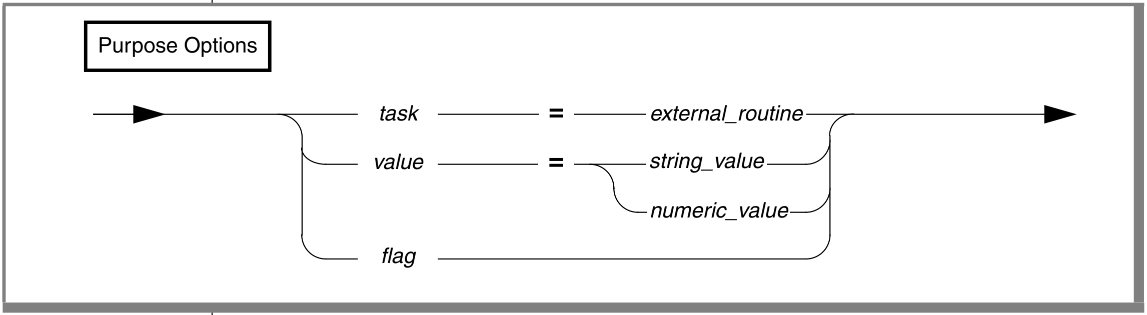
+

IDS

Purpose Options

The CREATE ACCESS_METHOD and ALTER ACCESS_METHOD statements specify purpose options with the following syntax.

Syntax



Element	Meaning	Restrictions	Syntax
task	A keyword that identifies a purpose function	Keywords to which you can assign a function (whose name cannot match the keyword)	*Task* Purpose Category in the table on p. 4-239.
value	A keyword that identifies configuration information	Predefined configuration keywords to which you can assign values	*Value* Purpose Category in the table on p. 4-239.
flag	A keyword that indicates which feature a flag enables	The interface specifies flag names	*Flag* Purpose Category in the table on p. 4-239.
external _routine	The user-defined function that performs a *task*	Must be registered in the database by a CREATE FUNCTION statement	Database Object Name, p. 4-46
string _value	A value that is expressed as one or more characters	Characters must be from the code set of the database	Quoted String, p. 4-243
numeric _value	A value of a real number	Must be within the range of a numeric data type	Literal Number, p. 4-216

Usage

The database server recognizes a registered access method as a set of attributes, including a name and options called *purpose options*.

You can use purpose options to accomplish the following tasks:

- Specify which functions perform data access and manipulation tasks, such as opening, reading, and closing a data source.
- Set configuration options, such as a storage-space type.
- Set flags, such as enabling rowid interpretation.

You specify purpose options when you create an access method with the CREATE ACCESS_METHOD statement. To change the purpose options of an access method, use the ALTER ACCESS_METHOD statement.

Each *task*, *value*, or *flag* keyword corresponds to a column name in the **sysams** system catalog table. The keywords let you set the following attributes:

- Purpose function

  A *purpose-function attribute* maps the name of a user-defined function or method to a *task* keyword, such as **am_create**, **am_beginscan**, or **am_getnext**. For a complete list of these keywords, see the "Task" category in the table on page 4-239.
  The *external_routine* specifies the corresponding function (C) that you supply for the access method. Example setting:

  ```
  am_create = FS_create
  ```

- Purpose flag

  A *purpose flag* indicates whether an access method supports a given SQL statement or keyword. Example setting:

  ```
  am_rowids
  ```

- Purpose value

  These string, character, or numeric values provide configuration information that a flag cannot supply. Example setting:

  ```
  am_sptype = 'X'
  ```

To enable a user-defined function or method as a purpose function, you must first register the C function or Java method that performs the appropriate tasks, using the CREATE FUNCTION statement, and then set the purpose keyword equal to the registered function or method name. This example creates a new access method. Another example on page 2-15 adds a purpose method to an existing access method.

To enable a purpose flag, specify the name without a corresponding value.

To clear a purpose-option setting in the **sysams** table, use the DROP clause of the ALTER ACCESS_METHOD statement.

Purpose Functions, Methods, Flags, and Values

The following table describes the possible settings for the **sysams** columns that contain purpose functions or methods, flags, and values. The entries appear in the same order as the corresponding **sysams** columns.

Keyword	Explanation	Category	Default
am_sptype	A character that specifies from what type of storage space a primary or secondary-access method can access data. The **am_sptype** character can have any of the following settings: ■ `'X'` indicates the method accesses only extspaces. ■ `'S'` indicates the method accesses only sbspaces. ■ `'A'` indicates the method can access extspaces and sbspaces. Valid only for a new access method. You cannot change or add an **am_sptype** value with ALTER ACCESS_METHOD. Do not set **am_sptype** to `'D'` or attempt to store a virtual table in a dbspace.	Value	Virtual-Table Interface (C): `'A'`
am_defopclass	The default operator class for a secondary-access method. The access method must exist before you can define its operator class, so you set this value in the ALTER ACCESS_METHOD statement.	Value	None
am_keyscan	A flag that, if set, indicates that **am_getnext** returns rows of index keys for a secondary-access method. If a query selects only the columns in the index key, the database server uses the row of index keys that the secondary-access method puts in shared memory, without reading the table.	Flag	Not set
am_unique	A flag to set if a secondary-access method checks for unique keys	Flag	Not set

(1 of 3)

Keyword	Explanation	Category	Default
am_cluster	A flag that you set if a primary- or secondary-access method supports clustering of tables	Flag	Not set
am_rowids	A flag that you set if a primary-access method can retrieve a row from a specified address	Flag	Not set
am_readwrite	A flag to set if a primary-access method supports data changes. The default setting, not set, indicates that the virtual data is read-only. For the C Virtual-Table Interface, set this flag if your application will write data, to avoid the following problems: ■ An INSERT, DELETE, UPDATE, or ALTER FRAGMENT statement causes an SQL error. ■ Function **am_insert**, **am_delete**, or **am_update** is not executed.	Flag	Not set
am_parallel	A flag that the database server sets to indicate which purpose functions or methods can execute in parallel in a primary or secondary-access method. If set, the hexadecimal **am_parallel** bitmap contains one or more of the following bit settings: ■ The `1` bit is set for parallelizable scan. ■ The `2` bit is set for parallelizable delete. ■ The `4` bit is set for parallelizable update. ■ The `8` bit is set for parallelizable insert. Insertions, deletions, and updates are not supported in the Java Virtual-Table Interface.	Flag	Not set
am_costfactor	A value by which the database server multiplies the cost that the am_scancost purpose function or method returns for a primary or secondary-access method. An **am_costfactor** value from `0.1` to `0.9` reduces the cost to a fraction of the value that **am_scancost** calculates. An **am_costfactor** value of `1.1` or greater increases the **am_scancost** value.	Value	1.0
am_create	A keyword that you associate with a user-defined function or method (UDR) name that creates a virtual table or virtual index	Task	None
am_drop	A keyword that you associate with the name of a UDR that drops a virtual table or virtual index	Task	None
am_open	A keyword that you associate with the name of a UDR that makes a fragment, extspace, or sbspace available	Task	None

(2 of 3)

Keyword	Explanation	Category	Default
am_close	A keyword that you associate with the name of a UDR that reverses the initialization that **am_open** performs	Task	None
am_insert	A keyword that you associate with the name of a UDR that inserts a row or an index entry	Task	None
am_delete	A keyword that you associate with the name of a UDR that deletes a row or an index entry	Task	None
am_update	A keyword that you associate with the name of a UDR that changes the values in a row or key	Task	None
am_stats	A keyword that you associate with the name of a UDR that builds statistics based on the distribution of values in storage spaces	Task	None
am_scancost	A keyword that you associate with the name of a UDR that calculates the cost of qualifying and retrieving data	Task	None
am_check	A keyword that you associate with the name of a UDR that tests the physical structure of a table or performs an integrity check on an index	Task	None
am_beginscan	A keyword that you associate with the name of a UDR that sets up a scan	Task	None
am_endscan	A keyword that you associate with the name of a UDR that reverses the setup that **am_beginscan** initializes	Task	None
am_rescan	A keyword that you associate with the name of a UDR that scans for the next item from a previous scan to complete a join or subquery	Task	None
am_getnext	A keyword that you associate with the name of the required UDR that scans for the next item that satisfies a query	Task	None
am_getbyid	A keyword that you associate with the name of a UDR that fetches data from a specific physical address; **am_getbyid** is available only for primary-access methods	Task	None
am_truncate	A keyword that you associate with the name of a UDR that deletes all rows of a virtual table (primary-access method) or that deletes all corresponding keys in a virtual index (secondary- access method)	Task	None

(3 of 3)

The following rules apply to the purpose-option specifications in the CREATE ACCESS_METHOD and ALTER ACCESS_METHOD statements:

- To specify multiple purpose options in one statement, separate them with commas.
- The CREATE ACCESS_METHOD statement must specify a user-defined function or method name that corresponds to the **am_getnext** keyword.

  The ALTER ACCESS_METHOD statement cannot drop the function or method name that corresponds to **am_getnext** but can modify it.
- The ALTER ACCESS_METHOD statement cannot add, drop, or modify the **am_sptype** value.
- You can specify the **am_defopclass** value only with the ALTER ACCESS_METHOD statement.

  You must first register a secondary-access method with the CREATE ACCESS_METHOD statement before you can assign a default operator class.

Related Information

Related statements: CREATE FUNCTION and CREATE OPCLASS

For the following topics, see the *IBM Informix Virtual-Table Interface Programmer's Guide* (for C):

- Managing storage spaces, executing in parallel, and calculating statement costs
- Registering the access method and purpose functions
- Purpose-function reference

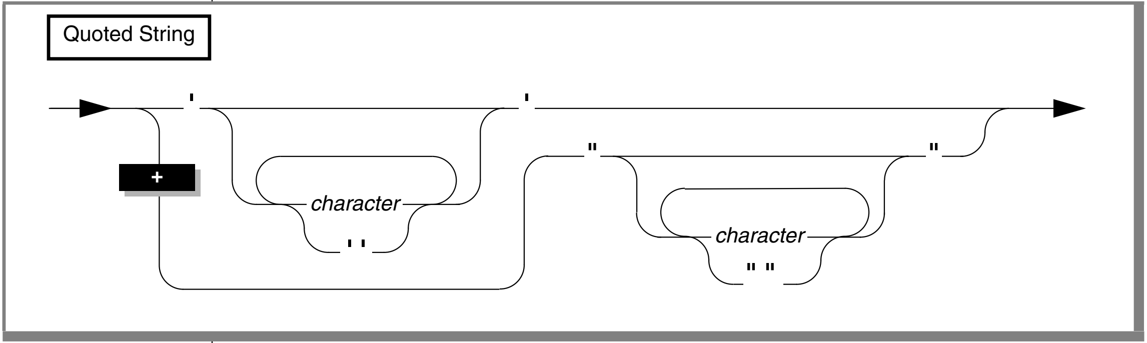
For the following topics, see the *IBM Informix Virtual-Index Interface Programmer's Guide* (for C):

- Managing storage spaces, executing in parallel, calculating statement costs, bypassing table scans, and enforcing unique-index constraints
- Registering the access method and purpose functions
- Purpose-function reference

Quoted String

A quoted string is a string literal between quotation marks. Use this segment whenever you see a reference to a quoted string in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
character	Code set element within quoted string	Cannot enclose between double quotes if the **DELIMIDENT** environment variable is set.	Literal value from the keyboard.

Usage

Use quoted strings to specify string literals in data-manipulation statements and other SQL statements. For example, you can use a quoted string in an INSERT statement to insert a value into a column of a character data type.

Restrictions on Specifying Characters in Quoted Strings

You must observe the following restrictions on *character* in quoted strings:

- If you are using the ASCII code set, you can specify any printable ASCII character, including a single quote or double quote. For restrictions that apply to using quotes in quoted strings, see "Using Quotes in Strings" on page 4-245.
- GLS In some locales, you can specify non-ASCII characters, including multibyte characters, that the locale supports. See the discussion of quoted strings in the *IBM Informix GLS User's Guide*. ♦
- If you enable newline characters for quoted strings, you can embed newline characters in quoted strings. For further information, see "Newline Characters in Quoted Strings" on page 4-245.
- After you set the **DELIMIDENT** environment variable, you cannot use double quotes (**"**) to delimit quoted strings. If **DELIMIDENT** is set, the database server interprets strings enclosed in double quotes as SQL identifiers, not quoted strings. If **DELIMIDENT** is not set, a string between double quotes is a quoted string, not an identifier. For further information, see "Using Quotes in Strings" on page 4-245.
- You can enter DATETIME and INTERVAL data values as quoted strings. For the restrictions that apply to entering DATETIME and INTERVAL data in quoted-string format, see "DATETIME and INTERVAL Values as Strings" on page 4-246.
- Quoted strings that are used with the LIKE or MATCHES keyword in a search condition can include wildcard characters that have a special meaning in the search condition. For further information, see "LIKE and MATCHES in a Condition" on page 4-246.
- When you insert a value that is a quoted string, you must observe a number of restrictions. For further information, see "Inserting Values as Quoted Strings" on page 4-246.

Newline Characters in Quoted Strings

By default, the string constant must be written on a single line. That is, you cannot use embedded newline characters in a quoted string. You can, however, override this default behavior in one of two ways:

- To enable newline characters in quoted strings in all sessions, set the ALLOW_NEWLINE parameter to 1 in the **ONCONFIG** file.
- To enable newline characters in quoted strings for the current session, execute the built-in function **IFX_ALLOW_NEWLINE**.

In the following example, the user enables newline characters in quoted strings for the current session:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('T')
```

If newline characters in quoted strings are not enabled for a session, the following statement is illegal and results in an error:

```
SELECT 'The quick brown fox
   jumped over the old gray fence'
   FROM customer
   WHERE customer_num = 101
```

If you enable newline characters in quoted strings for the session, however, the statement in the preceding example is valid and executes successfully.

For more information on the **IFX_ALLOW_NEWLINE** function, see "IFX_ALLOW_NEWLINE Function" on page 4-164. For more information on the ALLOW_NEWLINE parameter in the **ONCONFIG** file, see your *Administrator's Reference*.

Using Quotes in Strings

The single quote (') has no special significance in string literals delimited by double quotes. Conversely, double quote (") has no special significance in strings delimited by single quotes. For example, these strings are valid:

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "No!"'
```

A string delimited by double quotes can include a double quote character by preceding it with another double quote, as the following string shows:

```
"Enter ""y"" to select this row"
```

When the **DELIMIDENT** environment variable is set, double quotes can only delimit identifiers, not strings. For more information on delimited identifiers, see "Delimited Identifiers" on page 4-191.

DATETIME and INTERVAL Values as Strings

You can enter DATETIME and INTERVAL data in the literal forms described in the "Literal DATETIME" on page 4-212 and "Literal INTERVAL" on page 4-214, or you can enter them as quoted strings.

Valid literals that are entered as character strings are converted automatically into DATETIME or INTERVAL values.

These statements enter INTERVAL and DATETIME values as quoted strings:

```
INSERT INTO cust_calls(call_dtime) VALUES ('1997-5-4 10:12:11')
INSERT INTO manufact(lead_time) VALUES ('14')
```

The format of the value in the quoted string must exactly match the format specified by the INTERVAL or DATETIME qualifiers of the column. For the first INSERT in the preceding example, the **call_dtime** column must be defined with the qualifiers YEAR TO SECOND for the INSERT statement to be valid.

LIKE and MATCHES in a Condition

Quoted strings with the LIKE or MATCHES keyword in a condition can include wildcard characters. For a complete description of how to use wildcard characters, see "Condition" on page 4-24.

Inserting Values as Quoted Strings

In the default locale, if you are inserting a value that is a quoted string, you must adhere to the following restrictions:

- Enclose CHAR, VARCHAR, NCHAR, NVARCHAR, DATE, DATETIME, INTERVAL, and (for Dynamic Server) LVARCHAR values in quotation marks.
- Specify DATE values in the *mm/dd/yyyy* format (or in the format that the **DBDATE** or **GL_DATE** environment variable specifies, if set).

XPS

- You cannot insert strings longer than 256 bytes. ♦

IDS

- You cannot insert strings longer than 32 kilobytes. ♦

- Numbers with decimal values must include a decimal separator. Comma (,) is not valid as a decimal separator in the default locale.
- MONEY values cannot include a dollar sign ($) or commas.
- You can enter NULL in a column only if it accepts NULL values.

Related Information

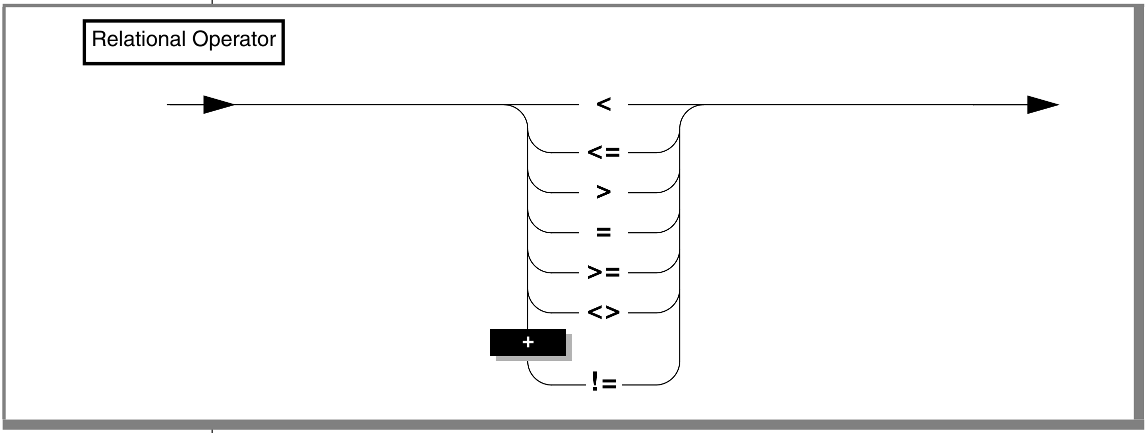
For a discussion of the DELIMIDENT environment variable, see the *IBM Informix Guide to SQL: Reference*.

For a discussion of the GLS aspects of quoted strings, see the *IBM Informix GLS User's Guide*.

Relational Operator

A relational operator compares two expressions quantitatively. Use the Relational Operator segment whenever you see a reference to a relational operator in a syntax diagram.

Syntax



Usage

The relational operators of SQL have the following meanings.

Relational Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
=	Equal to
>=	Greater than or equal to
<>	Not equal to
!=	Not equal to

Usage

For number expressions, *greater than* means to the right on the real line.

For DATE and DATETIME expressions, *greater than* means later in time.

For INTERVAL expressions, *greater than* means a longer span of time.

For CHAR, VARCHAR, and LVARCHAR expressions, *greater than* means *after* in code-set order. (For NCHAR and NVARCHAR expressions, *greater than* means *after* in the localized collation order, if one exists; otherwise, it means in code-set order.)

GLS

Locale-based collation order is used for NCHAR and NVARCHAR expressions. So for NCHAR and NVARCHAR expressions, *greater than* means *after* in the locale-based collation order. For more information on locale-based collation order and the NCHAR and NVARCHAR data types, see the *IBM Informix GLS User's Guide*. ♦

Using Operator Functions in Place of Relational Operators

Each relational operator is bound to a particular operator function, as the table shows. The operator function accepts two values and returns a boolean value of true, false, or unknown.

Relational Operator	Associated Operator Function
<	**lessthan( )**
<=	**lessthanorequal( )**
>	**greater than( )**
>=	**greaterthanorequal( )**
=	**equal( )**
<>	**notequal( )**
!=	**notequal( )**

Connecting two expressions with a relational operator is equivalent to invoking the operator function on the expressions. For example, the next two statements both select orders with a shipping charge of $18.00 or more.

The >= operator in the first statement implicitly invokes the **greaterthanorequal( )** operator function.

```
SELECT order_num FROM orders
   WHERE ship_charge >= 18.00

SELECT order_num FROM orders
   WHERE greaterthanorequal(ship_charge, 18.00)
```

The database server provides the operator functions associated with the relational operators for all built-in data types. When you develop a user-defined data type, you must define the operator functions for that type for users to be able to use the relational operator on the type.

If you define **less_than( )**, **greater_than( )**, and the other operator functions for a user-defined type, then you should also define **compare( )**. Similarly, if you define **compare( )**, then you should also define **less_than( )**, **greater_than( )**, and the other operator functions. All of these functions must be defined in a consistent manner, to avoid the possibility of incorrect query results when UDT values are compared in the WHERE clause of a SELECT.

Collating Order for U.S. English Data

If you are using the default locale (U.S. English), the database server uses the code-set order of the default code set when it compares the character expressions that precede and follow the relational operator.

UNIX

On UNIX, the default code set is the ISO8859-1 code set, which consists of the following sets of characters:

- The ASCII characters have code points in the range of 0 to 127.

  This range contains control characters, punctuation symbols, English-language characters, and numerals.
- The 8-bit characters have code points in the range 128 to 255.

  This range includes many non-English-language characters (such as é, â, ö, and ñ) and symbols (such as £, ©, and ¿). ♦

Windows

In Windows, the default code set is Microsoft 1252. This code set includes both the ASCII code set and a set of 8-bit characters. ♦

This table lists the ASCII code set. The **Num** columns show ASCII code point numbers, and the **Char** columns display corresponding ASCII characters. In the default locale, ASCII characters are sorted according to their codeset order. Thus, lowercase letters follow uppercase letters, and both follow digits. In this table, ASCII 32 is the blank character, and the caret symbol (^) stands for the CTRL key. For example, ^X means CONTROL-X.

Num	Char	Num	Char	Num	Char	Num	Char	Num	Char	Num	Char	Num	Char
0	^@	20	^T	40	(	60	<	80	P	100	d	120	x
1	^A	21	^U	41	)	61	=	81	Q	101	e	121	y
2	^B	22	^V	42	*	62	>	82	R	102	f	122	z
3	^C	23	^W	43	+	63	?	83	S	103	g	123	{
4	^D	24	^X	44	,	64	@	84	T	104	h	124	|
5	^E	25	^Y	45	-	65	A	85	U	105	i	125	}
6	^F	26	^Z	46	.	66	B	86	V	106	j	126	~
7	^G	27	esc	47	/	67	C	87	W	107	k	127	del
8	^H	28	^\	48	0	68	D	88	X	108	l		
9	^I	29	^]	49	1	69	E	89	Y	109	m		
10	^J	30	^^	50	2	70	F	90	Z	110	n		
11	^K	31	^_	51	3	71	G	91	[	111	o		
12	^L	32		52	4	72	H	92	\	112	p		
13	^M	33	!	53	5	73	I	93	]	113	q		
14	^N	34	"	54	6	74	J	94	^	114	r		
15	^O	35	#	55	7	75	K	95	_	115	s		
16	^P	36	$	56	8	76	L	96	`	116	t		
17	^Q	37	%	57	9	77	M	97	a	117	u		
18	^R	38	&	58	:	78	N	98	b	118	v		
19	^S	39	'	59	;	79	O	99	c	119	w		

GLS

Support for ASCII Characters in Nondefault Code Sets

Most code sets for nondefault locales (called *nondefault code sets*) support the ASCII characters. If you are using a nondefault locale, the database server uses ASCII code-set order for any ASCII data in CHAR and VARCHAR expressions, if the code set supports these ASCII characters. If the current collation (as specified by **DB_LOCALE** or by SET COLLATION) supports a localized collating order, however, that localized order is used when the database server sorts NCHAR or NVARCHAR values.

Literal Numbers as Operands

You might obtain unexpected results if a literal number that you specify as an operand is not in a format that can exactly represent the data type of another value with which it is compared by a relational operator. Because of rounding errors, for example, a relational operator like = or the **equals( )** operator function generally cannot return TRUE if one operand returns a FLOAT value and the other an INTEGER. For information about which of the built-in data types store values that can be exactly represented as literal numbers, see the section "Literal Number" on page 4-216.

Related Information

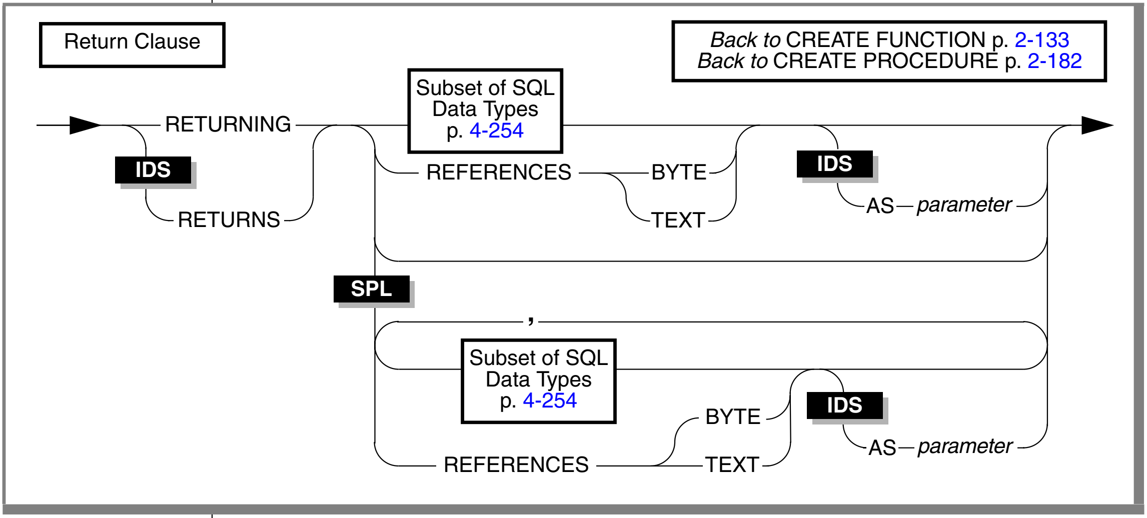
For a discussion of relational operators in the SELECT statement, see the *IBM Informix Guide to SQL: Tutorial*.

For a discussion of the GLS aspects of relational operators, see the *IBM Informix GLS User's Guide*.

Return Clause

The Return clause specifies the data type of a value or values that a user-defined function returns. You can use this segment in UDR definitions.

Syntax


Return Clause
Back to CREATE FUNCTION p. 2-133
Back to CREATE PROCEDURE p. 2-182
RETURNING
IDS
RETURNS
Subset of SQL Data Types p. 4-254
REFERENCES
BYTE
TEXT
IDS
AS
parameter
SPL
,
Subset of SQL Data Types p. 4-254
BYTE
TEXT
REFERENCES
IDS
AS
parameter

Element	Purpose	Restrictions	Syntax
parameter	Name that you declare here for a returned parameter of the UDR	Must be unique among returned *parameters* of UDRs. If any returned value of the UDR has a name, then all must have names.	Identifier, p. 4-189

Usage

IDS

For backward compatibility, you can continue to create SPL functions with the CREATE PROCEDURE statement (that is, include a Return clause in the CREATE PROCEDURE statement). You should use CREATE FUNCTION, however, to create new SPL routines that return one or more values. ♦

After the Return clause has indicated what data types are to be returned, you can use the RETURN statement of SPL at any point in the statement block to return SPL variables that correspond to the values in the Return clause.

Limits on Returned Values

SPL

An SPL function can specify more than one data type in the Return clause. ♦

Ext

An external function can specify only one data type in the Return clause, but an external function can return more than one row of data if it is an iterator function. For more information, see "ITERATOR" on page 4-260. ♦

Subset of SQL Data Types

Not all data types are valid in a Return clause. For more information, see the table that follows. See also "Data Type" on page 4-49.

XPS

A user-defined function can return values of any built-in data type *except* SERIAL, TEXT, or BYTE. ♦

IDS

A UDF can return values of any built-in data type *except* the complex, serial, and large object data types that are not blank in the following table:

Data Type	C	Java	SPL	Data Type	C	Java	SPL
BLOB	✔		✔	ROW		✔	✔
BYTE	✔	✔	✔	SET		✔	✔
COLLECTION		✔		SERIAL	✔	✔	✔
CLOB	✔		✔	SERIAL8	✔	✔	✔
LIST		✔		TEXT	✔	✔	✔
MULTISET		✔					

If you use a complex data type in the Return clause, the calling user-defined routine must define variables of the appropriate complex types to hold the values that the C or SPL user-defined function returns.

User-defined functions can return a value of OPAQUE or DISTINCT data types that are defined in the database. ♦

SPL

The default precision of a DECIMAL that an SPL function returns is 16 digits. For a function to return a DECIMAL with a different number of significant digits, you must specify the precision explicitly in the Return clause. ♦

Using the REFERENCES Clause to Point to a Simple Large Object

A user-defined function cannot return a BYTE or TEXT value (collectively called *simple large objects*) directly. A user-defined function can, however, use the REFERENCES keyword to return a descriptor that contains a pointer to a BYTE or TEXT object.

The following example shows how to select a TEXT column within an SPL routine and then return the value:

```
CREATE FUNCTION sel_text()
   RETURNING REFERENCES text;
   DEFINE blob_var REFERENCES text;
   SELECT blob_col INTO blob_var
      FROM blob_table
      WHERE key-col = 10;
   RETURN blob_var
END FUNCTION
```

XPS

In Extended Parallel Server, to re-create this example, use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

IDS

Named Return Parameters

You can declare names for the returned parameters of an SPL routine, or a name for the single value that an external function can return.

If an SPL routine returns more than one value, you must either declare names for all of the returned parameters, or else none of them can have names. The names must be unique. Here is an example of named parameters:

```
CREATE PROCEDURE p (inval INT DEFAULT 0)
RETURNING INT AS serial_num,
   CHAR(10) AS name,
   INT AS points;
RETURN (inval + 1002), "Newton", 100;
END PROCEDURE;
```

Executing this UDR would return:

```
serial_num    name     points
1002          Newton    100
```

There is no relationship between the names of return parameters and the names of any variables in the body of the function or procedure. For example, you can define a function to return an INTEGER AS **xval**, but in the body of the same function, a variable called **xval** could be of the data type INTERVAL YEAR TO MONTH.

Cursor and Noncursor Functions

A *cursor* function allows the fetching of the returned values one by one by iterating the generated result set of returned values. Such a function is an implicitly iterated function.

A function that returns only one set of values (such as one or more columns from a single row of a table) is a *noncursor* function.

The Return clause can occur in a cursor function or in a noncursor function. In the following example, the Return clause can return zero (0) or one value if it occurs in a noncursor function. If this clause is associated with a cursor function, however, it returns more than one row from a table, and each returned row contains zero or one value.

```
RETURNING INT;
```

In the following example, the Return clause can return zero (0) or two values if it occurs in a noncursor function. If this clause is associated with a cursor function, however, it returns more than one row from a table, and each returned row contains zero or two values.

```
RETURNING INT, INT;
```

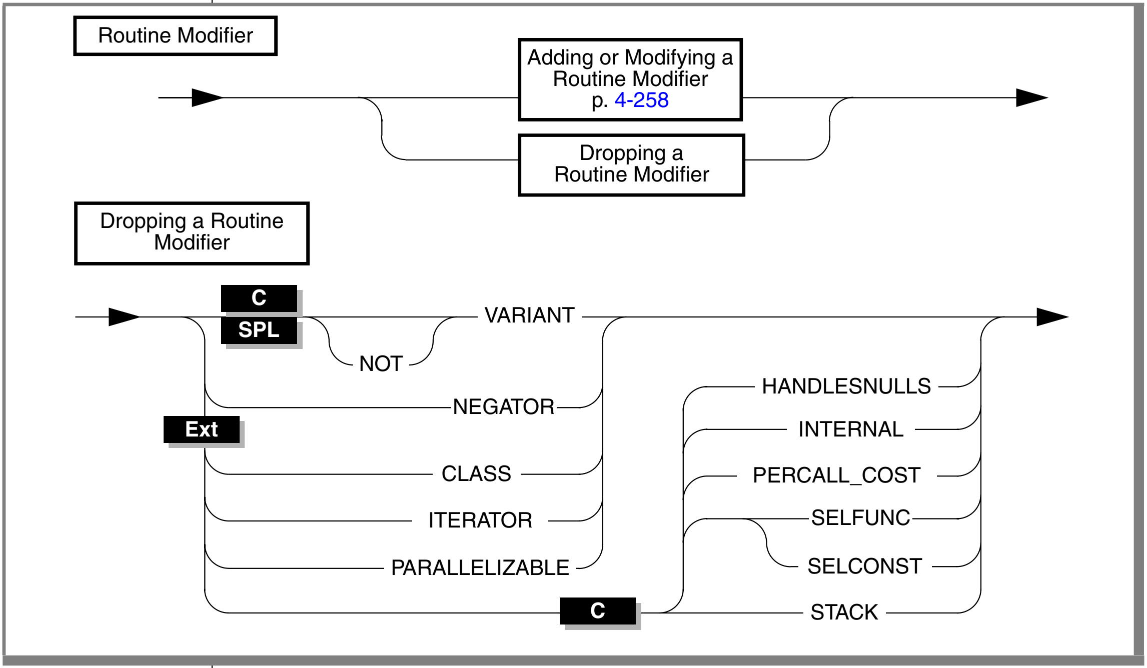
In both of the preceding examples, the receiving function or program must be written appropriately to accept the information that the function returns.

IDS

Routine Modifier

A routine modifier specifies characteristics of how a user-defined routine (UDR) behaves.

Syntax



Usage

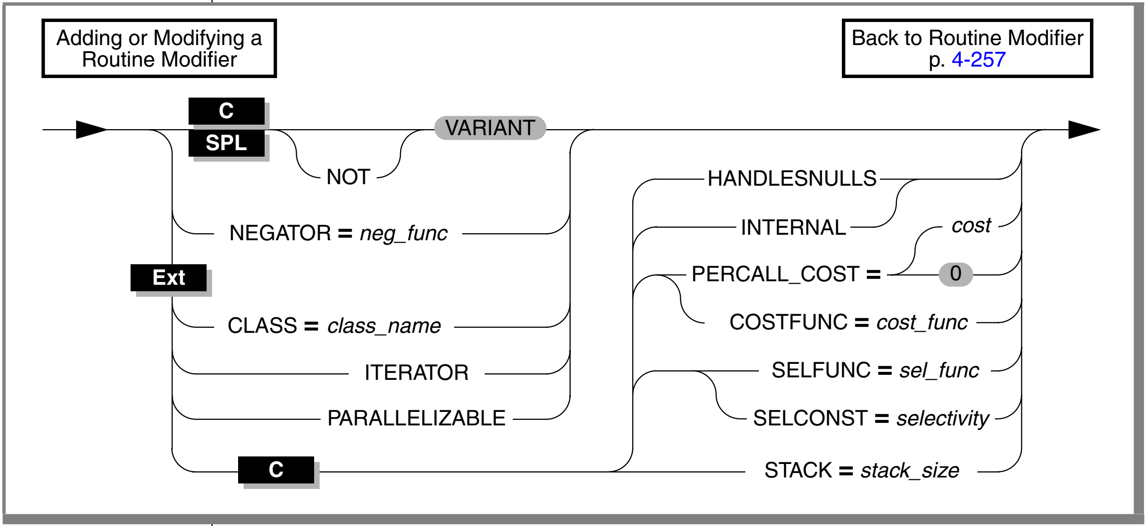
When you drop an existing modifier, the database server sets the value of the modifier to the default value, if a default exists.

Some modifiers are available only with user-defined functions. For information on whether a specific routine modifier applies only to user-defined functions (that is, if it does not apply to user-defined procedures), see the description of the modifier in the sections that follow.

The options in this segment are not valid for SPL procedures.

Adding or Modifying a Routine Modifier

Use this segment to add or modify values for routine modifiers of a UDR.



Element	Purpose	Restrictions	Syntax
class_name	Virtual processor (VP) class in which to run the external routine	Any C UDR must run in the CPU VP or in a user-defined VP class.	Quoted String, p. 4-243
cost	CPU use cost for each invocation of a C language UDR. Default is 0.	Integer; 1 ≤ *cost* ≤ 2^{31}-1 (highest cost).	Literal Number, p. 4-216
cost_func	Name of a companion user-defined cost function to invoke	Must have same owner as the UDR. Execute privilege needed to invoke.	Identifier, p. 4-189
neg_func	Negator function that can be invoked instead of the UDR	Must have same owner as the UDR. Execute privilege needed to invoke.	Identifier, p. 4-189
sel_func	Name of a companion user-defined selectivity function to invoke	Must have same owner as the UDR. Execute privilege needed to invoke.	Identifier, p. 4-189
selectivity	CPU use cost for each invocation of a C language UDR. Default is 0 .	See "Concept of Selectivity" on page 4-263.	Literal Number, p. 4-216
stack_size	Size (in bytes) of stack of the thread that executes the C-language UDR	Must be a positive integer.	Literal Number, p. 4-216

You can add these modifiers in any order. If you list the same modifier more than once, the last setting overrides any previous values.

Modifier Descriptions

The following sections describe the modifiers that you can use to help the database server optimally execute a UDR.

Ext

CLASS

Use the CLASS modifier to specify the name of a virtual-processor (VP) class in which to run an external routine. A user-defined VP class must be defined before the UDR can be invoked.

C

You can execute C UDRs in the following types of VP classes:

- The CPU virtual-processor class (CPU VP)
- A user-defined virtual-processor class

If you omit the CLASS modifier to specify a VP class, the UDR runs in the CPU VP. User-defined VP classes protect the database server from ill-behaved C UDRs. An ill-behaved C UDR has at least one of the following characteristics:

- It runs in the CPU VP for a long time without yielding.
- It is not thread safe.
- It calls an unsafe operating-system routine.

A well-behaved C UDR has none of these characteristics. Execute only well-behaved C UDRs in the CPU VP.

Warning: *Execution of an ill-behaved C UDR in the CPU VP can cause serious interference with the operation of the database server. In addition, the UDR itself might not produce correct results. For a more detailed discussion of ill-behaved UDRs, see the IBM Informix DataBlade API Programmer's Guide.* ♦

Java

By default, a UDR written in Java runs in a Java virtual processor class (JVP). Therefore, the CLASS modifier is optional for a UDR written in Java. However, use the CLASS modifier when you register a UDR written in Java to improve readability of your SQL statements. ♦

C

COSTFUNC

Use the COSTFUNC modifier to specify the cost of a C UDR. The *cost* of the UDR is an estimate of the time required to execute it.

Occasionally, the cost of a UDR depends on its inputs. In that case, you can use a user-defined function to calculate a cost that depends on input values.

To execute *cost_func*, you must have Execute privilege on it and on the UDR.

HANDLESNULLS

C

Use the HANDLESNULLS modifier to specify that a C UDR can handle NULL values that are passed to it as arguments. If you do not specify HANDLESNULLS for a C language UDR, and if you pass an argument with a NULL value to it, the UDR does not execute, and returns a NULL value.

By default, a C language UDR does *not* handle NULL values. ♦

SPL

The HANDLESNULLS modifier is not available for SPL routines because SPL routines handle NULL values by default. ♦

Ext

INTERNAL

Use the INTERNAL modifier with an external routine to specify that an SQL or SPL statement cannot call the external routine. An external routine that is specified as INTERNAL is not considered during routine resolution. Use the INTERNAL modifier for external routines that define access methods, language managers, and so on.

By default, an external routine is *not* internal; that is, an SQL or SPL statement can call the routine.

Ext

ITERATOR

Use the ITERATOR modifier with external functions to specify that the function is an *iterator function*. An iterator function is a function that returns a single element per function call to return a set of data; that is, it is called with an initial call and zero or more subsequent calls until the set is complete.

By default, an external C or Java language function is *not* an iterator function.

SPL

An SPL iterator function requires the RETURN WITH RESUME statement, rather than the ITERATOR modifier. ♦

E/C

An iterator function requires a cursor. The cursor allows the client application to retrieve the values one at a time with the FETCH statement. ♦

For more information on how to write iterator functions, see *IBM Informix User-Defined Routines and Data Types Developer's Guide* and the *IBM Informix DataBlade API Programmer's Guide*.

For information about using an iterator function with a virtual table interface in the FROM clause of a query, see "Iterator Functions" on page 2-603.

NEGATOR

Use the NEGATOR modifier with UDRs that return Boolean values.

The NEGATOR modifier names a companion user-defined function, called a *negator function*, to the current function. A negator function takes the same arguments as its companion function, in the same order, but returns the Boolean complement.

That is, if a function returns `TRUE` for a given set of arguments, its negator function returns `FALSE` when passed the same arguments, in the same order. For example, the following functions are negator functions:

```
equal(a,b)
notequal(a,b)
```

Both functions take the same arguments, in the same order, but return complementary Boolean values. When it is more efficient to do so, the optimizer can use the negator function instead of the function you specify.

To invoke a user-defined function that has a negator function, you must have the Execute privilege on both functions. In addition, the function must have the same owner as its negator function.

Ext

PARALLELIZABLE

Use the PARALLELIZABLE modifier to indicate that an external routine can be executed in parallel in the context of a parallelizable data query (PDQ).

By default, an external routine is non-parallelizable; that is, it executes in sequence.

If your UDR has a complex data type as either a parameter or a returned value, you cannot use the PARALLELIZABLE modifier.

If you specify the PARALLELIZABLE modifier for an external routine that cannot be parallelizable, the database server returns a runtime error.

C

A C language UDR that calls only PDQ thread-safe DataBlade API functions is parallelizable. These categories of DataBlade API functions are PDQ thread safe:

- Data handling

  An exception in this category is that collection manipulation functions (**mi_collection_***) are not PDQ thread safe.
- Session, thread, and transaction management
- Function execution
- Memory management
- Exception handling
- Callbacks
- Miscellaneous

For details of the DataBlade API functions that are included in each category, see the *IBM Informix DataBlade API Function Reference*.

If your UDR calls a function that is not included in one of these categories, it is not PDQ thread safe and therefore not parallelizable. ♦

Java

To parallelize UDR calls, the database server must have multiple instances of JVPs. UDRs written in Java that open a JDBC connection are not parallelizable. ♦

C

PERCALL_COST

Use the PERCALL_COST modifier to specify the approximate CPU usage cost that a C language UDR incurs each time it executes. The optimizer uses the cost you specify to determine the order in which to evaluate SQL predicates in the UDR for best performance. For example, the following query has two predicates joined by a logical AND:

```
SELECT * FROM tab1 WHERE func1() = 10 AND func2() = 'abc';
```

In this example, if one predicate returns `FALSE`, the optimizer need not evaluate the other predicate.

The optimizer uses the specified cost to order the predicates so that the least expensive predicate is evaluated first. The CPU usage cost must be an integer between `1` and 2^{31}`-1`, with `1` the lowest cost and 2^{31}`-1` the most expensive.

To calculate an approximate cost per call, add the following two figures:

- The number of lines of code executed each time the C UDR is called
- The number of predicates that require an I/O access

The default cost per execution is `0`. When you drop the PERCALL_COST modifier, the cost per execution returns to `0`.

C

SELCONST

Use the SELCONST modifier to specify the selectivity of a C UDR. The *selectivity* of the UDR is an estimate of the fraction of the rows that the query will select. That is, the number of times the UDR will need to be executed.

The value of selectivity constant, **selconst**, is a floating-point number between 0 and 1 that represents the fraction of the rows for which you expect the UDR to return TRUE.

C

SELFUNC

Use the SELFUNC modifier with a C UDR to name a companion user-defined function, called a *selectivity function*, to the current UDR. The selectivity function provides selectivity information about the current UDR to the optimizer.

The *selectivity* of a UDR is an estimate of the fraction of the rows that the query will select. That is, it is an estimate of the number of times the UDR will execute.

To execute *sel_func*, you must have Execute privilege on it and on the UDR.

Concept of Selectivity

Selectivity refers to the number of rows that would qualify for a query that does a search based on an equality predicate. The fewer the number of rows that qualify, the more selective the query.

For example, the following query has a search condition based on the **customer_num** column in the **customer** table:

```
SELECT * FROM customer WHERE customer_num = 102;
```

Because each row in the table has a different customer number, this query is highly selective. In contrast, the following query is not selective:

```
SELECT * FROM customer WHERE state = 'CA';
```

Because most of the rows in the **customer** table are for customers in California, more than half of the rows in the table would be returned.

Restrictions on the SELFUNC Modifier

The selectivity function that you specify must satisfy the following criteria:

- It must take the same number of arguments as the current C UDR.
- The data type of each argument must be SELFUNC_ARG.
- It must return a value of type FLOAT between 0 and 1, which represents the percentage of selectivity of the function. (1 is highly selective; 0 is not at all selective.)
- It can be written in any language the database server supports.

A user who invokes the C UDR must have the Execute privilege both on that UDR and on the selectivity function that the SELFUNC modifier specifies.

Both the C UDR and the selectivity function must have the same owner.

For information on how to use the **mi_funcarg*** functions to extract information about the arguments of a selectivity function, see the *IBM Informix DataBlade API Programmer's Guide*.

C

STACK

Use the STACK modifier with a C UDR to override the default stack size that the STACKSIZE configuration parameter specifies.

The STACK modifier specifies the size (in bytes) of the thread stack, which a user thread that executes the UDR uses to hold information such as routine arguments and returned values from functions.

A UDR needs to have enough stack space for all its local variables. For a particular UDR, you might need to specify a stack size larger than the default size to prevent stack overflow.

When a UDR that includes the STACK modifier executes, the database server allocates a thread-stack size of the specified number of bytes. Once the UDR completes execution, subsequent UDRs execute in threads with a stack size that the STACKSIZE configuration parameter specifies (unless any of these subsequent UDRs have also specified the STACK modifier).

For more information about the thread stack, see your *Administrator's Guide* and the *IBM Informix DataBlade API Function Reference*.

C

SPL

VARIANT and NOT VARIANT

Use the VARIANT and NOT VARIANT modifiers with C user-defined functions and SPL functions. A function is *variant* if it returns different results when it is invoked with the same arguments or if it modifies a database or variable state. For example, a function that returns the current date or time is a variant function.

By default, user-defined functions are variant. If you specify NOT VARIANT when you create or modify a user-defined function, the function cannot contain any SQL statements.

If the user-defined function is nonvariant, the database server might cache the returned values of expensive functions. You can create functional indexes only on nonvariant functions. For more information on functional indexes, see "CREATE INDEX" on page 2-144.

C

You can specify VARIANT or NOT VARIANT in this clause or in the EXTERNAL Routine Reference. For more information, see "External Routine Reference" on page 4-187. If you specify the modifier in both places, however, you must use the same modifier in both clauses. ♦

Related Information

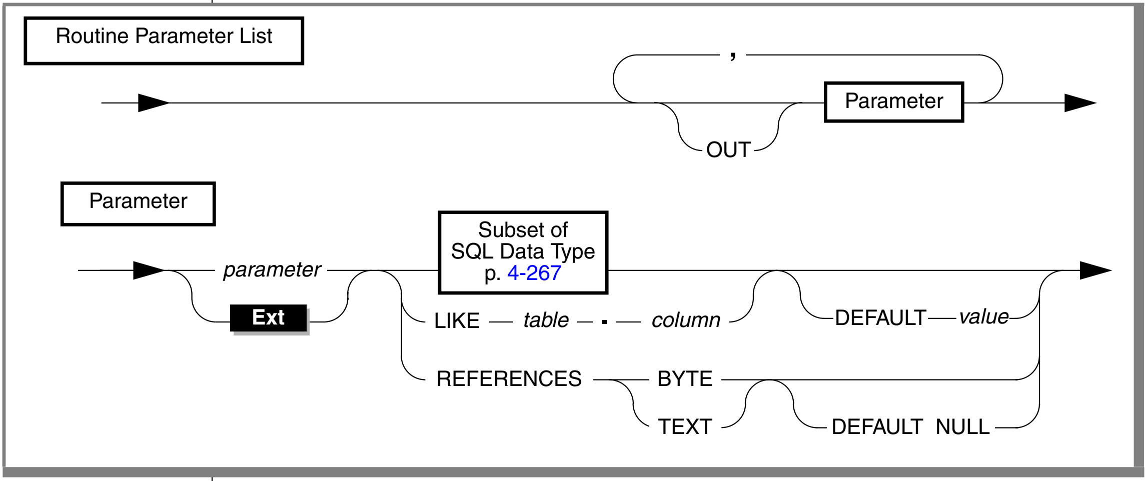
For more information on user-defined routines, see *IBM Informix User-Defined Routines and Data Types Developer's Guide* and the *IBM Informix DataBlade API Programmer's Guide*.

For more information about how these modifiers can affect performance, see your *Performance Guide*.

Routine Parameter List

Use the appropriate part of the Routine Parameter List segment whenever you see a reference to a Routine Parameter List in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
column	Name of a column whose data type is declared for *parameter*	Must exist in the specified table.	Database Object Name, p. 4-46
parameter	Name of a parameter of the UDR	Name is required for SPL routines.	Identifier, p. 4-189
table	Table that contains *column*	The table must exist in the database.	Identifier, p. 4-189
value	Default used if UDR is called with no value for *parameter*	Must be a literal, of the same data type as *parameter*. For opaque types, an input function must be defined.	Literal Number, p. 4-216

Usage

A *parameter* is a formal argument in the declaration of a UDR. (When you subsequently invoke a UDR that has parameters, you must substitute an actual argument for the parameter, unless the parameter has a default value.)

IDS

The name of the parameter is optional for external routines. ♦

XPS

Extended Parallel Server supports no more than a single OUT parameter. If one is specified, it must be the last item in the parameter list. ♦

IDS

The OUT parameters of UDFs of Dynamic Server are described in "Specifying OUT Parameters for a User-Defined Routine" on page 4-269. ♦

When you create a UDR, you declare a *name* and *data type* for each parameter. You can specify the data type directly, or use the LIKE or REFERENCES clause to specify the data type. You can optionally specify a default value.

SPL

You can define any number of SPL routine parameters, but the total length of all parameters passed to an SPL routine must be less than 64 kilobytes. ♦

Ext

No more than nine arguments to a UDR can be DECIMAL data types of SQL that the UDR declares as BigDecimal data types of the Java language.

Any C language UDR that returns an opaque data type must specify **opaque_type** in the `var binary` declaration of the C host variable. ♦

Subset of SQL Data Types

Serial and large-object data types are not valid as parameters. A UDR can declare a parameter of any other data type defined in the database, including any built-in data types except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.

IDS

On Dynamic Server, a parameter can also be a complex data type or a UDT. ♦

Java

Complex types are not valid for parameters of UDRs written in Java. ♦

Using the LIKE Clause

Use the LIKE clause to specify that the data type of a parameter is the same as a column defined in the database. If the ALTER TABLE statement changes the data type of the column, the data type of the parameter also changes.

IDS

If you use the LIKE clause to declare any parameter, you cannot overload the UDR. For example, suppose you create the following user-defined procedure:

```
CREATE PROCEDURE cost (a LIKE tableX.colY, b INT)
. . .
END PROCEDURE;
```

You cannot create another procedure named **cost( )** in the same database with two arguments.

You can, however, create a procedure named **cost( )** with a number of arguments other than two. (Another way to circumvent this restriction on the LIKE clause is through user-defined data types.)

Using the REFERENCES Clause

Use the REFERENCES clause to specify that a parameter contains BYTE or TEXT data. The REFERENCES keyword allows you to use a pointer to a BYTE or TEXT object as a parameter. If you use the DEFAULT NULL option in the REFERENCES clause, and you call the UDR without a parameter, a NULL value is used as the default value.

Using the DEFAULT Clause

Use the DEFAULT keyword followed by an expression to specify a default value for a parameter. If you provide a default value for a parameter, and the UDR is called with fewer arguments than were defined for that UDR, the default value is used. If you do not provide a default value for a parameter, and the UDR is called with fewer arguments than were defined for that UDR, the calling application receives an error.

The following example shows a CREATE FUNCTION statement that specifies a default value for a parameter. This function finds the square of the *i* parameter. If the function is called without specifying the argument for the *i* parameter, the database server uses the default value `0` for the *i* parameter.

```
CREATE FUNCTION square_w_default
   (i INT DEFAULT 0) {Specifies default value of i}
RETURNING INT; {Specifies return of INT value}

   DEFINE j INT; {Defines routine variable j}
   LET j = i * i; {Finds square of i and assigns it to j}
   RETURN j; {Returns value of j to calling module}
END FUNCTION;
```

XPS

In Extended Parallel Server, to re-create this example, use the CREATE PROCEDURE statement instead of the CREATE FUNCTION statement. ♦

***Warning:** When you specify a date value as the default value for a parameter, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 2-digit year, the **DBCENTURY** environment variable setting can affect how the database server interprets the date value, so the UDR might not use the default value that you intended. For more information, see the "IBM Informix Guide to SQL: Reference."*

IDS

Specifying OUT Parameters for a User-Defined Routine

When you register a user-defined routine, you can use the OUT keyword to specify that any parameter in the list is an OUT parameter. Each OUT parameter corresponds to a value the routine returns indirectly, through a pointer. The value that the routine returns through the pointer is an extra value, in addition to any values that it returns explicitly.

After you have registered a user-defined function that has one or more OUT parameters, you can use the function with a statement-local variable (SLV) in an SQL statement. (For information about statement-local variables, see "Statement-Local Variable Expressions" on page 4-169.)

If you specify any OUT parameters, and you use Informix-style parameters, the arguments are passed to the OUT parameters by reference. The OUT parameters are not significant in determining the routine signature.

C

For example, the following declaration of a C user-defined function allows you to return an extra value through the **y** parameter:

```
int my_func( int x, int *y );
```

Register the C function with a CREATE FUNCTION statement similar to this:

```
CREATE FUNCTION my_func( x INT, OUT y INT )
   RETURNING INT
   EXTERNAL NAME "/usr/lib/local_site.so"
   LANGUAGE C
END FUNCTION; ♦
```

Java

For example, this Java method returns an extra value by passing an array:

```
public static String allVarchar(String arg1, String[] arg2)
throws SQLException
{
arg2[0] = arg1;
return arg1;
}
```

You would register the user-defined function with a CREATE FUNCTION statement similar to the following example:

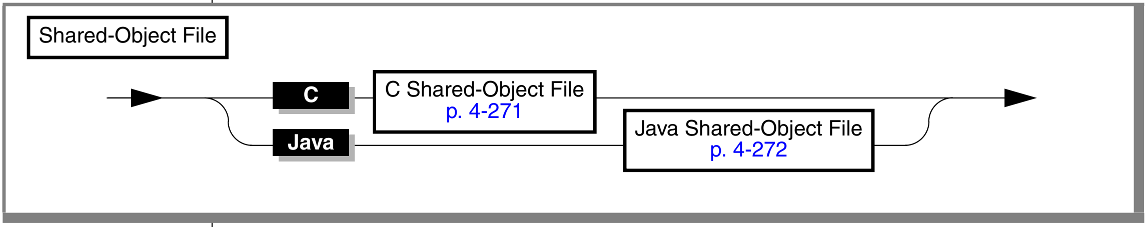
```
CREATE FUNCTION all_varchar(VARCHAR(10), OUT VARCHAR(7))
   RETURNING VARCHAR(7)
   WITH (class = "jvp")
EXTERNAL NAME
'informix.testclasses.jlm.Param.allVarchar(java.lang.String,
java.lang.String[])'
LANGUAGE JAVA; ♦
```

IDS
EXT

Shared-Object Filename

Use a shared-object filename to specify a pathname to an executable object file when you register or alter an external routine.

Syntax



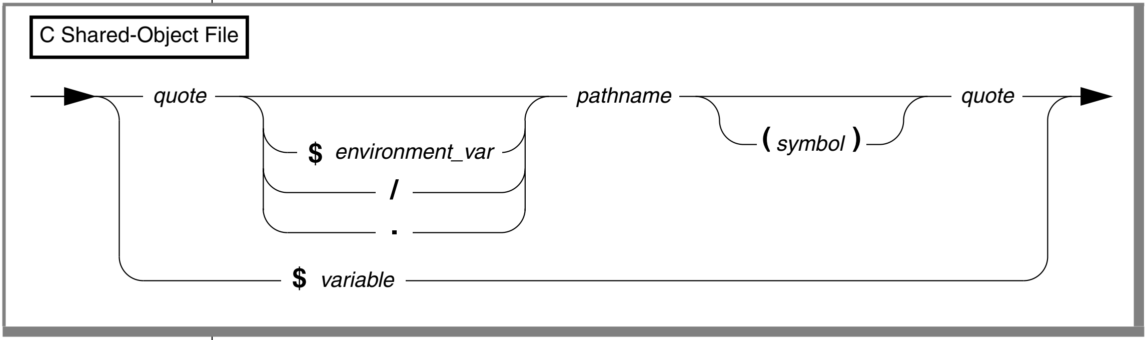
Usage

The syntax by which you specify a shared-object filename depends on whether the external routine is written in the C language or in the Java language. Sections that follow describe each of these external languages.

C

C Shared-Object File

To specify the location of a C shared-object file, specify the path to the dynamically loaded executable file within a quoted pathname or as a variable.



Element	Purpose	Restrictions	Syntax
environment_var	Platform-independent indicator	Must begin with a dollar sign ($)	Identifier, p. 4-189
pathname	Pathname to the file	See notes that follow this table	Identifier, p. 4-189
quote	Either single (**'**) or double (**"**) quotation mark symbol	Opening and closing quotation mark symbols must match	Literal symbol (either **'** or **"**)
symbol	Entry point to the file	Must be enclosed in parentheses	Identifier, p. 4-189
variable	Platform-independent indicator	Must begin with a dollar sign ($)	Identifier, p. 4-189

The following rules affect *pathname* and filename specifications in C:

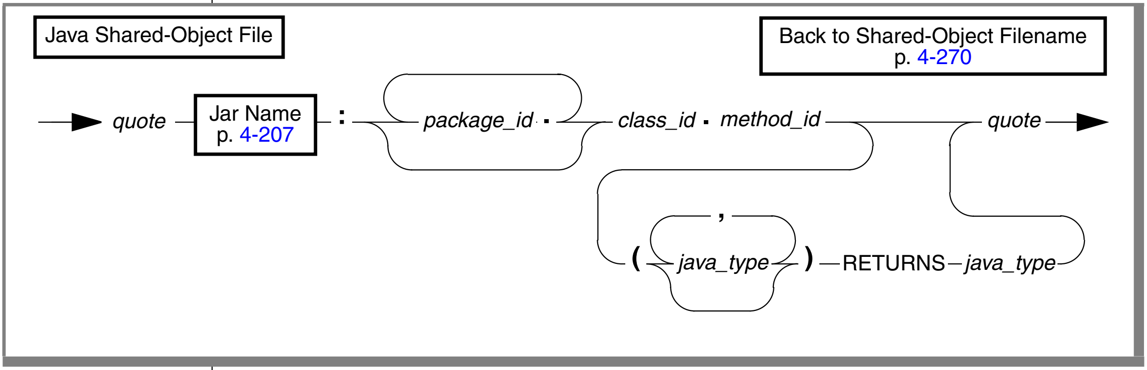
- A filename (with no *pathname*) can specify an internal function.
- You can omit the period (.) symbol if *pathname* is relative to the current directory when the CREATE or ALTER statement is run.
- UNIX An absolute pathname must begin with a slash (/) symbol, and each directory name must end with a slash (/) symbol. ♦
- Windows An absolute pathname must begin with a backslash (\ ) symbol, and each directory name must end with a backslash (\ ) symbol. ♦
- The filename at the end of *pathname* must have the **.so** file extension and must refer to an executable file in a shared object library.
- Use a *symbol* only if the entry point to the dynamically loadable executable object file has a different name from the UDR that you are registering with CREATE FUNCTION or CREATE PROCEDURE.

- If you specify a *variable*, it must contain the full pathname to the executable file.
- You can include whitespace characters, such as blank spaces or tab characters, within a quoted pathname.

Java

Java Shared-Object File

To specify the name of a Java shared-object file, specify the name of the static Java method to which the UDR corresponds and the location of the Java binary that defines the method.



Element	Purpose	Restrictions	Syntax
class_id	Java class whose method implements the UDR	Class must exist in the **.jar** file that Jar Name identifies	Must conform to rules for Java identifiers
java_type	Java data type for a parameter in the Java-method signature	Must be defined in a JDBC class or by an SQL-to-Java mapping	Must conform to rules for Java identifiers
method_id	Name of the Java method that implements the UDR	Must exist in the Java class that *java_class_name* specifies	Must conform to rules for Java identifiers
package_id	Name of package that contains the Java class	Must exist	Must conform to rules for Java identifiers
quote	Single (**'**) or double (**"**) quotation mark	Opening and closing quotation marks must match	Literal symbol (**'** or **"**) entered at the keyboard

Before you can create a UDR written in the Java language, you must assign a jar identifier to the external jar file with the **sqlj.install_jar** procedure. For more information, see "sqlj.install_jar" on page 2-418.

You can include the Java signature of the method that implements the UDR in the shared-object filename.

- If you do *not* specify the Java signature, the routine manager determines the *implicit* Java signature from the SQL signature in the CREATE FUNCTION or CREATE PROCEDURE statement.

  It maps SQL data types to the corresponding Java data types with the JDBC and SQL-to-Java mappings. For information on mapping user-defined data types to Java data types, see "sqlj.setUDTExtName" on page 2-422.
- If you do specify the Java signature, the routine manager uses this *explicit* Java signature as the name of the Java method to use.

For example, if the Java method **explosiveReaction( )** implements the Java UDR **sql_explosive_reaction( )** (as discussed in "sqlj.install_jar" on page 2-418), its shared-object filename could be:

```
course_jar:Chemistry.explosiveReaction
```

The preceding shared-object filename provides an implicit Java signature. The following shared-object filename is the equivalent with an explicit Java signature:

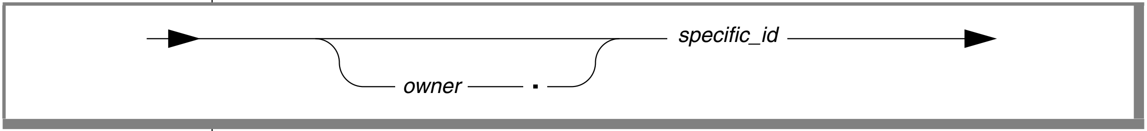
```
course_jar:Chemistry.explosiveReaction(int)
```

IDS

Specific Name

Use a specific name to give a UDR a name that is unique in the database or name space. Use the Specific Name segment whenever you see a reference to a specific name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
owner	Owner of the UDR	Must be same as *owner* of function or procedure name of this UDR. See also "Restrictions on the Owner Name" on page 4-274.	Identifier, p. 4-189
specific_id	Unique name of the UDR	Must be no more than 128 characters long. See also "Restrictions on the Specific_ID" on page 4-275.	Identifier, p. 4-189

Usage

A *specific name* is a unique identifier that the CREATE PROCEDURE or CREATE FUNCTION statement declares as an alternative name for a UDR.

Because you can overload routines, a database can have more than one UDR with the same name and different parameter lists. You can assign a UDR a specific name that uniquely identifies the specific UDR.

If you declare a specific name when you create the UDR, you can later use that name when you alter, drop, grant, or revoke privileges, or update statistics on that UDR. Otherwise, you need to include the parameter data types with the UDR name, if the name alone does not uniquely identify the UDR.

Restrictions on the Owner Name

When you give a UDR a specific name, the *owner* must be the same *authorization identifier* used in the function name or procedure name for the UDR that you create. That is, whether or not you specify the owner name in either the UDR name or the specific name or both, the owner names must match.

When you do not specify an owner name, the database server uses the user ID of the person who creates the UDR. Therefore, if you specify the owner name in one location and not the other, the owner name that you specify must match your user ID.

Restrictions on the Specific_ID

In a database that is not ANSI-compliant, *specific_id* must be unique among routine names within the database. Two UDRs cannot have the same *specific_id,* even if they have different owners.

ANSI

In an ANSI-compliant database, the combination *owner.specific_id* must be unique. That is, the specific name must be unique among UDRs that have the same owner.

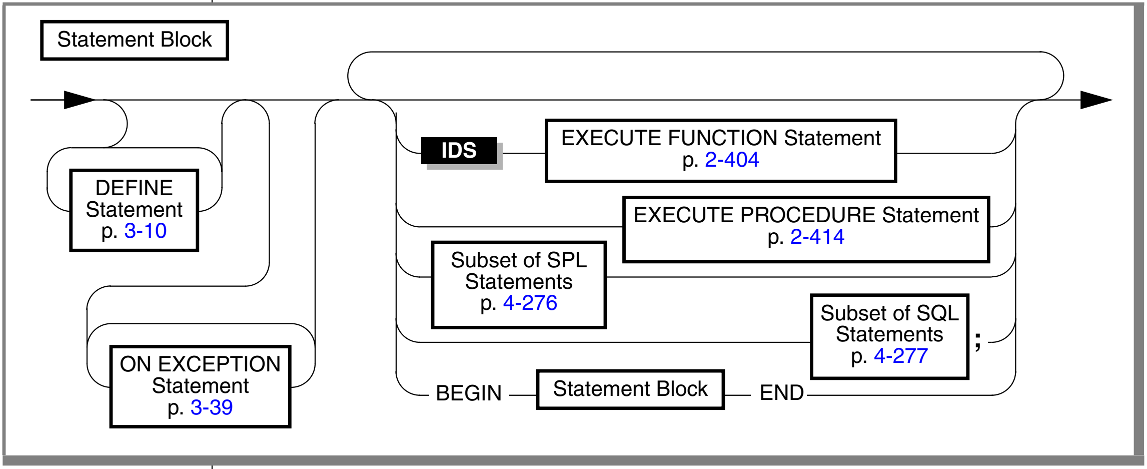
The specific name must be unique within the schema. ♦

SPL

Statement Block

Use a statement block to specify SPL and SQL operations to take place when an SPL statement that includes this segment is executed. Use this segment whenever you see a reference to a Statement Block in a syntax diagram.

Syntax


Statement Block
DEFINE Statement p. 3-10
ON EXCEPTION Statement p. 3-39
IDS
EXECUTE FUNCTION Statement p. 2-404
EXECUTE PROCEDURE Statement p. 2-414
Subset of SPL Statements p. 4-276
Subset of SQL Statements p. 4-277
;
BEGIN
Statement Block
END

Usage

SPL and SQL statements can appear in a statement block. If the statement block is empty, no operation takes place when control of execution within the SPL routine passes to the empty SPL statement block.

Subset of SPL Statements Valid in the Statement Block

The diagram for the "Statement Block" on page 4-276 refers to this section. You can use any of the following SPL statements in the statement block:

- CALL
- CONTINUE
- EXIT
- FOR
- FOREACH
- IF
- LET
- RAISE EXCEPTION
- RETURN
- SYSTEM
- TRACE
- WHILE

SQL Statements Not Valid in an SPL Statement Block

The diagram for the "Statement Block" on page 4-276 refers to this section. The following SQL statements are *not* valid in an SPL statement block:

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
CLOSE
CLOSE DATABASE
CONNECT
CREATE DATABASE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE PROCEDURE
CREATE PROCEDURE FROM
DATABASE
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
INFO
LOAD
OPEN
OUTPUT
PREPARE
PUT
SET CONNECTION
SET DESCRIPTOR
UNLOAD
UPDATE STATISTICS
WHENEVER

For example, you cannot close the current database or select a new database within an SPL routine. Likewise you cannot drop the current SPL routine within the same routine. You can, however, drop another SPL routine.

You can use a SELECT statement in only two cases:

- You can use the INTO TEMP clause to put the results of the SELECT statement into a temporary table.
- You can use the SELECT... INTO form of the SELECT statement to put the resulting values into SPL variables.

If an SPL routine is later to be called as part of a data-manipulation statement, additional restrictions exist. For more information, see "Restrictions on SPL Routines in Data-Manipulation Statements" on page 4-279.

Nested Statement Blocks

You can use the BEGIN and END keywords to delimit a statement block that is nested within another statement block.

Scope of Reference of SPL Variables and Exception Handlers

The BEGIN-END keywords can limit the scope of SPL variables and exception handlers. Declarations of variables and definitions of exception handlers inside a BEGIN-END statement block are local to that statement block and are not visible from outside the statement block. The following code uses a BEGIN-END statement block to delimit the scope of reference of variables:

```
CREATE DATABASE demo;
CREATE TABLE tracker (
   who_submitted CHAR(80), -- Show what code was running.
   value INT,              -- Show value of the variable.
    sequential_order SERIAL
-- Show order in which statements were executed.
    );
CREATE PROCEDURE demo_local_var()
DEFINE var1 INT;
DEFINE var2 INT;
   LET var1 = 1;
   LET var2 = 2;
   INSERT INTO tracker (who_submitted, value)
    VALUES ('var1 param before sub-block', var1);
BEGIN
   DEFINE var1 INT;   -- same name as global parameter.
   LET var1 = var2;
   INSERT INTO tracker (who_submitted, value)
   VALUES ('var1 var defined inside the "IF/BEGIN".', var1);
END
INSERT INTO tracker (who_submitted, value)
   VALUES ('var1 param after sub-block (unchanged!)', var1);
END PROCEDURE;
EXECUTE PROCEDURE demo_local_var();
SELECT sequential_order, who_submitted, value FROM tracker
 ORDER BY sequential_order;
```

This example declares three variables, two of which are named **var1**. (Name conflicts are created here to illustrate which variables are visible. Using the same name for different variables is generally not recommended, because conflicting names of variables can make your code more difficult to read.)

Because of the statement block, only one **var1** variable is in scope at a time.

The **var1** variable that is declared inside the statement block is the only **var1** variable that can be referenced from within the statement block.

The **var1** variable that is declared outside the statement block is not visible within the statement block. Because it is out of scope, it is unaffected by the change in value to the **var1** variable that takes place inside the statement block. After all the statements run, the outer **var1** still has a value of `1`.

The **var2** variable is visible within the statement block because it was not superseded by a name conflict with a block-specific variable.

Restrictions on SPL Routines in Data-Manipulation Statements

IDS

If an SPL routine is called as part of an INSERT, UPDATE, DELETE, or SELECT statement, the routine cannot execute any statement in the following list:

ALTER ACCESS_METHOD
ALTER FRAGMENT
ALTER INDEX
ALTER OPTICAL CLUSTER
ALTER TABLE
BEGIN WORK
COMMIT WORK
CREATE ACCESS_METHOD
CREATE AGGREGATE
CREATE DISTINCT TYPE
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE ROLE
CREATE ROW TYPE
CREATE TRIGGER
DROP ACCESS_METHOD
DROP AGGREGATE
DROP INDEX
DROP OPCLASS
DROP OPTICAL CLUSTER
DROP ROLE
DROP ROW TYPE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP VIEW
RENAME COLUMN
RENAME TABLE
ROLLBACK WORK
SET CONSTRAINTS

♦

XPS

If an SPL routine is called as part of an INSERT, UPDATE, DELETE, or SELECT statement, the routine can execute *only* the following statements:

SELECT
SET PLOAD FILE
SET DEBUG FILE TO
SET EXPLAIN
SET OPTIMIZATION

♦

If the SPL routine is called within a statement that is not a data- manipulation statement (namely EXECUTE FUNCTION or EXECUTE PROCEDURE), the SPL routine can execute any statement that is not listed in the section "SQL Statements Not Valid in an SPL Statement Block" on page 4-277.

Transactions in SPL Routines

In a database that is not ANSI-compliant, you can use the BEGIN WORK and COMMIT WORK statements in an SPL statement block to start a transaction, finish a transaction, or start and finish a transaction in the same SPL routine. If you start a transaction in a routine that is executed remotely, you must finish the transaction before the routine exits.

As previously noted, however, the ROLLBACK WORK statement is not valid in an SPL statement block.

IDS

Support for Roles and User Identity

You can use roles with SPL routines. You can execute role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements within an SPL routine. You can also grant privileges to roles with the GRANT statement within an SPL routine. Privileges that a user has acquired through enabling a role or by a SET SESSION AUTHORIZATION statement are not relinquished when an SPL routine is executed.

For further information about roles, see the CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE statements.

Appendix

A

Reserved Words for IBM Informix Dynamic Server

The SQL language has no "reserved words," in the sense of a character string that obeys the rules for identifiers (page 4-189) but always produces a compilation error or runtime error. Your application might encounter restricted functionality, however, or unexpected results, if you define a user-defined function or procedure whose name is the same as a built-in SQL function, expression, or operator.

This appendix lists the keywords in the Informix implementation of the SQL language in Dynamic Server. In general, you should not declare any of these keywords as SQL identifiers. If you do, errors or syntactic ambiguities can occur if the identifier appears in a context where the keyword is valid, and your code will be more difficult to read and to maintain.

If you receive an error message that seems unrelated to the SQL statement that caused the error, you might wish to review this appendix to see if a keyword has been used as an identifier.

To avoid using a keyword as an identifier, you can qualify the identifier with an owner name or modify the identifier. For example, rather than name a database object **CURRENT**, you might name it `o_current` or `juanita.current`. For a discussion of potential problems in using keywords as identifiers, and of additional workarounds for specific keywords, see "Potential Ambiguities and Syntax Errors" on page 4-194. See also *IBM Informix Guide to SQL: Tutorial* for more information about using keywords as identifiers in SQL applications.

A

ABSOLUTE
ACCESS
ACCESS_METHOD
ADD
AFTER
AGGREGATE
ALIGNMENT
ALL
ALL_ROWS
ALLOCATE
ALTER
AND
ANSI
ANY
APPEND
AS
ASC
AT
ATTACH
AUDIT
AUTHORIZATION
AUTO
AUTOFREE
AVG
AVOID_EXECUTE
AVOID_SUBQF

B

BEFORE
BEGIN
BETWEEN
BINARY
BOOLEAN
BOTH
BUFFERED
BUILTIN
BY
BYTE

C

CACHE
CALL
CANNOTHASH
CARDINALITY
CASCADE
CASE
CAST
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLASS
CLIENT
CLOSE
CLUSTER
CLUSTERSIZE
COARSE
COBOL
CODESET
COLLATION
COLLECTION
COLUMN
COMMIT
COMMITTED
COMMUTATOR
CONCURRENT
CONNECT
CONNECTION
CONST
CONSTRAINT
CONSTRAINTS
CONSTRUCTOR
CONTINUE
COPY
COSTFUNC
COUNT
CRCOLS
CREATE
CROSS
CURRENT
CURSOR
CYCLE

D

DATABASE
DATAFILES
DATASKIP
DATE
DATETIME
DAY
DBA
DBDATE
DBMONEY
DBPASSWORD
DEALLOCATE
DEBUG
DEC
DEC_T
DECIMAL
DECLARE
DECODE
DEFAULT
DEFERRED
DEFERRED_PREPARE
DEFINE
DELAY
DELETE
DELIMITER
DELUXE
DEREF
DESC
DESCRIBE
DESCRIPTOR
DETACH
DIAGNOSTICS
DIRTY
DISABLED
DISCONNECT
DISTINCT
DISTRIBUTEBINARY
DISTRIBUTESREFERENCES
DISTRIBUTIONS
DOCUMENT
DOMAIN
DONOTDISTRIBUTE
DORMANT
DOUBLE
DROP
DTIME_T

E

EACH
ELIF
ELSE
ENABLED
END
ENUM
ENVIRONMENT
ERROR
ESCAPE
EXCEPTION
EXCLUSIVE
EXEC
EXECUTE
EXECUTEANYWHERE
EXISTS
EXIT
EXPLAIN
EXPLICIT
EXPRESS
EXPRESSION
EXTEND
EXTENT
EXTERNAL

F

FALSE
FAR
FETCH
FILE
FILLFACTOR
FILTERING
FIRST
FIRST_ROWS
FIXCHAR
FIXED
FLOAT
FLUSH
FOR
FOREACH
FOREIGN
FORMAT
FORTRAN
FOUND
FRACTION
FRAGMENT
FREE
FROM
FULL
FUNCTION

G - H

GENERAL
GET
GK
GLOBAL
GO
GOTO
GRANT
GROUP
HANDLESNULLS
HASH
HAVING
HIGH
HOLD
HOUR
HYBRID

I

IF
IFX_INT8_T
IFX_LO_CREATE_SPEC_T
IFX_LO_STAT_T
IMMEDIATE
IMPLICIT
IN
INCREMENT
INDEX
INDEXES
INDICATOR
INFORMIX
INIT
INNER
INSERT
INSTEAD
INT
INT8
INTEG
INTEGER
INTERNAL
INTERNALLENGTH
INTERVAL
INTO
INTRVL_T
IS
ISCANONICAL
ISOLATION
ITEM
ITERATOR

J - K

JOIN
KEEP
KEY

L

LABELEQ
LABELGE
LABELGLB
LABELGT
LABELLE
LABELLT
LABELLUB
LABELTOSTRING
LANGUAGE
LAST
LEADING
LEFT
LET
LEVEL
LIKE
LIST
LISTING
LOC_T
LOCAL
LOCATOR
LOCK
LOCKS
LOG
LONG
LOW
LOWER
LVARCHAR

M

MATCHES
MAX
MAXERRORS
MAXLEN
MAXVALUE
MDY
MEDIAN
MEDIUM
MEMORY_RESIDENT
MIDDLE
MIN
MINUTE
MINVALUE
MODE
MODERATE
MODIFY
MODULE
MONEY
MONTH
MOUNTING
MULTISET

N

NAME
NCHAR
NEGATOR
NEW
NEXT
NO
NOCACHE
NOCYCLE
NOMAXVALUE
NOMIGRATE
NOMINVALUE
NON_RESIDENT
NONE
NOORDER
NORMAL
NOT
NOTEMPLATEARG
NULL
NUMERIC
NVARCHAR
NVL

O

OCTET_LENGTH
OF
OFF
OLD
ON
ONLY
OPAQUE
OPCLASS
OPEN
OPERATIONAL
OPTICAL
OPTIMIZATION
OPTION
OR
ORDER
OUT
OUTPUT
OUTER

P

PAGE
PARALLELIZABLE
PARAMETER
PASCAL
PASSEDBYVALUE
PDQPRIORITY
PERCALL_COST
PLI
PLOAD
PRECISION
PREPARE
PREVIOUS
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROCEDURE
PUBLIC
PUT

R

RAISE
RANGE
RAW
READ
REAL
RECORDEND
REF
REFERENCES
REFERENCING
REGISTER
REJECTFILE
RELATIVE
RELEASE
REMAINDER
RENAME
REOPTIMIZATION
REPEATABLE
REPLICATION
RESERVE
RESOLUTION
RESOURCE
RESTART
RESTRICT
RESUME
RETAIN
RETURN
RETURNING
RETURNS
REUSE
REVOKE
RIGHT
ROBIN
ROLE
ROLLBACK
ROLLFORWARD
ROUND
ROUTINE
ROW
ROWID
ROWIDS
ROWS

S

SAMEAS
SAMPLES
SCHEDULE
SCHEMA
SCRATCH
SCROLL
SECOND
SECONDARY
SECTION
SELCONST
SELECT
SELFUNC
SEQUENCE
SERIAL
SERIAL8
SERIALIZABLE
SERVERUUID
SESSION
SET
SHARE
SHORT
SIGNED
SIZE
SKALL
SKINHIBIT
SKSHOW
SMALLFLOAT
SMALLINT
SOME
SPECIFIC
SQL
SQLCODE
SQLCONTEXT
SQLERROR
SQLSTATE
SQLWARNING
STABILITY
STACK
STANDARD
START
STATIC
STATISTICS
STDEV
STEP
STOP
STORAGE
STRATEGIES
STRING
STRINGTOLABEL
STRUCT
STYLE
SUBSTR
SUBSTRING
SUM
SUPPORT
SYNC
SYNONYM
SYSTEM

T

- TABLE
- TEMP
- TEXT
- THEN
- TIME
- TIMEOUT
- TO
- TODAY
- TRACE
- TRAILING
- TRANSACTION
- TRIGGER
- TRIGGERS
- TRIM
- TRUE
- TRUNCATE
- TYPE
- TYPEDEF

U

- UNCOMMITTED
- UNDER
- UNION
- UNIQUE
- UNITS
- UNKNOWN
- UNLOCK
- UNSIGNED
- UPDATE
- UPPER
- USAGE
- USE_SUBQF
- USER
- USING

V

- VALUE
- VALUES
- VAR
- VARCHAR
- VARIABLE
- VARIANCE
- VARIANT
- VARYING
- VIEW
- VIOLATIONS
- VOID
- VOLATILE

W

- WAIT
- WARNING
- WHEN
- WHENEVER
- WHERE
- WHILE
- WITH
- WITHOUT
- WORK
- WRITE

X - Z

- XLOAD
- XUNLOAD
- YEAR

Appendix

B

Reserved Words for IBM Informix Extended Parallel Server

The SQL language has no “reserved words,” in the sense of a character string that obeys the rules for identifiers (page 4-189) but always produces a compilation error or runtime error. Your application might encounter restricted functionality, however, or unexpected results, if you define an SPL routine whose name is the same as a built-in SQL function, expression, or operator.

This appendix lists the keywords in the Informix implementation of the SQL language in Extended Parallel Server. In general, you should not use any of these keywords as SQL identifiers. If you do, errors or syntactic ambiguities can occur if the identifier appears in a context where the keyword is valid, and your code will be more difficult to read and to maintain.

If you receive an error message that seems unrelated to the SQL statement that caused the error, you might wish to review this appendix to see if a keyword has been used as an identifier.

To avoid using a keyword as an identifier, you can qualify the identifier with an owner name or modify the identifier. For example, rather than name a database object **CURRENT**, you might name it `o_current` or `juanita.current`. For a discussion of potential problems in using keywords as identifiers, and of additional workarounds for specific keywords, see “Potential Ambiguities and Syntax Errors” on page 4-194. See also *IBM Informix Guide to SQL: Tutorial* for more information about using keywords as identifiers in SQL applications.

A

ADD
AFTER
ALL
ALTER
AND
ANSI
ANY
APPEND
AS
ASC
ATTACH
AUDIT
AUTHORIZATION
AVG

B

BEFORE
BEGIN
BETWEEN
BITMAP
BOTH
BUFFERED
BY
BYTE

C

CACHE
CALL
CASCADE
CASE
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLOSE
CLUSTER
CLUSTERSIZE
COARSE
COBOL
CODESET
COLUMN
COMMIT
COMMITTED
CONNECT
CONSTRAINT
CONSTRAINTS
CONTINUE
COPY
COUNT
CREATE
CURRENT
CURSOR

D

DATABASE
DATAFILES
DATASKIP
DATE
DATETIME
DAY
DBA
DBDATE
DBMONEY
DEBUG
DEC
DECIMAL
DECLARE
DECODE
DEFAULT
DEFERRED
DEFINE
DELETE
DELIMITER
DELUXE
DESC
DETACH
DIRTY
DISTINCT
DISTRIBUTIONS
DOCUMENT
DOUBLE
DROP

E

EACH
ELIF
ELSE
END
ENVIRONMENT
ESCAPE
EXCEPTION
EXCLUSIVE
EXEC
EXECUTE
EXISTS
EXIT
EXPLAIN
EXPRESS
EXPRESSION
EXTEND
EXTENT
EXTERNAL

F

FETCH
FILE
FILLFACTOR
FILTERING
FIRST
FLOAT
FOR
FOREACH
FOREIGN
FORMAT
FORTRAN
FOUND
FRACTION
FRAGMENT
FROM

G

GK
GLOBAL
GO
GOTO
GRANT
GROUP

H

HASH
HAVING
HIGH
HOLD
HOUR
HYBRID

I

IF
IMMEDIATE
IN
INDEX
INDICATOR
INIT
INSERT
INT
INTEGER
INTERVAL
INTO
IS
ISOLATION

K

KEY

L

LABELEQ
LABELGE
LABELGT
LABELLE
LABELLT
LANGUAGE
LEADING
LET
LEVEL
LIKE
LISTING
LOCAL
LOCK
LOCKS
LOG
LOW

M

MATCHES
MAX
MAXERRORS
MEDIUM
MEMORY_RESIDENT
MIDDLE
MIN
MINUTE
MODE
MODIFY
MODULE
MONEY
MONTH
MOUNTING

N

NCHAR
NEW
NEXT
NO
NON_RESIDENT
NORMAL
NOT
NULL
NUMERIC
NVARCHAR
NVL

O

OCTET_LENGTH
OF
OFF
OLD
ON
ONLY
OPEN
OPERATIONAL
OPTICAL
OPTIMIZATION
OPTION
OR
ORDER
OUTER

P

PAGE
PASCAL
PDQPRIORITY
PLI
PLOAD
PRECISION
PRIMARY
PRIVATE
PRIVILEGES
PROCEDURE
PUBLIC

R

RAISE
RANGE
RAW
READ
REAL
RECORDEND
RECOVER
REFERENCES
REFERENCING
REJECTFILE
RELEASE
REMAINDER
RENAME
REPEATABLE
RESERVE
RESOLUTION
RESOURCE
RESTRICT
RESUME
RETAIN
RETURN
RETURNING
RETURNS
REVOKE
RIDLIST
ROBIN
ROLLBACK
ROLLFORWARD
ROUND
ROW
ROWS

S

SAMEAS
SAMPLES
SCHEDULE
SCHEMA
SCRATCH
SCROLL
SECOND
SECTION
SELECT
SERIAL
SERIALIZABLE
SET
SHARE
SIZE
SKALL
SKINHIBIT
SKSHOW
SMALLFLOAT
SMALLINT
SOME
SQL
SQLCODE
SQLERROR
SHARE
STANDARD
START
STATIC
STATISTICS
STDEV
STEP
STOP
SUBSTRING
SUM
SYNC
SYNONYM
SYSTEM

T

TABLE
TEMP
TEXT
THEN
TIMEOUT
TO
TRACE
TRAILING
TRANSACTION
TRIGGER
TRIM
TRUNCATE
TYPE

U

UNCOMMITTED
UNION
UNIQUE
UNITS
UNLOCK
UPDATE
USAGE
USING

V

VALUES
VARCHAR
VARIANCE
VARYING
VIEW
VIOLATIONS

W

WAIT
WHEN
WHENEVER
WHERE
WHILE
WITH
WORK
WRITE

X

XLOAD
XUNLOAD

Y

YEAR

Appendix

C

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

Index