INFORMIX®-Universal Server

Informix Guide to SQL: Syntax

Version 9.1
March 1997
Set No. 000-3878

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Documentation Team: Diana Chase, Sally Cox, Barbara Daniell, Brian Deutscher, Geeta Karmarkar, Abby Knott, Dawn Maneval, Virginia Panlasigui, Judith Sherwood

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

Table of Contents

Introduction

Chapter 1 SQL Statements

Chapter 2 SPL Statements

Index

Introduction

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

The *Informix Guide to SQL: Syntax* manual contains syntax descriptions for the Structured Query Language (SQL) and Stored Procedure Language (SPL) statements that Universal Server supports.

This manual is part of a series of manuals that discusses the Informix implementation of SQL. This volume and the *Informix Guide to SQL: Reference* are references that you can use on a daily basis after you finish reading the *Informix Guide to SQL: Tutorial*.

Important: *This manual does not cover the product called INFORMIX-SQL or any other Informix application development tool.*

Organization of This Manual

This manual includes the following chapters:

- This Introduction provides an overview of the manual and describes the documentation conventions used.
- Chapter 1, "SQL Statements," describes SQL statements and segments. The chapter is divided into six sections. The first four sections provide an introduction to the statements and segments. These sections cover the following subjects: entry of SQL statements, entry of SQL comments, categories of SQL statements, and categories of ANSI compliance. The fifth and sixth sections, "Statements" and "Segments," are the major sections of the chapter.
    - ❑ "Statements" explains the workings of all the SQL statements that Informix products support. Detailed syntax diagrams walk you through every clause of each SQL statement, and syntax tables explain the input parameters for each clause. Thorough usage instructions, pertinent examples, and references to related material complete the description of each SQL statement.
    - ❑ "Segments" explains all the SQL segments. SQL segments are language elements, such as table names and expressions, that occur in many SQL statements. Instead of describing each segment in each statement where it occurs, this manual provides a comprehensive stand-alone description of each segment. Whenever a segment occurs within a given syntax diagram, the diagram points to the stand-alone description of the segment in this section for further information.
- Chapter 2, "SPL Statements," presents all the detailed syntax diagrams and explanations for SPL statements. You can use stored procedures to perform any function you can perform in SQL as well as to expand what you can accomplish with SQL alone. You write a stored procedure using SPL and SQL statements. For task-oriented information about using stored procedures, see the *Informix Guide to SQL: Tutorial*.
- The Index is a combined index for the manuals in the SQL series. Each page reference in the index ends with a code that identifies the manual in which the page appears. The same index also appears in the *Informix Guide to SQL: Reference* and the *Informix Guide to SQL: Tutorial*.

The following items are an integral part of this manual although they do not appear in it:

- A description of the structure and contents of the **stores7** demonstration database appears in the *Informix Guide to SQL: Reference*.
- A glossary of object-relational database terms that are used in the SQL manual series appears in the *Informix Guide to SQL: Reference*.

Types of Users

This manual is written for SQL users, database administrators and SQL developers who use Informix products and SQL on a regular basis.

Software Dependencies

This manual assumes that you are using the following Informix software:

- INFORMIX-Universal Server, Version 9.1

  The database server must be installed either on your computer or on another computer to which your computer is connected over a network.

  In this manual, all instances of Universal Server refer to INFORMIX-Universal Server.
- An Informix SQL application programming interface (API), such as INFORMIX-ESQL/C, Version 9.1, or the DB-Access database access utility, which is shipped as part of your database server.

  The SQL API or DB-Access enables you to compose queries, send them to the database server, and view the results that the database server returns.

Assumptions About Your Locale

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

This manual assumes that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale(s). For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *Guide to GLS Functionality*.

Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Sample command files are also included.

Many examples in Informix manuals are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in Appendix A of the *Informix Guide to SQL: Reference*.

The script that you use to install the demonstration database is called **dbaccessdemo7** and is located in the **$INFORMIXDIR/bin** directory. For a complete explanation of how to create and populate the demonstration database on your database server, refer to the *DB-Access User Manual*.

Major Features

The following SQL features are new with Universal Server, Version 9.1.

ALLOCATE COLLECTION
ALLOCATE ROW
CREATE CAST
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE ROUTINE FROM
CREATE ROW TYPE
DEALLOCATE COLLECTION
DEALLOCATE ROW
DROP CAST
DROP FUNCTION
DROP OPCLASS
DROP ROUTINE
DROP ROW TYPE
DROP TYPE
EXECUTE FUNCTION
SET AUTOFREE
SET DEFERRED_PREPARE
Argument
Collection Derived Table
External Routine Reference
Function Name
Literal Collection
Literal Row
Quoted Pathname
Return Clause
Routine Modifier
Routine Parameter List
Specific Name
Statement Block

The following SQL features are enhanced for use with Universal Server, Version 9.1.

ALLOCATE DESCRIPTOR
ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
CREATE INDEX
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE SCHEMA
CREATE SYNONYM
CREATE TABLE
CREATE VIEW
DEALLOCATE DESCRIPTOR
DECLARE
DELETE
DESCRIBE
DROP INDEX
DROP PROCEDURE
DROP TABLE
EXECUTE
EXECUTE PROCEDURE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
GRANT
INFO
INSERT
OPEN
PREPARE
PUT
REVOKE
SELECT
SET DESCRIPTOR
SET EXPLAIN
UPDATE
UPDATE STATISTICS
Condition
Data Type
Expression
Procedure Name
Quoted String

The Introduction to each Version 9.1 product manual contains a list of major features for that product. The Introduction to each manual in the Version 9.1 *Informix Guide to SQL* series contains a list of new SQL features.

Major features for Version 9.1 Informix products also appear in release notes.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Syntax conventions
- Sample-code conventions

Typographical Conventions

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

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.

(1 of 2)

Convention	Meaning
`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 feature-, product-, platform-, or compliance-specific information.

(2 of 2)

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 warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
⚠	The *warning* icon identifies vital instructions, cautions, or critical information.
⇨	The *important* icon identifies significant information about the feature or operation that is being described.
💡	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

Feature and Product Icons

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

Icon	Description
GLS	Identifies information that relates to the Informix Global Language Support (GLS) feature.
D/B	Identifies information that is valid only for DB-Access.
ESQL	Identifies information that is valid only for SQL statements in INFORMIX-ESQL/C.
E/C	Identifies information that is valid only for INFORMIX-ESQL/C.
OP	Identifies information that is valid only for INFORMIX-OnLine/Optical.
SPL	Identifies information that is valid only if you are using Informix Stored Procedure Language (SPL).

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature- or product-specific information.

Compliance Icons

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

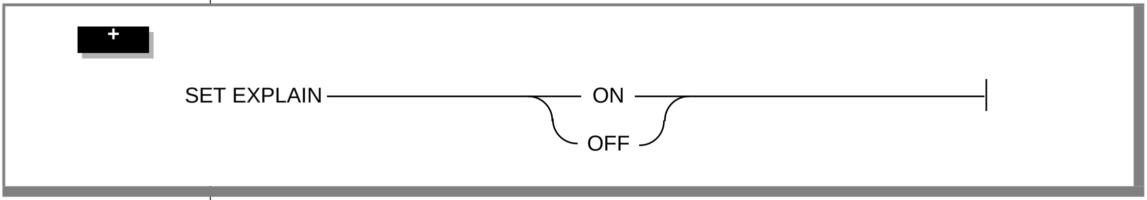
Icon	Description
ANSI	Identifies information that is specific to an ANSI-compliant database.
X/O	Identifies functionality that conforms to X/Open. This functionality is available when you compile your SQL API with the **-xopen** flag.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the compliance information.

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 from the right. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

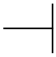
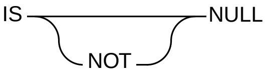
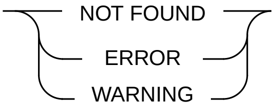
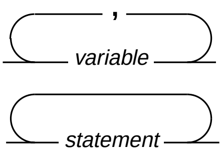
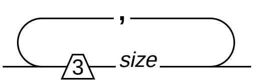
You might encounter one or more of the following elements on a path.

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. 1-14 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.

(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 a syntax diagram: D/B This path is valid only for DB-Access. E/C This path is valid only for INFORMIX-ESQL/C. EXT This path is valid for external routines. SPL This path is valid only if you are using Informix Stored Procedure Language (SPL). SQLE This path is valid for the SQL Editor. OP This path is valid only for INFORMIX-OnLine/Optical. + This path is an Informix extension to ANSI SQL-92 entry-level standard SQL. If you initiate Informix extension checking and include this syntax branch, you receive a warning. If you have set the **DBANSIWARN** environment variable at compile time, or have used the **-ansi** compile flag, you receive warnings at compile time. If you have **DBANSIWARN** set at runtime, or if you compiled with the **-ansi** flag, warning flags are set in the **sqlwarn** structure. GLS This path is valid only if your database or application uses a nondefault GLS locale.
– ALL –	A shaded option is the default action.
──►──►	Syntax that is enclosed between a pair of arrows is a subdiagram.

(2 of 3)

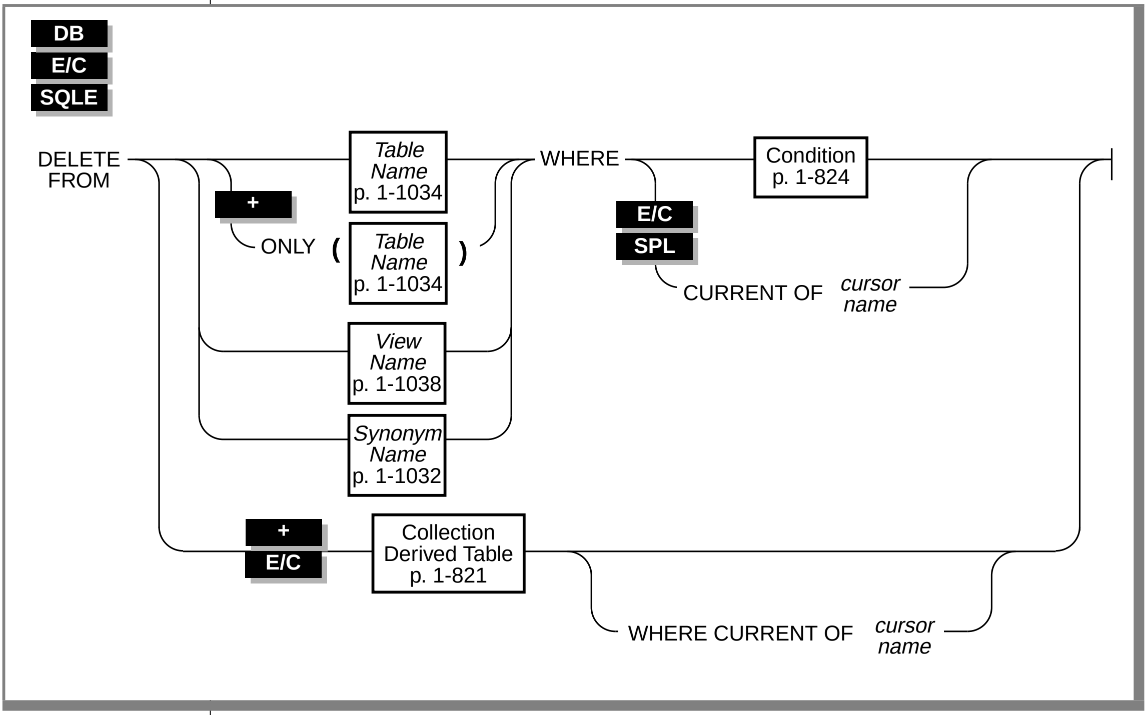
Element	Description
	The vertical line terminates the syntax diagram.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available.
	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.
	A gate (/3\ ) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify *size* no more than three times.

(3 of 3)

How to Read a Syntax Diagram

Figure 2 shows a syntax diagram that uses many of the elements that are listed in the previous table.

Figure 2
Example of a Syntax Diagram



The three icons at the top left of this diagram indicate that you can construct this statement if you are using DB-Access, ESQL/C, or the SQL Editor. To use the diagram to construct a statement, begin at the far left with the keywords DELETE FROM. Then follow the diagram to the right, proceeding through the options that you want.

To construct a DELETE statement

1. You must type the words `DELETE FROM`.
2. If you are using DB-Access, ESQL/C, or the SQL Editor, you can delete a table, view, or synonym:
   - Follow the diagram by typing the table name, view name, or synonym, as desired. Refer to the appropriate segment for available syntax options.
   - You must type the keyword `WHERE`.
   - If 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 deleting a condition, go to the “Condition” segment on page 1-803.
   - If you are using ESQL/C or SPL, you can include either the Condition clause to delete a specific condition or the CURRENT OF *cursorname* clause to delete a row from the table.
3. If you are using ESQL/C, you can also choose to delete from a collection-derived table:
   - Follow the diagram by going to the segment “Collection Derived Table” on page 1-800. Follow the syntax for the segment.
   - You can stop, taking the direct route to the terminator, or you can include the WHERE CURRENT OF *cursorname* clause to delete a row from a collection-derived table.
4. Follow the diagram to the terminator. Your DELETE statement is complete.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single 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 stores7
...

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 the Query-language option of 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: *Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.*

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

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes

On-Line Manuals

A CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see either the installation guide for your product or the installation insert that accompanies the documentation CD.

The documentation set that is provided on the CD describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

Printed Manuals

The Universal Server documentation set describes Universal Server, its implementation of SQL, and its associated application-programming interfaces. For an overview of the manuals in the Universal Server documentation set, see *Getting Started with INFORMIX-Universal Server*.

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com.

Please provide the following information:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). For a detailed description of these scripts, see the Introduction to the *Informix Error Messages* manual.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following on-line files, located in the **$INFORMIXDIR/release/en_us/0333** directory, supplement the information in this manual.

On-Line File	Purpose
SQLSDOC_9.1	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
SERVERS_9.1	The release-notes file describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.
IUNIVERSAL_9.1	The machine-notes file describes any special actions that are required to configure and use Informix products on your computer. Machine notes are named for the product described.

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

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. 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, on INFORMIX-Universal Server. In addition, many features of Universal Server comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about corrections or clarifications that you would find useful. Include the following information:

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

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send email, our address is:

doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

415-926-6571

We appreciate your feedback.

Chapter

SQL Statements

1

This chapter provides comprehensive reference information about SQL statements and the SQL segments that recur in SQL statements. It is organized into the following sections:

- "How to Enter SQL Statements" shows how to use the information in the statement descriptions to enter SQL statements correctly.
- "How to Enter SQL Comments" shows how to enter comments for your SQL statements in DB-Access command files, SQL APIs, and stored procedures.
- "Categories of SQL Statements" divides SQL statements into several functional categories and lists the statements within each category. Some examples of these categories are data definition statements, data manipulation statements, and data integrity statements.
- "ANSI Compliance and Extensions" explains how the SQL statements in this manual comply with the ANSI SQL standard. This section provides a list of ANSI-compliant statements, a list of ANSI-compliant statements with Informix extensions, and a list of statements that are Informix extensions to the ANSI standard.
- "Statements" gives comprehensive descriptions of SQL statements. The statements are listed in alphabetical order.
- "Segments" gives comprehensive descriptions of SQL segments. The segments are listed in alphabetical order. SQL segments are language elements, such as table names and expressions, that occur in many SQL statements. Instead of describing each segment in each statement where it occurs, this chapter provides a comprehensive stand-alone description of each segment. Whenever a segment occurs within the syntax diagram for an SQL statement, the diagram points to the stand-alone description of the segment for further information.

The following table summarizes the sections of this chapter.

Section	Starting Page	Scope
"How to Enter SQL Statements"	1-6	This section shows how to use the statement descriptions to enter SQL statements correctly.
"How to Enter SQL Comments"	1-9	This section shows how to enter comments for SQL statements.
"Categories of SQL Statements"	1-12	This section lists SQL statements by functional category.
"ANSI Compliance and Extensions"	1-17	This section lists SQL statements by degree of ANSI compliance.
"Statements"	1-19	This section gives reference descriptions of all SQL statements.
"Segments"	1-821	This section gives reference descriptions of all SQL segments.

How to Enter SQL Statements

The purpose of the statement descriptions in this chapter is to help you to enter SQL statements successfully and to understand the behavior of the statements. Each statement description 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 consists of multiple clauses, the statement description provides the same set of information for each clause.

Each statement description concludes with references to related information in this manual and other manuals.

The major aids for entering SQL statements successfully 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 "Syntax Conventions" on page 12 of the Introduction. This section is the key to understanding the syntax diagrams in the statement descriptions.

The "Syntax Conventions" section explains the elements that can appear in a syntax diagram and the paths that connect the elements to each other. This section also includes a sample syntax diagram that illustrates the major elements of all syntax diagrams. The narrative that follows the sample diagram shows how to read the diagram in order to enter the statement successfully.

When a syntax diagram within a statement description includes input parameters, the syntax diagram is followed by a syntax table that shows how to enter the parameters without generating errors. Each syntax table includes the following columns:

- The **Elements** column lists the name of each parameter as it appears in the syntax diagram.
- The **Purpose** column briefly states the purpose of the parameter. If the parameter has a default value, it is listed in this column.
- The **Restrictions** column summarizes the restrictions on the parameter, such as acceptable ranges of values.
- The **Syntax** column refers to the SQL segment that gives the detailed syntax for the parameter.

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 particular 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 "Sample-Code Conventions" on page 18 of the Introduction.

Using References

For help in understanding the concepts and terminology in the SQL statement description, check the "References" section at the end of the description.

The "References" section points to related information in this manual and other manuals that helps you to understand the statement in question. This 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
- The chapters in the *Informix Guide to SQL: Tutorial* that provide a task-oriented discussion of topics in this statement

Tip: *If you do not have extensive knowledge and experience with SQL, the "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. Your comments can help you or others to understand the role of the statement within a program, stored procedure, or command file. The code examples in this manual sometimes include comments that clarify the role of an SQL statement within the code.

The following table shows the SQL comment symbols that you can enter in your code. A *Y* in a column signifies that you can use the symbol with the product or database type named in the column heading. An *N* in a column signifies that you cannot use the symbol with the product or database type that the column heading names.

Comment Symbol	SQL APIs	Stored Procedures (SPL)	DB-Access	ANSI-Compliant Databases	Databases That Are Not ANSI Compliant	Description
double dash (--)	Y	Y	Y	Y	Y	The double dash precedes the comment. The double dash can comment only a single line. To comment more than one line, you must put the double dash at the beginning of each comment line.
curly brackets ({})	N	Y	Y	Y	Y	Curly brackets enclose the comment. The { precedes the comment, and the } follows the comment. You can use curly brackets for single-line comments or for multiple-line comments.

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

- The double dash (--) complies with the ANSI SQL standard.
- Curly brackets ({}) are an Informix extension to the standard.

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

DB

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 *Informix Guide to SQL: Tutorial*. For information on creating and modifying command files with the SQL editor or a system editor in DB-Access, see the *DB-Access User Manual*. ♦

SPL

You can use either comment symbol in any line of a SPL routine. See the discussion of commenting and documenting a procedure in the *Informix Guide to SQL: Tutorial*. ♦

ESQL

You can use the double dash (--) to comment SQL statements in your SQL API. For further information on the use of SQL comment symbols and language-specific comment symbols in application programs, see the manual for your SQL API. ♦

Examples of SQL Comment Symbols

Some simple examples can help to illustrate the different ways of using the SQL comment symbols.

Examples of the Double-Dash Symbol

The following example shows the use of the double dash (--) to comment an SQL statement. In this example, the comment appears on the same line as the statement.

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

In the following example, the user enters the same SQL statement and the same comment as in the preceding example, but the user 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
```

Examples of the Curly-Brackets Symbols

DB

SPL

The following example shows the use of curly brackets ({}) to comment an SQL statement. In this example, the comment appears on the same line as the statement.

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

In the following example, the user enters 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 enters a multiple-line comment:

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

♦

Non-ASCII Characters in SQL Comments

GLS

You can enter non-ASCII characters (including multibyte characters) in SQL comments if your locale supports a code set with the non-ASCII characters. For further information on the GLS aspects of SQL comments, see the *Guide to GLS Functionality*. ♦

Categories of SQL Statements

SQL statements are divided into the following categories:

- Access method statements
- Auxiliary statements
- Client / server connection statements
- Cursor manipulation statements
- Data access statements
- Data definition statements
- Data integrity statements
- Data manipulation statements
- Dynamic management statements
- Query optimization information statements
- Routine definition statements
- SPL statements

The specific statements for each category are listed below.

Access Method Statements

ALTER ACCESS_METHOD (See the *Virtual-Table Interface Programmer's Manual*.)
CREATE ACCESS_METHOD (See the *Virtual-Table Interface Programmer's Manual*.)
CREATE OPCLASS
DROP ACCESS_METHOD (See the *Virtual-Table Interface Programmer's Manual*.)
DROP OPCLASS

Auxiliary Statements

INFO
OUTPUT
GET DIAGNOSTICS
WHENEVER

Client/Server Connection Statements

CONNECT
DISCONNECT
SET CONNECTION

Cursor Manipulation Statements

CLOSE
DECLARE
FETCH
FLUSH
FREE
OPEN
PUT

Data Access Statements

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

Data Definition Statements

ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
CLOSE DATABASE
CREATE CAST
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE INDEX
CREATE OPAQUE TYPE
CREATE ROLE
CREATE ROW TYPE
CREATE SCHEMA
CREATE SYNONYM
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DATABASE
DROP CAST
DROP DATABASE
DROP INDEX
DROP ROLE
DROP ROW TYPE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP VIEW
RENAME COLUMN
RENAME DATABASE
RENAME TABLE

Data Integrity Statements

BEGIN WORK
CHECK TABLE
COMMIT WORK
CREATE AUDIT
DROP AUDIT
RECOVER TABLE
REPAIR TABLE
ROLLBACK WORK
ROLLFORWARD DATABASE
SET
SET LOG
START DATABASE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE

Data Manipulation Statements

DELETE
INSERT
LOAD
SELECT
UNLOAD
UPDATE

Dynamic Management Statements

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

Query Optimization Information Statements

SET EXPLAIN
SET OPTIMIZATION
SET PDQPRIORITY
UPDATE STATISTICS

Routine Definition Statements

CREATE FUNCTION
CREATE FUNCTION FROM
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROUTINE FROM
DROP FUNCTION
DROP PROCEDURE
DROP ROUTINE
EXECUTE FUNCTION
EXECUTE PROCEDURE

SPL Statements

CALL
CONTINUE
DEFINE
EXIT
FOR
FOREACH
LET
ON EXCEPTION
RAISE EXCEPTION
RETURN
SET DEBUG FILE TO
SYSTEM
TRACE
WHILE

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
ROLLBACK WORK
SET SESSION AUTHORIZATION
SET TRANSACTION

ANSI-Compliant Statements with Informix Extensions

CREATE SCHEMA
CREATE TABLE
CREATE VIEW
DECLARE
DELETE
EXECUTE
FETCH
GRANT
INSERT
OPEN
SELECT
SET CONNECTION
UPDATE
WHENEVER

Statements That Are Extensions to the ANSI Standard

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
BEGIN WORK

CLOSE DATABASE
CONNECT
CREATE CAST
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE FUNCTION FROM
CREATE INDEX
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROLE
CREATE ROUTINE
CREATE ROUTINE FROM
CREATE ROW TYPE
CREATE SYNONYM
CREATE TRIGGER
DATABASE
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DESCRIBE
DISCONNECT
DROP CAST
DROP DATABASE
DROP FUNCTION
DROP INDEX
DROP OPCLASS
DROP PROCEDURE
DROP ROLE
DROP ROW TYPE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP TYPE
DROP VIEW
EXECUTE FUNCTION
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS

GRANT FRAGMENT
INFO
LOAD
LOCK TABLE
OUTPUT
PREPARE
PUT
RENAME COLUMN
RENAME DATABASE
RENAME TABLE
REVOKE
REVOKE FRAGMENT
SET
SET DATASKIP
SET DEBUG FILE TO
SET DESCRIPTOR
SET EXPLAIN
SET ISOLATION
SET LOCK MODE
SET LOG
SET OPTIMIZATION
SET PDQPRIORITY
SET ROLE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE
UNLOAD
UNLOCK TABLE
UPDATE STATISTICS

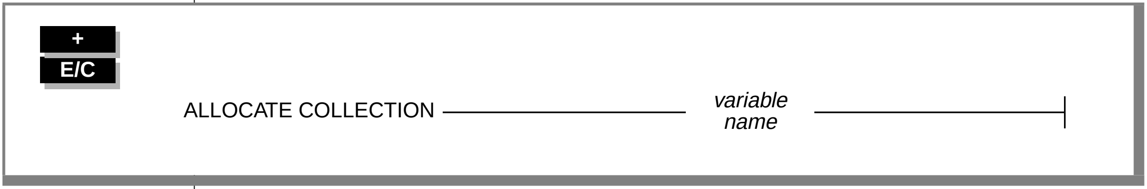
Statements

This section gives comprehensive reference descriptions of SQL statements. The statement descriptions appear in alphabetical order. For an explanation of the structure of statement descriptions, see "How to Enter SQL Statements" on page 1-6.

ALLOCATE COLLECTION

Use the ALLOCATE COLLECTION statement to allocate memory for an INFORMIX-ESQL/C **collection** variable.

Syntax



Element	Purpose	Restrictions	Syntax
variable name	Variable name that identifies a typed or untyped **collection** variable for which to allocate memory	Variable must contain the name of an unallocated ESQL/C **collection** host variable.	Name must conform to language-specific rules for variable names.

Usage

The ALLOCATE COLLECTION statement creates a place in memory for the data in the **collection** variable that *variable name* identifies. To create a **collection** variable for an ESQL/C program, perform the following steps:

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.

3. Populate the **collection** variable with elements.

   If you wish to modify elements into an existing collection, select the existing elements of the collection column into a **collection** variable with the SELECT statement (with no Collection Derived Table clause).

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;
...
EXEC SQL deallocate 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;
...
EXEC SQL deallocate collection :a_typed_set;
```

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

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

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

References

See the ALLOCATE ROW and DEALLOCATE COLLECTION statements in this manual.

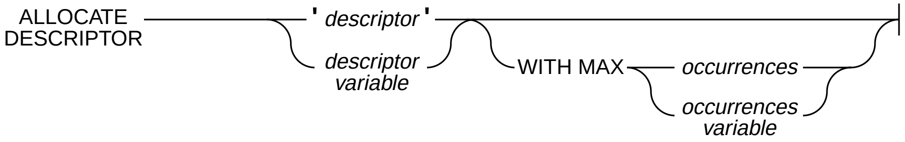
In the *Informix Guide to SQL: Tutorial*, see the discussion of **collection** data types in Chapter 10, “Understanding Complex Data Types.” In the *INFORMIX-ESQL/C Programmer’s Manual*, see the chapter that discusses complex data types.

ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate memory for a system-descriptor area.

Syntax

+ ESQL



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	Use single quotes. String must represent the name of an unallocated system-descriptor area.	Quoted String, p. 1-1010
descriptor variable	Host-variable name that identifies a system-descriptor area	Variable must contain the name of an unallocated system-descriptor area.	Name must conform to language-specific rules for variable names.
occurrences	The number of item descriptors in the system-descriptor area	Value must be unsigned INTEGER. Default value is 100.	Literal Number, p. 1-997
occurrences variable	Host variable that contains the number of *occurrences*	Data type must be INTEGER or SMALLINT.	Name must conform to language-specific rules for variable names.

Usage

The ALLOCATE DESCRIPTOR statement creates a place in memory for a system-descriptor area. The *descriptor* parameter or the *descriptor variable* parameter identifies this area. 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.

SPL

A DESCRIBE...USING SQL DESCRIPTOR statement also obtains information for the stored functions. For more information about stored functions, see the DESCRIBE statement on page 1-335 and Chapter 2, "SPL Statements." ♦

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 type, length, scale, precision, and nullability. Initially, all fields in the item-descriptor area are undefined.

The WITH MAX clause of ALLOCATE DESCRIPTOR sets the COUNT field to the number of occurrences that you specified in the *occurrences* parameter or the *occurrences variable* parameter. The DESCRIBE...USING SQL DESCRIPTOR statement sets other fields in the system-descriptor area. For more information, see "USING SQL DESCRIPTOR Clause" on page 1-338.

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 optional WITH MAX clause to indicate the number of item descriptors you need. Either the *occurrences* parameter or the *occurrences variable* parameter specifies the number of item descriptors that you want in the system-descriptor area. This number must be greater than zero. When you do not specify the WITH MAX clause, the database server uses a default value of `100` for the *occurrences* parameter.

The following examples show the ALLOCATE DESCRIPTOR statement that includes the WITH MAX clause. The first line uses an embedded variable name to identify the system-descriptor area and 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;
```

References

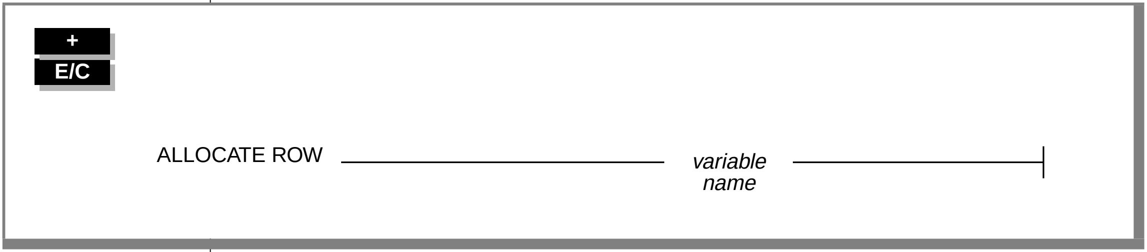
See the DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of system-descriptor areas in Chapter 5.

ALLOCATE ROW

Use the ALLOCATE ROW statement to allocate memory for an INFORMIX-ESQL/C **row** variable.

Syntax



Element	Purpose	Restrictions	Syntax
variable name	Variable name that identifies a typed or untyped **row** variable for which to allocate memory	Variable must contain the name of an unallocated ESQL/C **row** host variable.	Name must conform to language-specific rules for variable names.

Usage

The ALLOCATE ROW statement creates a place in memory for data in the **row** variable that *variable name* identifies. To create a **row** variable, perform the following steps in 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.

3. Populate the **row** variable with field values.

   Select the elements of an existing row-type column into a **row** variable with the SELECT statement (with no Collection Derived Table clause).

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;
. . .
EXEC SQL deallocate row :a_row;
```

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

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

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.

References

See the ALLOCATE COLLECTION and DEALLOCATE ROW statements in this manual.

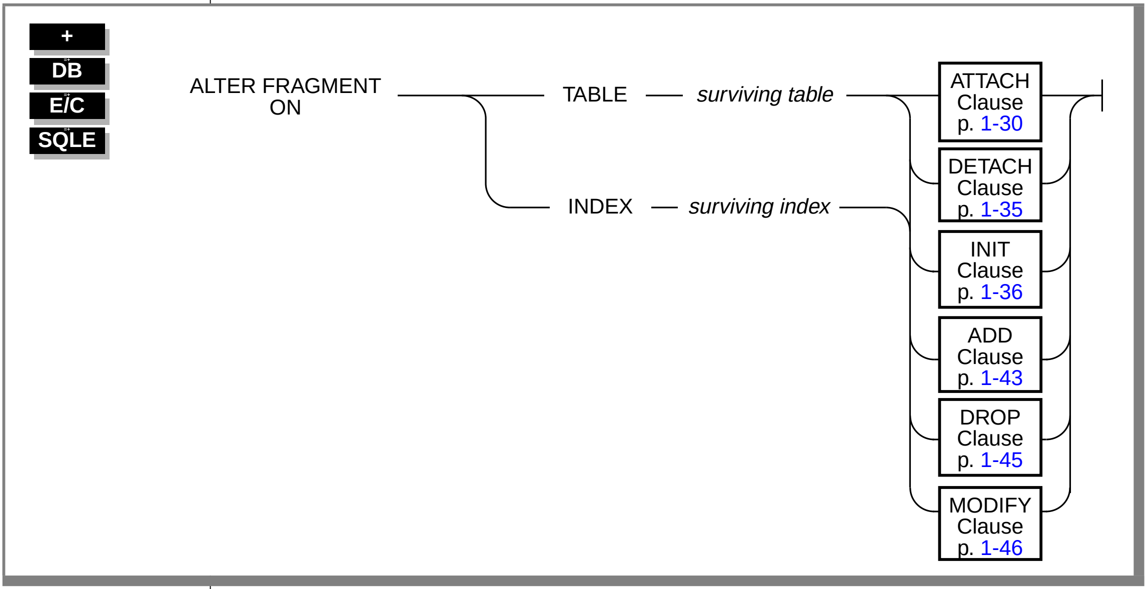
In the *Informix Guide to SQL: Tutorial*, see the discussion of rows in Chapter 10. In the *INFORMIX-ESQL/C Programmer's Manual*, see the chapter that discusses complex types.

ALTER FRAGMENT

Use the ALTER FRAGMENT statement to alter the fragmentation strategy of an existing table or index or to fragment an existing nonfragmented table.

Important: *You cannot use ALTER FRAGMENT on a typed table.*

Syntax



Element	**Purpose**	**Restrictions**	**Syntax**
surviving index	The index on which you execute the ALTER FRAGMENT statement	The index must exist at the time you execute the statement. All indexes are detached. You cannot alter an index to become attached or detached.	Index Name, p. 1-980
surviving table	The table on which you execute the ALTER FRAGMENT statement	The table must exist at the time you execute the statement.	Table Name, p. 1-1044

Usage

You can alter the fragmentation strategy of an existing table or index, or you can create a fragmentation strategy for nonfragmented tables. Use the ALTER FRAGMENT statement to tune your fragmentation strategy.

The clauses of the ALTER FRAGMENT statement let you perform the following tasks.

Clause	Purpose
ATTACH	Combines tables that contain identical table structures into a single fragmented table.
DETACH	Detaches a table fragment from a fragmentation strategy and places it in a new table.
INIT	Defines and initializes a new fragmentation strategy on a nonfragmented table or index, or modifies an existing fragmentation strategy. You can also use this clause to change the order of evaluation of fragment expressions.
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.

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.

INIT and ATTACH are the only operations that you can perform for tables that are not already fragmented.

You cannot use the ALTER FRAGMENT statement on a temporary table or a view.

How Is the ALTER FRAGMENT Statement Executed?

If your database uses logging, the ALTER FRAGMENT statement 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).

Making More 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.

  For more information about the **ontape** utility to start and stop logging, see the *INFORMIX-Universal Server Administrator's Guide*.
- Split the operations into multiple ALTER FRAGMENT statements, moving a smaller portion of records at each time.

For information about log-space requirements and disk-space requirements, refer to the *INFORMIX-Universal Server Administrator's Guide*. That guide also contains detailed instructions about how to turn off logging.

Determining the Number of Rows in the Fragment

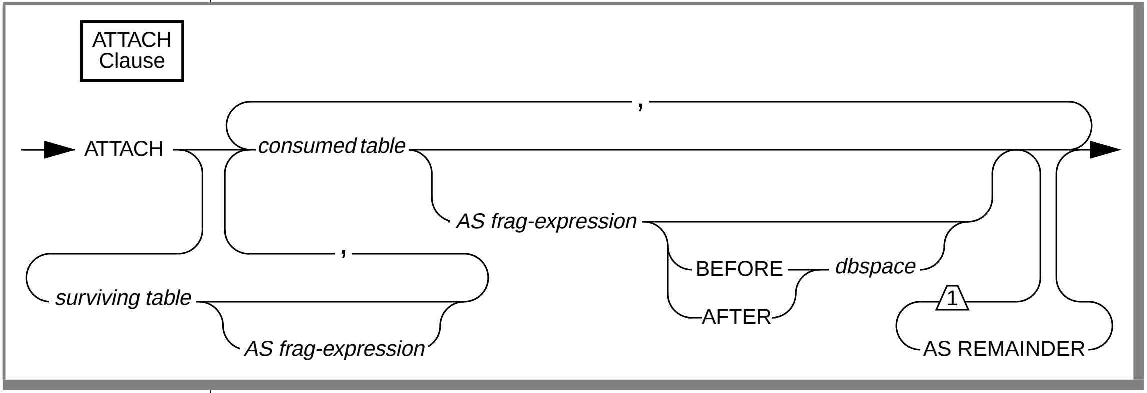
You can place as many rows into a fragment as the available space in the dbspace allows. To find out how many rows are in a fragment, perform these steps:

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

Important: *Use the CREATE TABLE statement or the ALTER FRAGMENT INIT statement to create fragmented tables.*

Use the ATTACH clause to combine tables that contain identical table structures into a fragmentation strategy. Transforming tables with identical table structures into fragments in a single table allows the database server to manage the fragmentation instead of the application managing the fragmentation. The distribution scheme can be either round-robin or expression based.



Element	Purpose	Restrictions	Syntax
consumed table	A nonfragmented table on which you execute the ATTACH clause	The table must exist at the time you execute the statement. No serial columns, referential constraints, primary-key constraints, or unique constraints are allowed in the table. The table can have check constraints and not-null constraints, but these constraints are dropped after the ATTACH clause is executed.	Table Name, p. 1-1044
dbspace	The dbspace name that specifies where the consumed table expression occurs in the fragmentation list	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
frag-expression	An expression that defines a fragment using a range, hash, or arbitrary rule	The *frag-expression* element can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-831
surviving table	The fragmented table that survives the execution of ALTER FRAGMENT	The table must exist at the time you execute the statement. No referential constraints, primary-key constraints, unique constraints, check constraints, or not-null constraints are allowed in the table.	Table Name, p. 1-1044

(2 of 2)

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

You must be the DBA or the owner of the tables that are involved to use the ATTACH clause.

After the tables are attached, the consumed table that is specified on the ATTACH clause no longer exists. The records that were in the consumed table must be referenced through the surviving table that is specified in the ALTER FRAGMENT ON TABLE statement.

Each table that is described in the ATTACH clause must be identical in structure; that is, all column definitions must match. The number, names, data types, and relative position of the columns must be identical. However, you cannot attach tables that contain serial columns. In addition, indexes and triggers on the surviving table survive the ATTACH, but indexes and triggers on the consumed table are dropped. Triggers are not activated with the ATTACH clause.

Tip: *In Universal Server, all indexes are detached.*

Combining Identically Structured Nonfragmented Tables

To make a single, fragmented table from two or more nonfragmented tables, the ATTACH clause must contain the surviving table as the first element of the *attach list*. The attach list is the list of tables in the ATTACH clause. For example, if you attach the tables **cur_acct** and **new_acct,** which were previously created in separate dbspaces, the surviving table **cur_acct** must be the first element in the attach list. The following statement illustrates this rule:

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH cur_acct, new_acct
```

If you want a new rowid column on the single fragmented table, attach all tables first and then add the rowid with the ALTER TABLE statement.

Attaching a Nonfragmented 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. The following example shows how to attach a nonfragmented table, **old_acct**, which was previously created in **dbsp3**, to a fragmented table, **cur_acct**:

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

BEFORE and AFTER Clauses

The BEFORE and AFTER clauses allow you to place a new fragment in a dbspace either before or after an existing dbspace. Use the BEFORE and AFTER clauses only when the distribution scheme is expression based (not round-robin). Attaching a new fragment without an explicit BEFORE or AFTER clause places the added fragment at the end of the fragmentation list. You cannot attach a new fragment after the remainder fragment.

Using ATTACH to Fragment Tables: Round-Robin

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**.

Using ATTACH to Fragment Tables: Fragment Expression

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 have altered 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 Chapter 2 of the *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 1-41.

Warning: *When you specify a date value in a fragment expression, 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 the distribution scheme. When you specify a 2-digit year, the **DBCENTURY** environment variable can affect the distribution scheme and can produce unpredictable results. For more information on the **DBCENTURY** environment variable, see the "Informix Guide to SQL: Reference."*

What Happens to Columns That Contain Large Objects?

In every table that is named in the ATTACH clause, each column that contains a large object must have the same storage type. For example, if a TEXT column is in a blobspace, the same column in all tables must be in the same blobspace. If the TEXT column is in the tblspace, the same column must be in the tblspace in all tables.

What Happens to Indexes and Triggers?

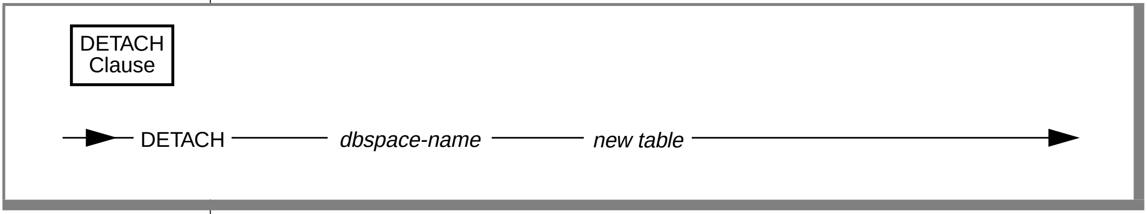
Unless you create separate index fragments, the index fragmentation is the same as the table fragmentation.

When you attach tables, any indexes or triggers that are defined on the consumed table no longer exist, and all rows in the consumed table (**new_acct**) are subject to the indexes and triggers that are defined in the surviving table (**cur_acct**). No triggers are activated with the ATTACH clause, but subsequent data manipulation operations on the "new" rows can fire triggers.

At the end of the ATTACH operation, indexes on the surviving table that were explicitly given a fragmentation strategy remain intact with that fragmentation strategy.

DETACH Clause

Use the DETACH clause to detach a table fragment from a distribution scheme and place the contents into a new nonfragmented table. For an explanation of distribution schemes, see "FRAGMENT BY Clause for Tables" on page 1-41.



Element	Purpose	Restrictions	Syntax
dbspace-name	The name of the dbspace that contains the fragment to be detached	The dbspace must exist when you execute the statement.	Identifier, p. 1-962
new table	The table that results after you execute the ALTER FRAGMENT statement	The table must not exist before you execute the statement.	Table Name, p. 1-1044

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.

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

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:

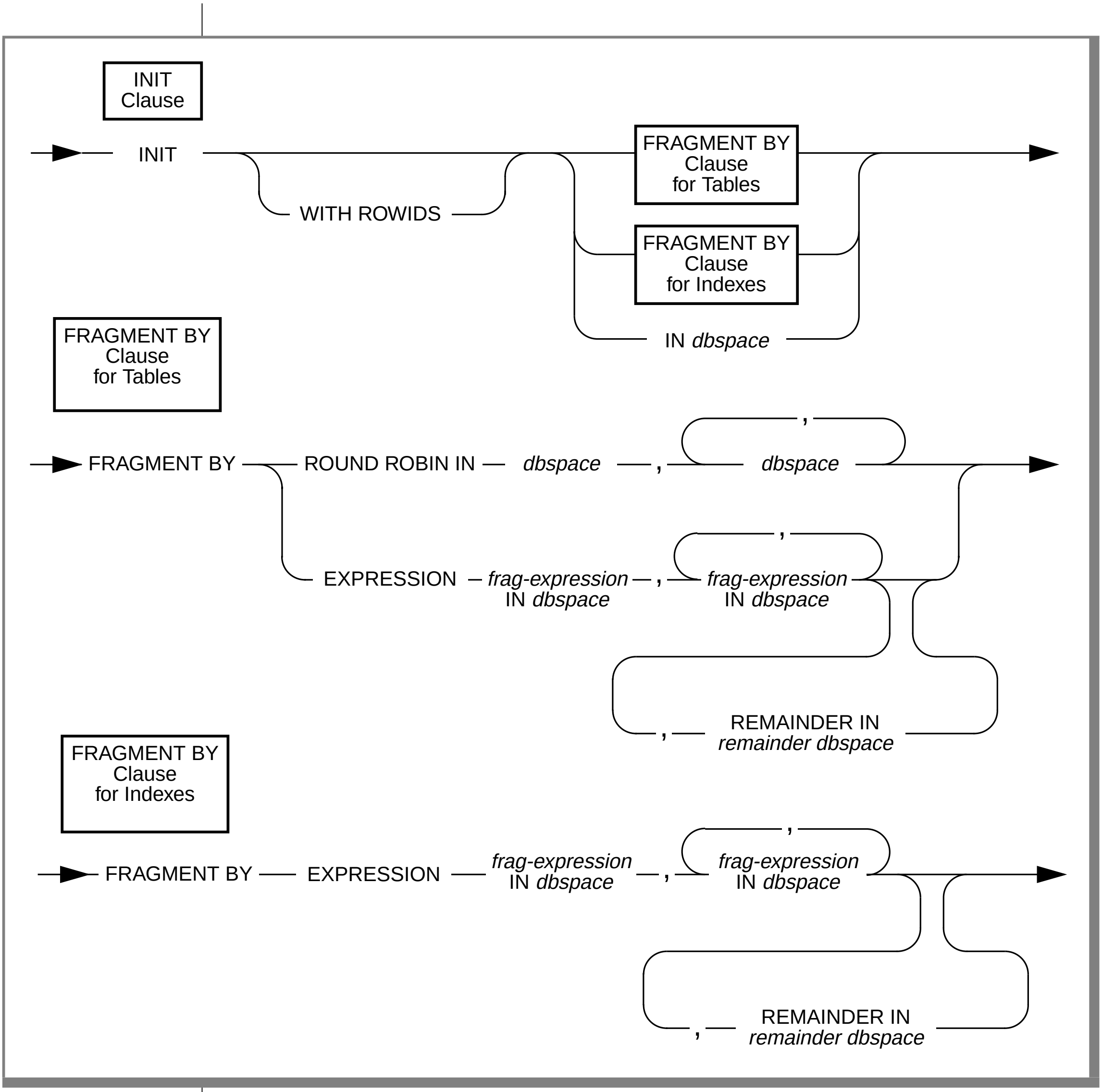
```
ALTER FRAGMENT ON TABLE bus_acct DETACH dbsp3 cli_acct
```

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.

INIT Clause

Use the INIT clause to perform the following functions:

- Change the fragmentation strategy on a single, fragmented table including changing the order of evaluating fragment expressions
- Define and initialize a new fragmentation strategy on a nonfragmented table
- Convert a fragmented table to a nonfragmented table


INIT Clause
INIT
WITH ROWIDS
FRAGMENT BY Clause for Tables
FRAGMENT BY Clause for Indexes
IN dbspace
FRAGMENT BY Clause for Tables
FRAGMENT BY
ROUND ROBIN IN
dbspace
,
dbspace
EXPRESSION
frag-expression IN dbspace
,
frag-expression IN dbspace
,
REMAINDER IN remainder dbspace
FRAGMENT BY Clause for Indexes
FRAGMENT BY
EXPRESSION
frag-expression IN dbspace
,
frag-expression IN dbspace
,
REMAINDER IN remainder dbspace

Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that contains the fragmented information	The dbspace must exist at the time you execute the statement. When you use the FRAGMENT BY clause, you must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 1-962
frag-expression	An expression that defines a fragment using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-831, and Expression, p. 1-876
remainder dbspace	The dbspace that contains data that does not meet the conditions defined in any fragment expression	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace that is specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-962

The INIT clause allows you to fragment an existing table or index that is not fragmented without redefining the table or index. With the INIT clause, you can also convert an existing fragmentation strategy on a table or index to another fragmentation strategy. Any existing fragmentation strategy is discarded, and records are moved to fragments as defined in the new fragmentation strategy. The INIT clause also allows you to convert a fragmented table or index to a nonfragmented table or index.

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.

Changing an Existing Fragmentation Strategy

You can redefine a fragmentation strategy if you decide that your initial strategy does not fulfill your needs. The following example shows the statement that originally defined the fragmentation strategy on the table **account** and then shows the 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
    MOD(col1, 3) = 0 in dbsp1,
    MOD(col1, 3) = 1 in dbsp2,
    MOD(col1, 3) = 2 in dbsp3;
```

When you want to redefine a fragmentation strategy, and any existing dbspaces are full, you must fragment the table in different dbspaces than the full dbspaces.

Fragmenting Unique and System Indexes

You can fragment unique indexes only if the table uses an expression-based distribution scheme. The columns that are 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.

System indexes (such as those used in referential constraints and unique constraints) utilize user indexes if the indexes 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 use the ALTER TABLE statement to add the constraint.

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 (that is, to rid the table of any fragmentation strategy), all indexes that are fragmented in the same way as the table become nonfragmented indexes. System indexes are not affected by the use of the INIT clause on the table.

Defining a Fragmentation Strategy on a Nonfragmented Table

You can use the INIT clause to define a fragmentation strategy on a nonfragmented table. It does not matter whether the table was created with a storage option. The following example shows the original table definition as well as how to use the ALTER FRAGMENT statement to fragment the table:

```
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;
```

WITH ROWIDS Clause

Nonfragmented tables contain a pseudocolumn called the rowid column. Fragmented tables do not contain this column unless it is explicitly created.

Use the WITH ROWIDS clause to add a new column called the rowid column. the database server assigns a unique number to each row that remains stable for the existence of the row. The database server creates an index that it uses to find the physical location of the row.After you add the WITH ROWIDS clause, each row contains an additional 4 bytes to store the rowid column.

You cannot use the WITH ROWIDS clause on typed tables.

Important: *Informix recommends that you use primary keys, rather than the rowid column, as an access method.*

FRAGMENT BY Clause for Tables

Use the FRAGMENT BY clause for tables to define the distribution scheme, which is either round-robin or expression based.

In a round-robin distribution scheme, specify at least two dbspaces where the fragments are 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 so 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.

In an expression-based distribution scheme, each fragment expression in a *rule* specifies a dbspace. The rule specifies how the database server determines the fragment into which a row is placed. Each fragment expression within the rule isolates data and aids the database server in searching for rows. You can specify one of the following rules:

- Range rule

  A range rule uses a range to specify which rows are placed in a fragment, as the following example shows:

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

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

  ```
  ...
      FRAGMENT BY EXPRESSION
      MOD(id_num, 3) = 0 IN dbsp1,
      MOD(id_num, 3) = 1 IN dbsp2,
      MOD(id_num, 3) = 2 IN dbsp3;
  ```

- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as the following example shows:

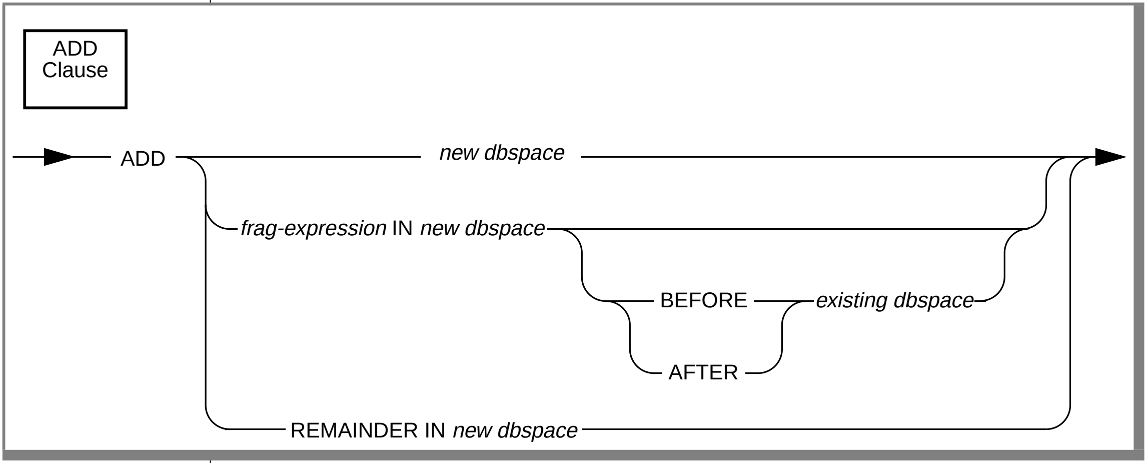
  ```
  ...
      FRAGMENT BY EXPRESSION
      zip_num = 95228 OR zip_num = 95443 IN dbsp2,
      zip_num = 91120 OR zip_num = 92310 IN dbsp4,
      REMAINDER IN dbsp5;
  ```

FRAGMENT BY Clause for Indexes

Use the FRAGMENT BY clause for indexes to define the expression-based distribution scheme. Like the FRAGMENT BY clause for tables, the FRAGMENT BY clause for indexes supports range rules, hash rules, and arbitrary rules. See "FRAGMENT BY Clause for Tables" on page 1-41 for an explanation of these rules.

ADD Clause

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



Element	Purpose	Restrictions	Syntax
existing dbspace	A dbspace name specified in an existing fragmentation list	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-962
frag-expression	The range, hash, or arbitrary expression that defines the added fragment	The *frag-expression* can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-831, and Expression, p. 1-876
new dbspace	The added dbspace in a round-robin distribution scheme	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-962

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 title)
FRAGMENT BY ROUND ROBIN in dbsp1, dbsp4;
```

To add another dbspace, use the ADD clause, as the following example shows:

```
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 shuffle records from some 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 following example shows the original expression definition:

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

If you want to add another fragment to the fragmentation list and have this fragment 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.

BEFORE and AFTER Clauses

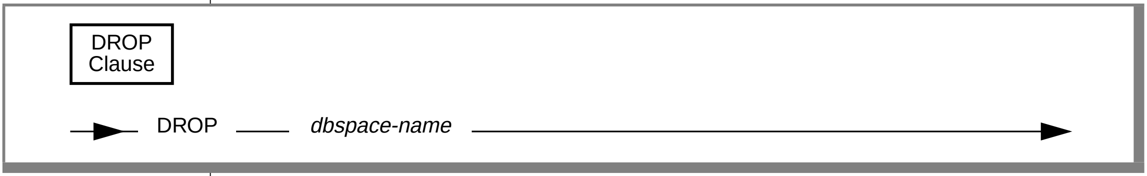
The BEFORE and AFTER clauses allow you to place a new fragment in a dbspace either before or after an existing dbspace. Use the BEFORE and AFTER clauses only when the distribution scheme is expression based (not round-robin). You cannot add a new fragment after the remainder fragment. Adding a new fragment without an explicit BEFORE or AFTER clause places the added fragment at the end of the fragmentation list. However, if the fragmentation list contains a REMAINDER clause, the added fragment is added before the remainder fragment (that is, the remainder remains the last item on the fragment list).

REMAINDER Clause

You cannot add a remainder fragment when one already exists. When you add a new fragment to the end of the fragmentation list, and a remainder fragment exists, the records in the remainder fragment are retrieved and re-evaluated. These records can be moved to the new fragment. The remainder fragment always remains the last item in the fragment list.

DROP Clause

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



Element	Purpose	Restrictions	Syntax
dbspace-name	The name of the dbspace that contains the dropped fragment	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-962

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 clause, 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.

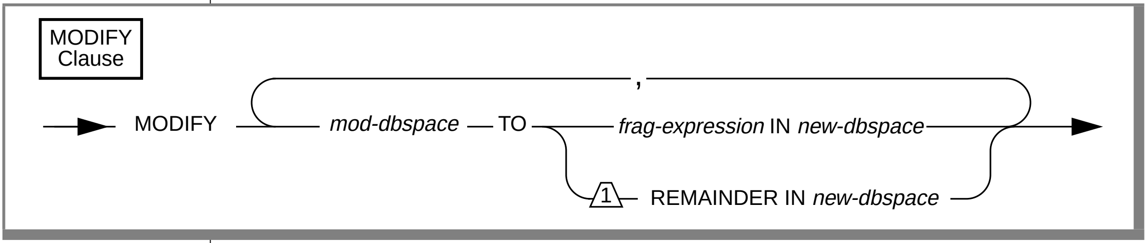
When you drop a fragment from a dbspace, the underlying dbspace is not affected. Only the fragment data within that dbspace is affected. When you drop a fragment all the records located in the fragment move to another fragment. The destination fragment might not have enough space for the additional records. When this happens, follow one of the procedures that are listed in "Making More Space" on page 1-29 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;
```

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
frag-expression	The modified range, hash, or arbitrary expression	The fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-831, and Expression, p. 1-876
mod-dbspace	The modified dbspace	The dbspace must exist when you execute the statement.	Identifier, p. 1-962
new-dbspace	The dbspace that contains the modified information	The dbspace must exist when you execute the statement.	Identifier, p. 1-962

General Usage

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

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

Changing the Expression in an Existing Dbspace

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*.

The following example shows how to use the MODIFY clause to change an existing expression:

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

Moving an Expression from One Dbspace to Another

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.

The following example shows how to use the MODIFY clause to move an expression from one dbspace to another:

```
ALTER FRAGMENT ON TABLE cust_acct
    MODIFY dbsp1 to acct_num < 35 in dbsp2
```

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

Changing the Expression and Moving It to a New Dbspace

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

References

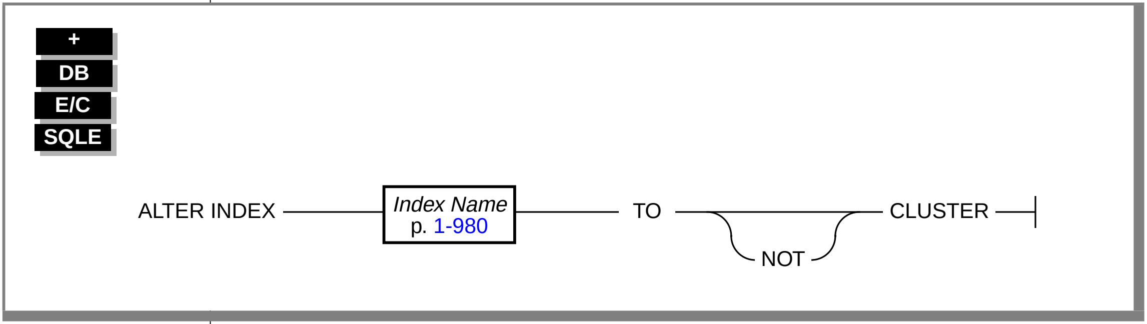
See the CREATE TABLE, CREATE INDEX, ALTER TABLE statements in this manual. Also see the Condition, Data Type, Expression, and Identifier segments.

For a task-oriented discussion of each clause in the ALTER FRAGMENT statement, see Chapter 9 of the *Informix Guide to SQL: Tutorial*.

ALTER INDEX

Use the ALTER INDEX statement to put the data in a table in the order of an existing index or to release an index from the clustering attribute.

Syntax



Usage

The ALTER INDEX statement works only on indexes that are created with the CREATE INDEX statement; it does not affect constraints that are created with the CREATE TABLE statement.

You cannot alter the index of a temporary table.

TO CLUSTER Option

The TO CLUSTER option causes the rows in the physical table to reorder in the indexed order.

The following 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;
```

Reordering causes rewriting the entire file. 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, the table is locked IN EXCLUSIVE MODE. When another process is using the table to which *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 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.

TO NOT CLUSTER Option

The NOT 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 NOT 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.

References

See the CREATE INDEX and CREATE TABLE statements in this chapter.

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of clustered indexes.

In the *Informix Guide to SQL: Tutorial*, see the discussion of data-integrity constraints and the discussion of the ON DELETE CASCADE clause in Chapter 4. Also see the discussion of creating a database and tables in Chapter 9.

See the SET statement in this manual for information on object modes.

ALTER TABLE

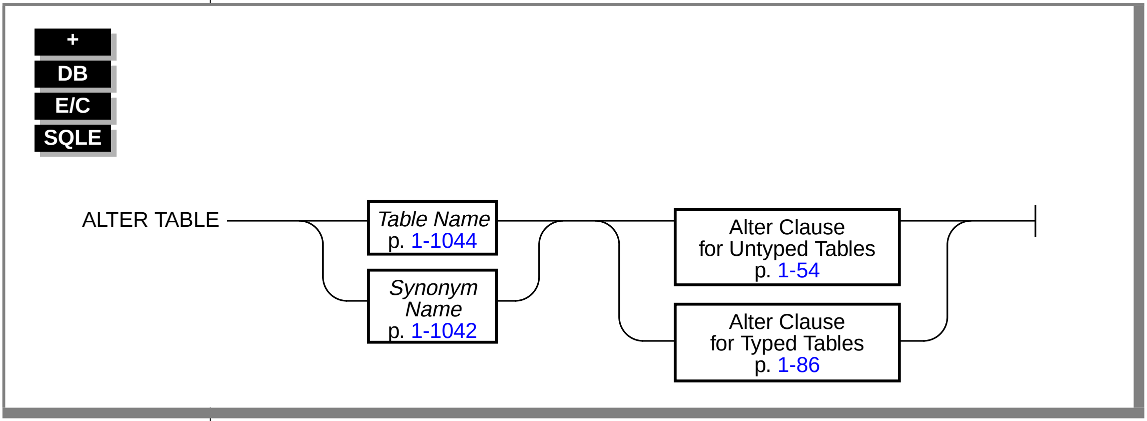
Use the ALTER TABLE statement to modify both typed and untyped tables.

You can add, modify, or drop the constraints that are placed on a column or composite list of columns or change the extent size. You can change an untyped table to a typed table or drop the type from a typed table.

For untyped tables, you can also add, drop, or modify a column from a table, and add or drop a rowid column for a fragmented table.

You cannot alter a temporary table.

Syntax



Usage

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

- You must have the DBA privilege on the database where the table resides.
- 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.

In addition to the basic privileges required for altering a table, you need the following privileges for specific operations:

- To add or drop a type, you must have the Usage privilege on the type.
- To drop a constraint in a database, 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.
- To add a referential constraint to an untyped table, you must have the DBA or References privilege on either the referenced columns or the referenced table.

When you add any kind of constraint, the name of the constraint must be unique within the database.

ANSI

When you add any kind of constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

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

Restrictions for Violations and Diagnostics Tables

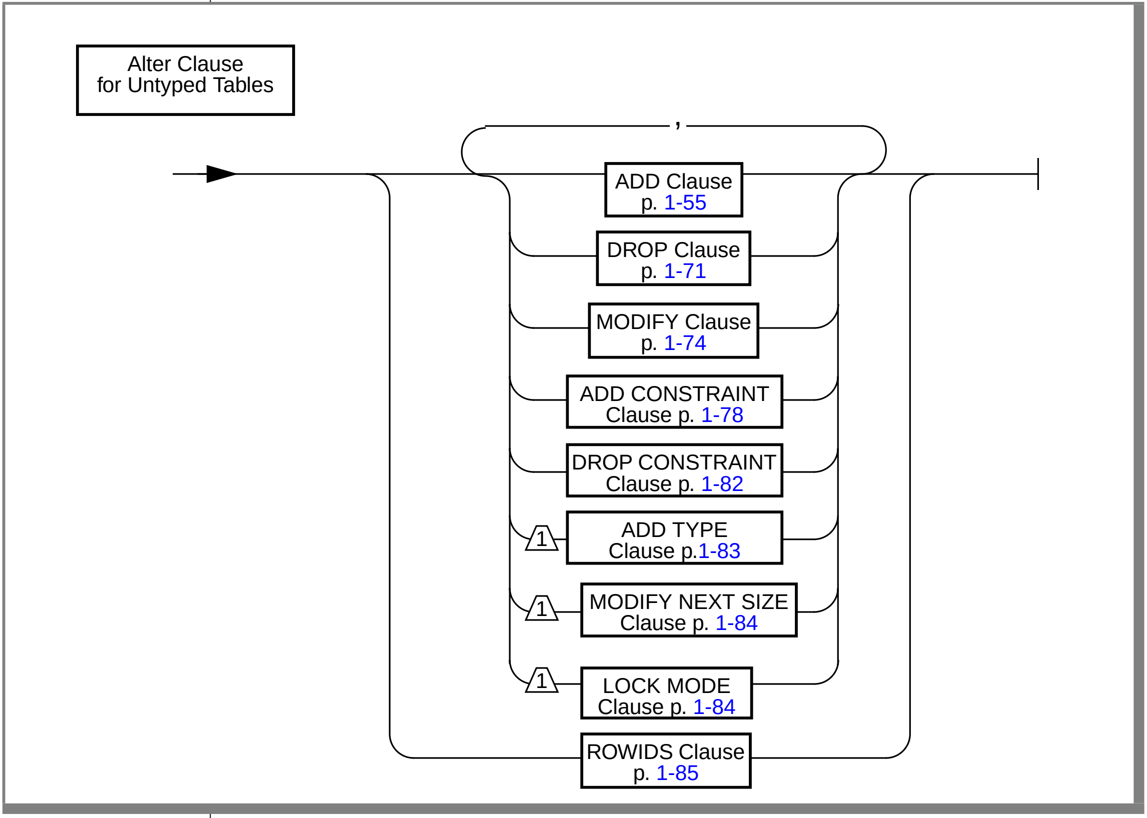
Keep the following considerations in mind when you use the ALTER TABLE statement in connection with violations and diagnostics tables:

- You cannot add, drop, or modify a column if the table that contains the column has violations and diagnostics tables associated with it.
- You cannot alter a violations or diagnostics table.
- You cannot add a constraint to a violations or diagnostics table.

See the START VIOLATIONS TABLE statement on page 1-744 for further information on violations and diagnostics tables.

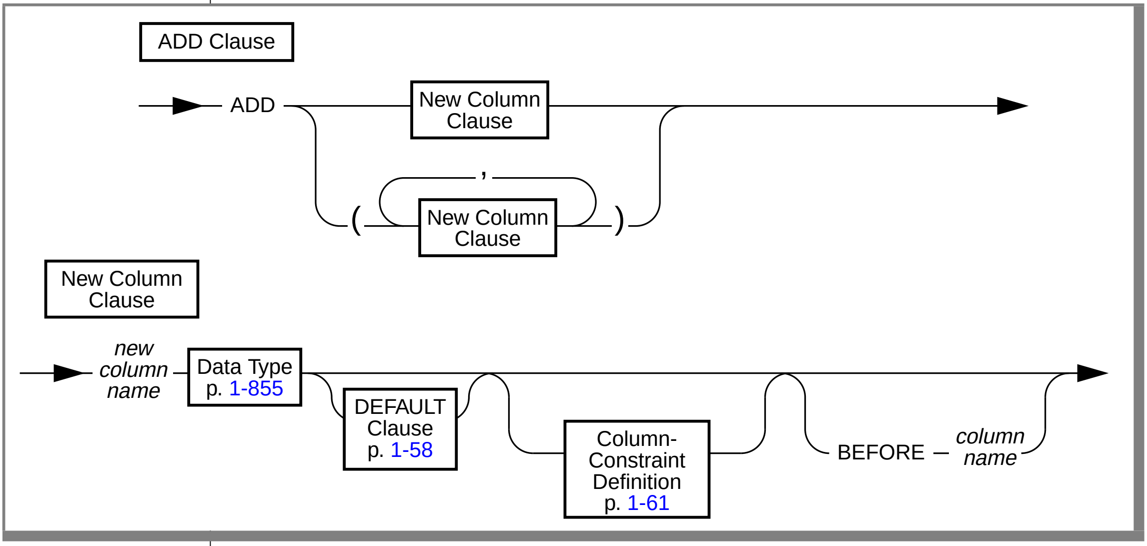
Alter Clause for Untyped Tables

The database server performs the actions in the Alter Clause in the order that you specify. If any of the actions fails, the entire operation is cancelled.



ADD Clause

Use the ADD clause to add a column to an existing untyped table. You cannot add a SERIAL or SERIAL8 column to a table if the table contains data.



Element	Purpose	Restrictions	Syntax
column name	The name of a column before which the new column is to be placed	The column must already exist in the table.	Identifier, p. 1-962
new column name	The name of the column that you are adding	This name must not be used for any existing columns in the table. You cannot add a SERIAL or SERIAL8 column if the table contains data.	Identifier, p. 1-962

The ADD clause appears in the Alter Clause for Untyped Table clause on page 1-54.

Algorithms for Adding Columns to Tables

INFORMIX-Universal Server uses the following two algorithms for adding columns to tables:

- If you execute an ALTER TABLE statement that adds a column or list of columns to the end of a table, the database server uses the in-place alter algorithm. This algorithm allows the database server to alter the table definition without making the table unavailable to users for longer than the time it takes to update the table definition. Furthermore, the physical addition of the new columns to the table definition occurs essentially in place as rows are updated, without requiring a second copy of the table to be created.
- If you execute an ALTER TABLE statement that does not add a column or list of columns to the end of a table, the database server uses a slower algorithm. When it uses this slower algorithm, the database server performs the alter operation by placing an exclusive lock on the table while it copies the table to be altered to a new table that contains the new table definition. After the copy operation is complete, the database server drops the older version of the table.

Tip: *To add a column to the end of a table, omit the BEFORE option from the ADD clause. When you do not specify a column before which the new column is to be added, the database server adds the new column to the end of the table by default.*

Scope of the In-Place Alter Algorithm

The database server uses the in-place alter algorithm if you specify the ADD clause without the BEFORE option and if you specify any clauses other than the following:

- The DROP clause
- A MODIFY clause that changes the data type of a column or changes the number of characters in a character column

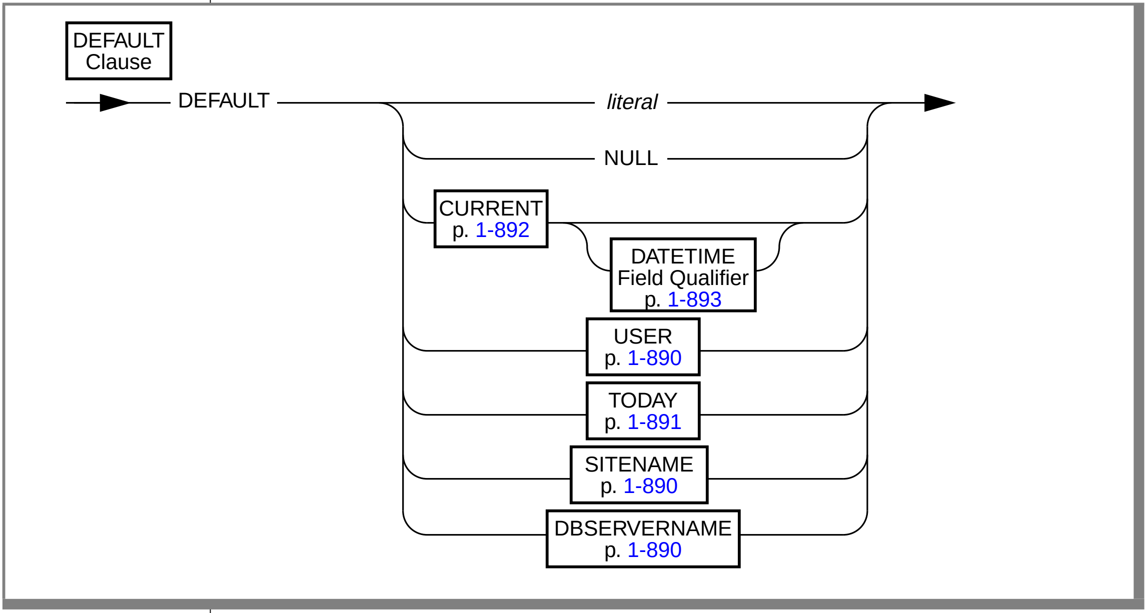
Benefits of the In-Place Alter Algorithm

The in-place alter algorithm lets you alter tables in place instead of creating a new table with the latest table definition and copying rows from the original table to the new table. The in-place alter method reduces the space that is required for altering tables and also increases the availability of the tables that are being altered.

The database server uses the slower algorithm for altering tables whenever your ALTER TABLE statement does not match the conditions for using the in-place alter algorithm. The database server uses the slower algorithm under the following conditions:

- The database server uses the slower algorithm if you specify an ADD clause with the BEFORE option.
- The database server uses the slower algorithm if you specify an ADD clause without the BEFORE option, but you also specify one of the following clauses:
  - The DROP clause
  - A MODIFY clause that changes the data type of a column or changes the number of characters in a character column

DEFAULT Clause



Element	Purpose	Restrictions	Syntax
literal	A literal term that defines alpha or numeric constant characters to be used as the default value for the column	Term must be appropriate type for the column. See "Literal Terms" on page 1-59.	Constant Expressions, p. 1-887

You can specify a default value that the database server inserts into the column when you do not specify an explicit value. When a default is not specified, and the column allows nulls, the default is NULL. When you designate NULL as the default value for a column, you cannot place a not-null constraint on the column.

You cannot place a default on SERIAL or SERIAL8 columns.

When the altered table already has rows in it, the new column contains the default value for all existing rows.

The DEFAULT clause appears in the ADD clause on page 1-55.

Literal Terms

You can designate *literal* terms as default values. Use a literal term to define alpha or numeric constant characters. To use a literal term as a default value, you must adhere to the rules in the following table.

Use a Literal	With Columns of Data Type
INTEGER	INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, SMALLFLOAT, INT8
DECIMAL	DECIMAL, MONEY, FLOAT, SMALLFLOAT
CHARACTER	CHAR, VARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, DATE
INTERVAL	INTERVAL
DATETIME	DATETIME
CHARACTER	Opaque data types

Characters must be enclosed in quotation marks. Date literals must be formatted in accordance with the DBDATE environment variable. When DBDATE is not set, the format *mm/dd/yyyy* is assumed.

Opaque data types support only string literals for default values. The default value must be specified at the column level and not at the table level.

For information on using a literal INTERVAL, see the Literal INTERVAL segment on page 1-994. For more information on using a literal DATETIME, see the Literal DATETIME segment on page 1-991.

Data-Type Requirements

The following table indicates the data type requirements for columns that specify the CURRENT, DBSERVERNAME, SITENAME, TODAY, or USER functions as the default value.

Function Name	Data Type Requirements
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 18 characters long
SITENAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 18 characters long
TODAY	DATE column
USER	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 18 characters long

You cannot designate a server-defined function (that is, CURRENT, USER, TODAY, SITENAME or DBSERVERNAME) as the default value for opaque or distinct data types.

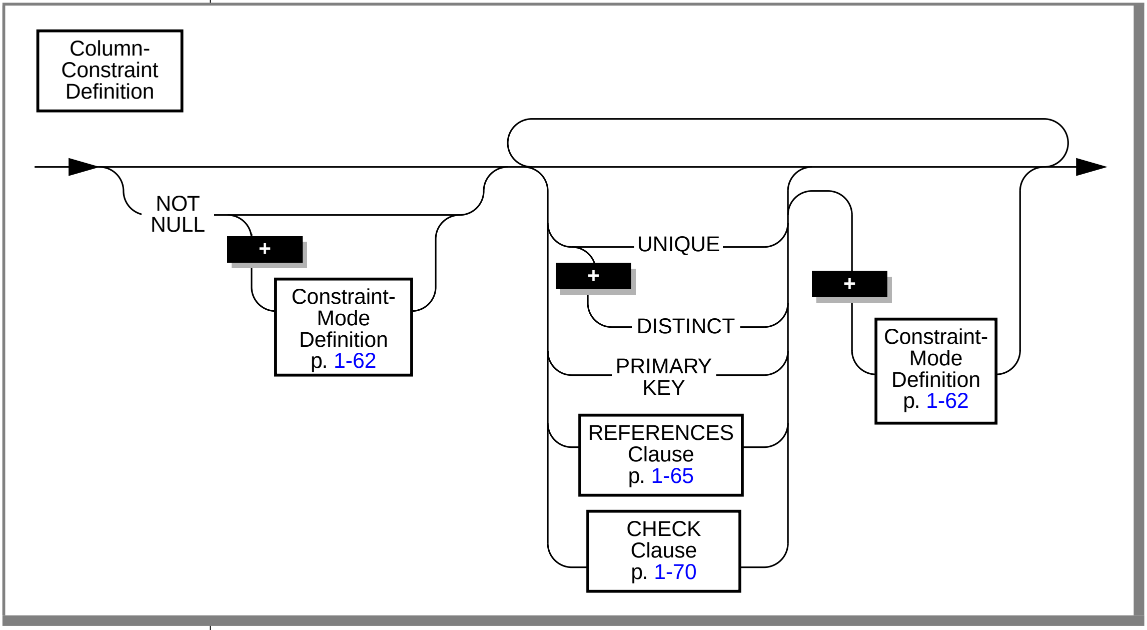
Example of a Literal Default Value

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 c**olumn.

Column-Constraint Definition


Column-Constraint Definition
NOT NULL
+
Constraint-Mode Definition p. 1-62
UNIQUE
+
DISTINCT
PRIMARY KEY
REFERENCES Clause p. 1-65
CHECK Clause p. 1-70
+
Constraint-Mode Definition p. 1-62

When 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 this case, if the not-null constraint is used, no default value exists for the column, and the column does not allow nulls. When the table contains data, however, you cannot specify a not-null constraint when you add a column (unless both the not-null constraint and a default value other than null are specified).

You cannot specify a unique or primary-key constraint on a new column if the table contains data. However, in the case of a unique constraint, 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.

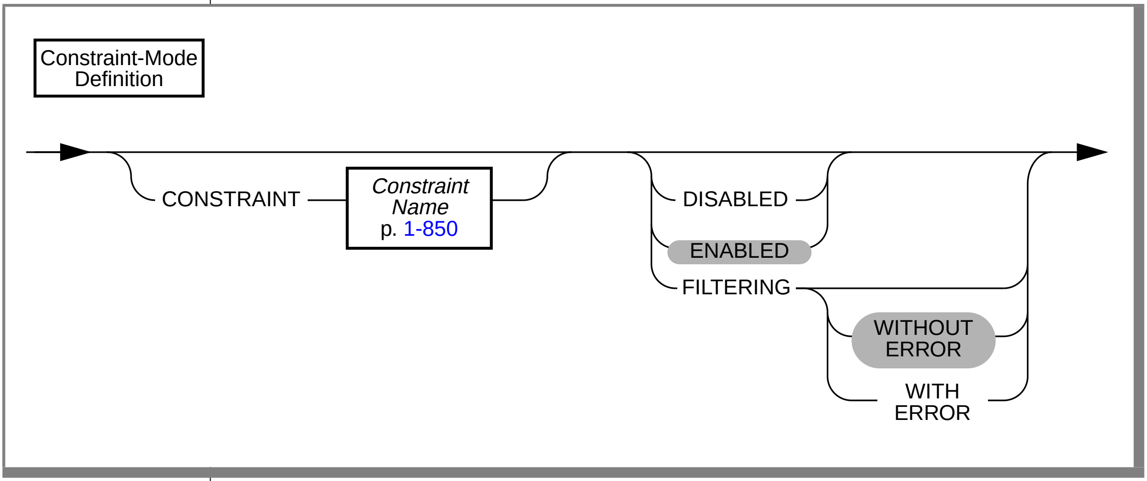
The following rules apply when you place unique or primary-key constraints on existing columns:

- When you place a unique or primary-key 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. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before you add the constraint.
- When you place a unique or primary-key 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 have a unique constraint on a BYTE or TEXT column, nor can you place referential or check constraints on these types of columns. A check constraint on a BYTE or TEXT column can check only for IS NULL, IS NOT NULL, or LENGTH.

The Column-Constraint Definition appears in the New Column clause on page 1-55.

Constraint-Mode Definition



You can use the Constraint-Mode Definition option for the following purposes:

- To assign a name to a constraint on a column
- To set a constraint to one of the following object modes: disabled, enabled, or filtering

The Constraint-Mode Definition appears in the Column-Constraint Definition on page 1-61.

Description of Constraint Modes

You can set constraints to the following modes: disabled, enabled, or filtering. These modes are described in the following table.

Constraint Mode	Effect
disabled	A constraint that is created in disabled mode is not enforced during insert, delete, and update operations.
enabled	A constraint that is created in enabled mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement fails.
filtering	A constraint that is created in filtering mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement continues processing, but the bad row is written to the violations table that is associated with the target table. Diagnostic information about the constraint violation is written to the diagnostics table that is associated with the target table.

If you chose the filtering mode, you can specify the WITHOUT ERROR options. The following table describes these options.

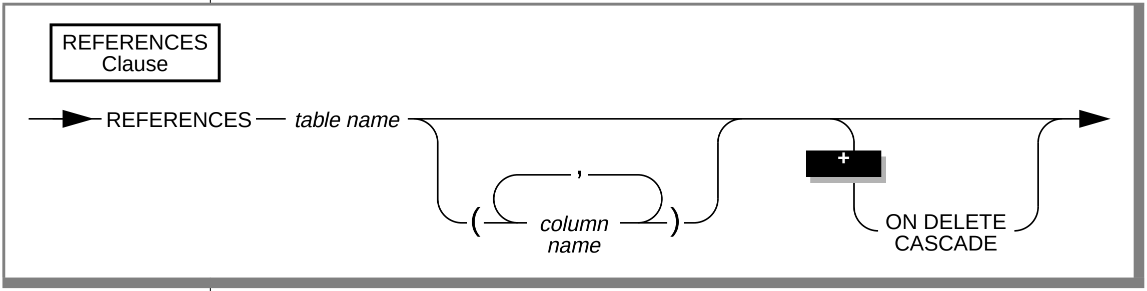
Error Option	Effect
WITHOUT ERROR	When a filtering mode constraint is violated during an insert, delete, or update operation, no integrity-violation error is returned to the user.
WITH ERROR	When a filtering mode constraint is violated during an insert, delete, or update operation, an integrity-violation error is returned to the user.

Using Constraint Modes

You must observe the following rules when you use constraint modes:

- If you do not specify the object mode of a column-level or table-level constraint explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering mode constraint, the default error option is WITHOUT ERROR.
- When you add a constraint to a table and specify the disabled object mode for the constraint, your ALTER TABLE statement succeeds even if existing rows in the table violate the constraint.
- When you add a column-level or table-level constraint to a table and specify the enabled or filtering object mode for the constraint, your ALTER TABLE statement succeeds if no existing rows in the table violate the new constraint. However, if any existing rows in the table violate the constraint, your ALTER TABLE statement fails and returns an error.
- When you add a column-level or table-level constraint to a table in the enabled or filtering object mode, and existing rows in the table violate the constraint, 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.

REFERENCES Clause



Element	Purpose	Restrictions	Syntax
column name	A referenced column or set of columns in the referenced table. 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.	You must observe restrictions on the number of columns you can specify, the data type of the columns, and the existing constraints on the columns. See "Restrictions on the REFERENCES Clause" on page 1-66.	Identifier, p. 1-962
table name	The name of the referenced table	The referenced table can be the same table as the referencing table, or it can be a different table in the same database.	Table Name, p. 1-1044

The REFERENCES clause appears in the Column-Constraint Definition on page 1-61.

Restrictions on the REFERENCES Clause

Observe the following restrictions on the referenced column (the column or set of columns that you specify in the *column name* variable).

The following restrictions apply to the number of columns that you can specify in the *column name* variable:

- The number of referenced columns in the referenced table must match the number of referencing columns in the referencing table.
- If you are using the REFERENCES clause within the ADD or MODIFY clauses, you can specify only one column in the *column name* variable.
- If you are using the REFERENCES clause within the ADD CONSTRAINT clause, you can specify one column or multiple columns in the *column name* variable.
- The maximum number of columns and the total length of columns vary with the database server:

  You can specify a maximum of 16 column names. The total length of all the columns cannot exceed 390 bytes.

The data type of each referenced column must be identical to the data type of the corresponding referencing column. The only exception is that a referencing column must be INTEGER or INT8 if the referenced column is SERIAL or SERIAL8.

The referenced column or set of columns must be a unique or primary-key column. That is, the referenced column in the referenced table must already have a unique or primary-key constraint placed upon it.

Using the REFERENCES Clause in ALTER TABLE

Use the REFERENCES clause to reference a column or set of columns in another table or the same table. When you are using the ADD or MODIFY clause, you can reference a single column. When you are using the ADD CONSTRAINT clause, you can reference a single column or a set of columns.

The table that is referenced in the REFERENCES clause must reside in the same database as the altered table.

A referential constraint establishes the relationship between columns in two tables or within the same table. The relationship between the columns is commonly called a *parent-child* relationship. For every entry in the child (referencing) columns, a matching entry must exist in the parent (referenced) columns.

The referenced column (parent or primary-key) must be a column that is a unique or primary-key constraint. When you specify a column in the REFERENCES clause that does not meet this criterion, the database server returns an error.

The referencing column (child or foreign key) that you specify in the Add Column clause can contain null or duplicate values, but every value (that is, all foreign-key columns that contain non-null values) in the referencing columns must match a value in the referenced column.

Relationship Between Referencing and Referenced Columns

A referential constraint has a one-to-one relationship between referencing and referenced columns. If the primary key is a set of columns, the foreign key also must be a set of columns that corresponds to the primary key. 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 reference a primary key in another table, you do not have to explicitly state the primary-key columns in that table. Referenced tables that do not specify the referenced column default to the primary-key column. In the previous example, because **order_num** is the primary key in the **orders** table, you do not have to reference that column explicitly.

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.

The data types of the referencing and referenced column must be identical, unless the primary-key column is SERIAL or SERIAL8 data type. When you add a column that references a SERIAL of SERIAL8 column, the column that you add must be an INTEGER or an INT8 column.

Locks Held During Creation of a Referential Constraint

When you create a referential constraint, an exclusive lock is placed 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 a table in a database with transactions, and you are using transactions).

Using ON DELETE CASCADE

Cascading deletes allow you to specify whether you want rows deleted in the child table when rows are deleted in the parent table. Normally, you cannot delete data in the parent table if child tables are associated with it. You can specify that you want the rows in the child table deleted with ON DELETE CASCADE. With ON DELETE CASCADE (or cascading deletes), when you delete a row in the parent table, any rows that are associated with that row (foreign keys) in a child table are also deleted. 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 **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. Of course, this cascading delete works only if the **stock_num** that you are deleting has not been ordered; otherwise, the constraint from the **items** table would disallow the cascading delete. For more information, see "What Happens to Multiple Child Tables?".

You specify cascading deletes with the REFERENCES clause on the ADD CONSTRAINT clause. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to specify cascading deletes in tables; however, you do need the Delete privilege on tables that are referenced in the DELETE statement. After you indicate cascading deletes, when you delete a row from a parent table, Universal Server deletes any associated matching rows from the child table.

Use the ADD CONSTRAINT clause to add a REFERENCES clause with the ON DELETE CASCADE constraint.

What Happens to Multiple Child Tables?

When you have a parent table with two child tables, one with cascading deletes specified and the other without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, the delete statement fails, and no rows are deleted from either the parent or child tables.

In the previous example, the **stock** table is also parent to the **items** table. However, you do not need to add the cascading-delete clause to the **items** table if you are planning to delete only unordered items. The **items** table is used only for ordered items.

Locking and Logging

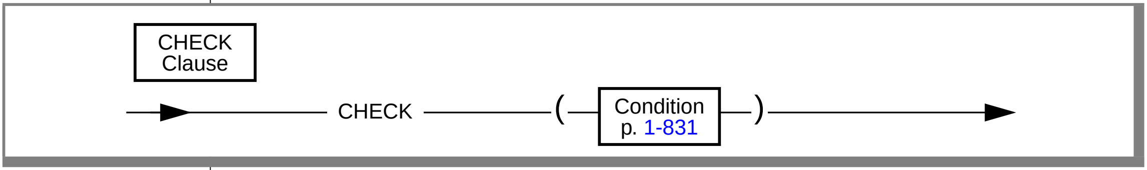
During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables. You must turn logging on when you perform the deletes. When logging is turned off in a database, even temporarily, deletes do not cascade. This restriction applies because you have no way to roll back actions if logging is turned off. For example, if a parent row is deleted, and the system crashes before the child rows are deleted, the database would have dangling child records. Such records would violate referential integrity. However, when logging is turned back on, subsequent deletes cascade.

Restriction on Cascading Deletes

Cascading deletes can be used for most deletes except correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a query that contains such a correlated subquery.

ON DELETE CASCADE appears in the REFERENCES clause on page 1-65.

CHECK Clause



A check constraint designates a condition that must be met *before* data can be inserted into a column. If a row evaluates to false for any check constraint that is defined on a table during an insert or update, the database server returns an error.

Check constraints are defined using *search conditions*. The search condition cannot contain the following items: subqueries, aggregates, host variables, rowids, or stored procedure calls. In addition, the search condition cannot contain the following functions: the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions.

You cannot create check constraints for columns across 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 following example adds a new column, **unit_price**, to the **items** table and includes a check constraint that ensures 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)
```

The CHECK clause appears in the Column-Constraint Definition on page 1-61.

BEFORE Option

Use the BEFORE option of the ADD clause to specify the column before which a new column or list of columns is to be added. The column that you specify in the BEFORE option must be an existing column in the table.

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

In the following example, to add the **item_weight** column before the **total_price** column, include the BEFORE option in the ADD clause:

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

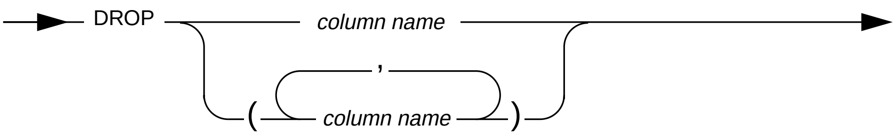
In the following example, to add the **item_weight** column to the end of the table, omit the BEFORE option from the ADD clause:

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

The BEFORE option appears in the ADD clause on page 1-55.

DROP Clause

DROP Clause



Element	Purpose	Restrictions	Syntax
column name	The name of the column that you want to drop	The column must already exist in the table. If the column is referenced in a fragment expression, it cannot be dropped.	Identifier, p. 1-962

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

The DROP clause appears in the Alter Clause for Untyped Tables on page 1-54.

How Dropping a Column Affects Constraints

When you drop a column, all constraints placed on that column are dropped, as the following list describes:

- 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 unique or primary-key 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

When you drop a column that occurs in the triggering column list of an UPDATE trigger, the column is dropped from the triggering column list. If the column is the only member of the triggering column list, the trigger is dropped from the table. See the CREATE TRIGGER statement on page 1-255 for more information on triggering columns in an UPDATE trigger.

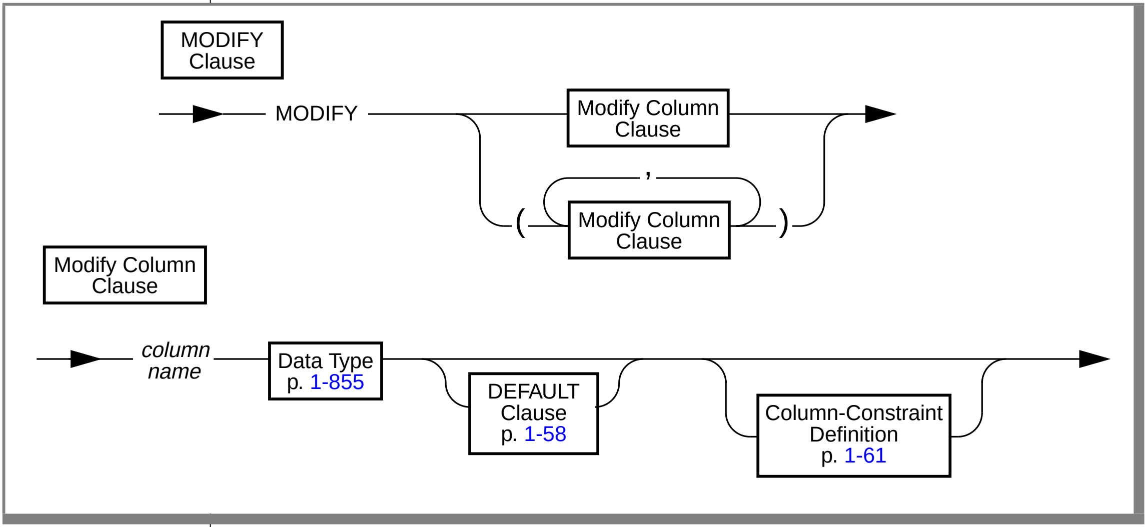
How Dropping a Column Affects Views

When you alter a table by dropping a column, views that depend on the column are not modified. However, if you attempt to use the view, you receive an error message indicating that the column was not found.

Views are not dropped because you can change the order of columns in a table by dropping a column and then adding a new column with the same name. Views based on that table continue to work. They retain their original sequence of columns.

MODIFY Clause

Use the MODIFY clause to change the data type of a column and the length of a character column, to add or change the default value for a column, and to allow or disallow nulls in a column.



Element	Purpose	Restrictions	Syntax
column name	The name of the column that you want to modify	The column must already exist in the table.	Identifier, p. 1-962

You cannot modify a column whose data type is a collection type. You cannot modify a column type to be a collection type 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 respecify those attributes. For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, and you want to keep the default value (in this case, 1) and non-null attributes for that column, you can issue the following ALTER TABLE statement:

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

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

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 unique or primary-key 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 unique or primary-key constraint, the multiple-column constraints are not dropped, but any referential constraints placed on the column by other tables *are* dropped. For example, 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.

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

The MODIFY clause appears in the Alter clause for Untyped Tables on page 1-54.

Altering Large-Object Characteristics

You cannot use the ALTER TABLE statement to modify the characteristics of a smart large object column. To modify a smart-large-object column, you must use one of the following:

- The **ifx_lo_alter()** function in ESQL/C

  For more information, refer to the *INFORMIX-ESQL/C Programmer's Manual*.
- The DataBlade API function **mi_lo_alter()** in external functions

  For more information, refer to the *DataBlade API Programmer's Manual*.

Altering the Next Serial Number

You can use the MODIFY clause to reset the next value of a SERIAL or SERIAL8 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. However, you can set the next value to any value higher than the current maximum, which creates gaps in the sequence.

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 unique or primary-key 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. The NOT NULL 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. However, 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.

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, you can take the following steps to add the constraint successfully:

1. Add the constraint in the disabled mode.

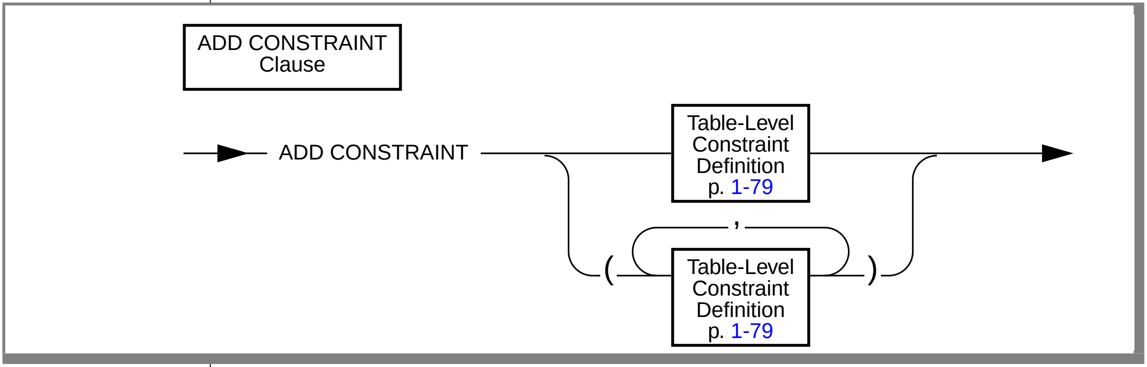
   Issue the ALTER TABLE statement again, but this time specify the DISABLED keyword in the MODIFY clause.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
3. Issue a SET statement to switch the object mode of the constraint to the enabled mode.

   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 switch the disabled constraint to the enabled mode.

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

ADD CONSTRAINT Clause


ADD CONSTRAINT Clause
ADD CONSTRAINT
Table-Level Constraint Definition p. 1-79
(
,
Table-Level Constraint Definition p. 1-79
)

Use the ALTER TABLE statement with the ADD CONSTRAINT keywords to specify a constraint on a new or existing column or on a set of columns. 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)
```

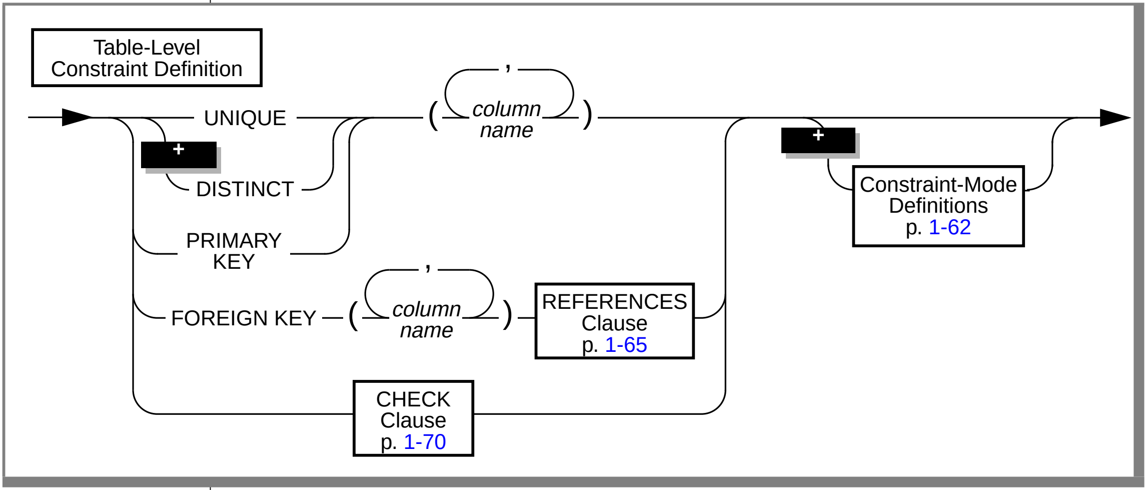
To name the constraint, change the preceding statement, as the following example shows:

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

When you do not provide a constraint name, 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, refer to Chapter 2 of the *Informix Guide to SQL: Reference*.

The ADD CONSTRAINT clause appears in the Alter Clause for Untyped Tables on page 1-54 and the Alter Clause for Typed Tables on page 1-86.

Table-Level Constraint Definition



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns on which the constraint is placed	The maximum number of columns is 16, and the total length of all the columns cannot exceed 390 bytes.	Identifier, p. 1-962

Use the Table-Level Constraint Definition option to add a table-level constraint. You can define a table-level constraint on one column or a set of columns. You can assign a name to the constraint and set its object mode by means of the Constraint Mode Definitions option. See page 1-62 for further information.

The Table-Level Constraint Definition clause appears in the Add Constraints clause on page 1-78.

Adding a Unique Constraint

You must follow certain rules when you add a unique constraint.

The column or columns can contain only unique values.

When you place a 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. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the unique constraint.

When you add a unique constraint, the name of the constraint must be unique within the database.

ANSI

When you add a unique constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

A composite list can include no more than 16 column names. The total length of all the columns cannot exceed 390 bytes.

Adding a Primary-Key or Unique Constraint

You must follow certain rules when you add a unique or primary-key constraint.

When you place a unique or primary-key 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. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the constraint.

When you place a unique or primary-key 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.

When you place a referential 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.

When you add a unique or primary-key constraint, the name of the constraint must be unique within the database.

ANSI

When you add a unique or primary-key constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

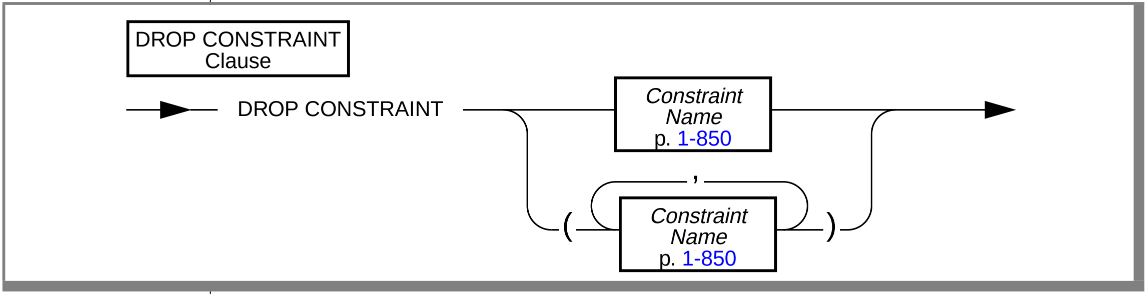
Privileges Required for Adding Constraints

When you own the table or have the Alter privilege on the table, you can create a unique, primary-key, or check 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.

Recovery from Constraint Violations

If you use the ADD CONSTRAINT clause to add a table-level constraint in the enabled mode and receive an error message because existing rows would violate the constraint, you can follow a procedure to add the constraint successfully. See "Adding a Constraint When Existing Rows Violate the Constraint" on page 1-77.

DROP CONSTRAINT Clause



Use the DROP CONSTRAINT clause to drop any type of constraint, including not-null constraints.

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

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

If a *constraint name* is not specified when the constraint is created, the database server generates the name. You can query the **sysconstraints** system catalog table for the names (including the owner) of constraints. For example, to find the name of the constraint placed on the **items** table, you can issue the following statement:

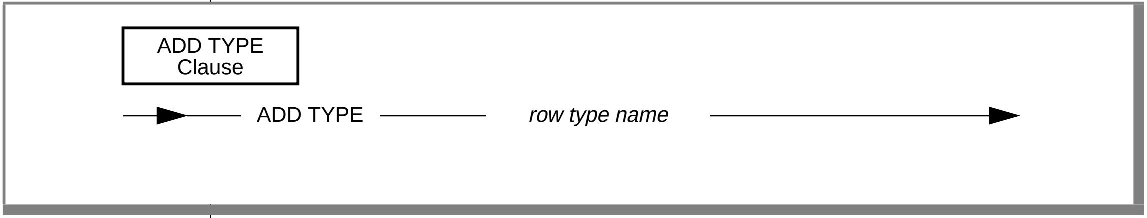
```
SELECT constrname FROM  sysconstraints
    WHERE tabid = (SELECT tabid FROM systables
        WHERE tabname = 'items')
```

When you drop a unique or primary-key constraint that has a corresponding foreign key, the referential constraints is 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.

The DROP CONSTRAINT clause appears in the Alter Clause for Untyped Tables on page 1-54.

ADD TYPE Clause

Use the ALTER TABLE command to change an untyped table into a typed table. When you specify ADD TYPE, you assign a named row type to the table.



Element	Purpose	Restrictions	Syntax
row type name	The 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. You cannot add a type to a fragmented table that has rowids.	Data Type, p. 1-855

Use the ADD TYPE clause to convert an untyped table to a typed table of the named row type.

You cannot combine the ADD TYPE clause with any clause that changes the structure of the table. That is, you cannot use an ADD, DROP, or MODIFY clause in the same statement as the ADD TYPE clause.

Tip: *To change the data type of a column within an untyped table, use the MODIFY clause.*

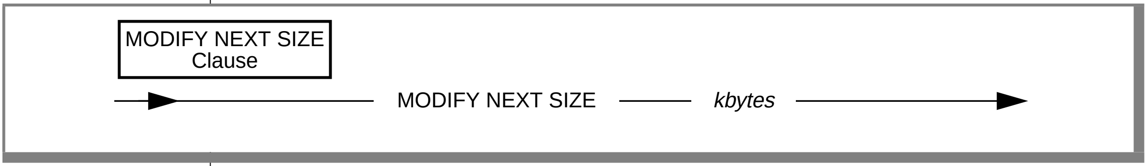
When you add a named row type to a table, be sure that:

- the type already exists.
- the fields in the named row type match the column types in the table.

Important: *You must have the Usage privilege to add a type to a table.*

The ADD TYPE clause appears in the Alter Clause for Untyped Tables on page 1-54.

MODIFY NEXTSIZE Clause



Element	Purpose	Restrictions	Syntax
kbytes	The length in kilobytes that you want to assign for the next extent for this table	The minimum length 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 maximum length is equal to the chunk size.	Expression, p. 1-876

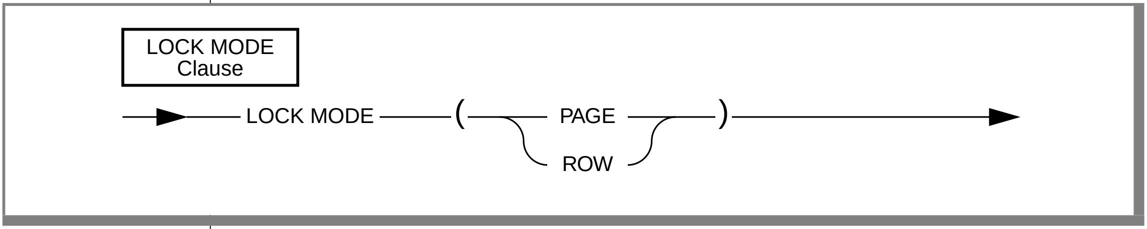
Use the MODIFY NEXT SIZE clause to change the size of new extents. If you want to specify an extent size of 32 kilobytes, use a statement such as the one in the following example:

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

The size of existing extents is not changed.

The MODIFY NEXT SIZE clause appears in the Alter Clause for Untyped Tables on page 1-54.

LOCK MODE Clause



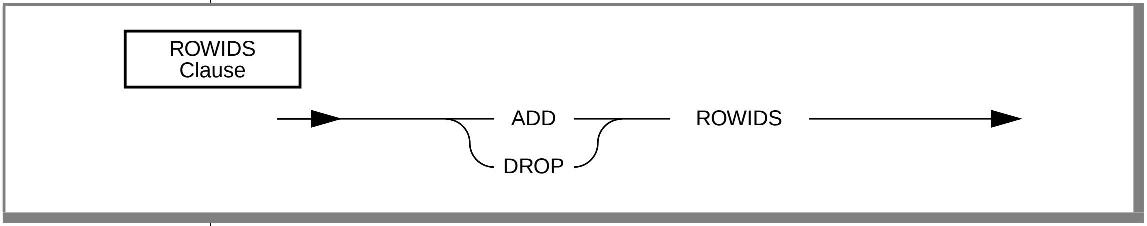
U se the LOCK MODE keywords to change the locking mode of a table. The default lock mode is PAGE; it is set if the table is created without using the LOCK MODE clause. You must use the LOCK MODE clause to change from page to row locking, as the following example shows:

```
ALTER TABLE items LOCK MODE (ROW)
```

The LOCK MODE clause appears in the Alter Clause for Untyped Tables on page 1-54.

ROWIDS Clause

Use the ROWIDS clause to add or remove rowids from a column in a fragmented table. By default, fragmented tables do not contain the *hidden* rowid column.



Use ADD ROWIDS to add a new column called **rowid** for use with fragmented tables. For each row, the database server assigns a unique number that remains stable for the life of the row. The database server creates an index that it uses when search to find the physical location of the row. After you add the rowid column, each row contains an additional 4 bytes to store the rowid value.

You can use DROP ROWIDS to drop a rowid column only if you created the rowid column with the CREATE TABLE or ALTER FRAGMENT statements on fragmented tables. You cannot drop the rowid columns of a nonfragmented table.

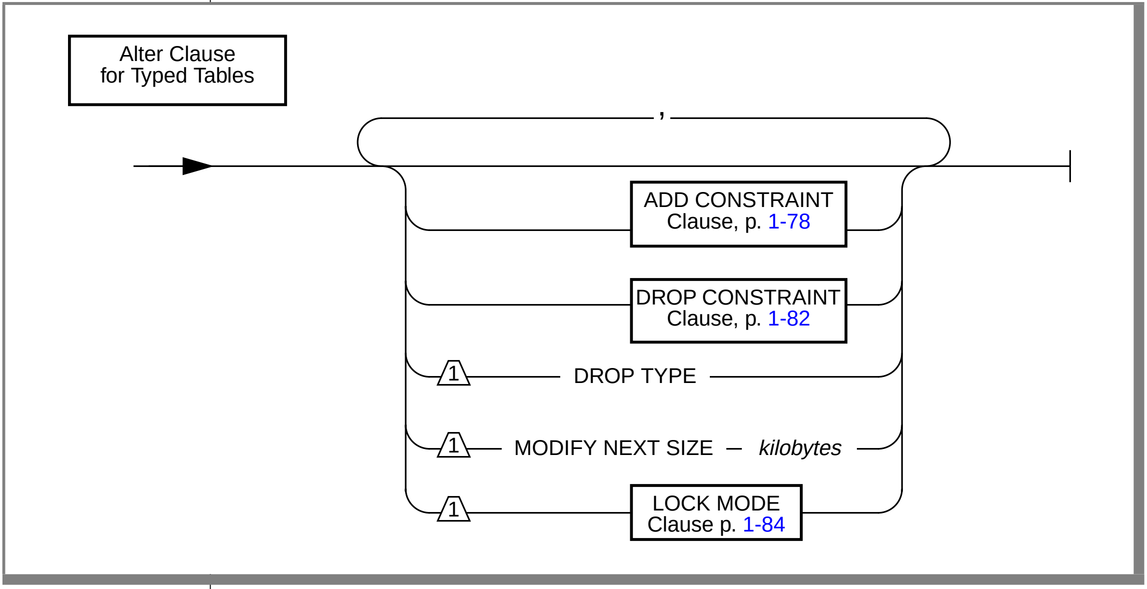
Tip: *Use the ADD ROWIDS clause only on fragmented tables. In nonfragmented tables, the rowid column remains unchanged. Informix recommends that you use primary keys as an access method rather than exploiting the rowid column.*

The ROWIDS clause appears in the Alter Clause for Untyped Tables on page 1-54.

For additional information about the rowid column, refer to the *INFORMIX-Universal Server Administrator's Guide*.

Alter Clause for Typed Tables

The database server performs the actions in the Alter Clause in the order that you specified. If any of the actions fails, the entire operation is cancelled.


Alter Clause for Typed Tables
,
ADD CONSTRAINT Clause, p. 1-78
DROP CONSTRAINT Clause, p. 1-82
1
DROP TYPE
1
MODIFY NEXT SIZE
kilobytes
1
LOCK MODE Clause p. 1-84

The Alter Clause for Typed Tables appears in the ALTER TABLE syntax on page 1-52.

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 Clause

Use DROP TYPE to drop the type from a table. DROP TYPE changes a typed table to an untyped table. 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.

MODIFY NEXT SIZE Clause

Use the MODIFY NEXT SIZE clause to change the size of new extents. If you want to specify an extent size of 32 kilobytes, use a statement such as the one in the following example:

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

The size of existing extents is not changed.

References

See the CREATE TABLE, DROP TABLE, and LOCK TABLE statements in this manual.

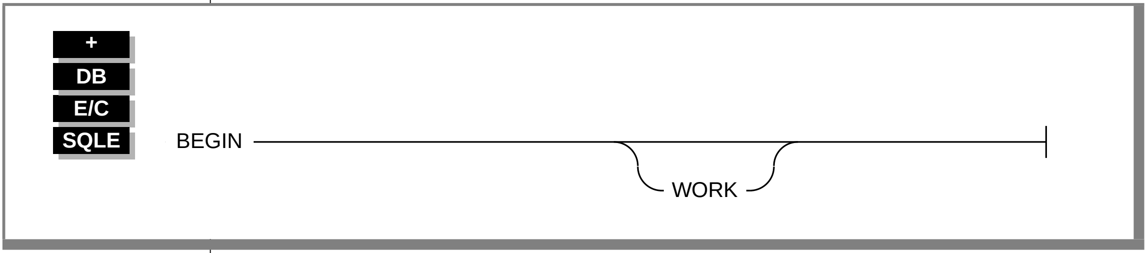
In the *Informix Guide to SQL: Tutorial*, see the discussion of data-integrity constraints and the discussion of the ON DELETE CASCADE clause in Chapter 4. Also see the discussion of creating a database and tables in Chapter 9.

See the SET statement in this manual for information on object modes.

BEGIN WORK

Use the BEGIN WORK statement to start a transaction (a sequence of database operations that the COMMIT WORK or ROLLBACK WORK statement terminates).

Syntax



Usage

The following code fragment shows how you might place statements within a transaction:

```
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
```

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 affecting many rows can exceed the limits that your operating system or the INFORMIX-Universal 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.

You can 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.

ESQL

If you use the BEGIN WORK statement within a routine 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. ♦

With ANSI-Compliant Databases

ANSI

The BEGIN WORK statement is not needed because transactions are implicit. A warning is generated if you use a BEGIN WORK statement immediately after one of the following statements:

- DATABASE
- COMMIT WORK
- CREATE DATABASE
- ROLLBACK WORK

An error is generated if you use a BEGIN WORK statement after any other statement. ♦

References

See the COMMIT WORK and ROLLBACK WORK statements in this manual.

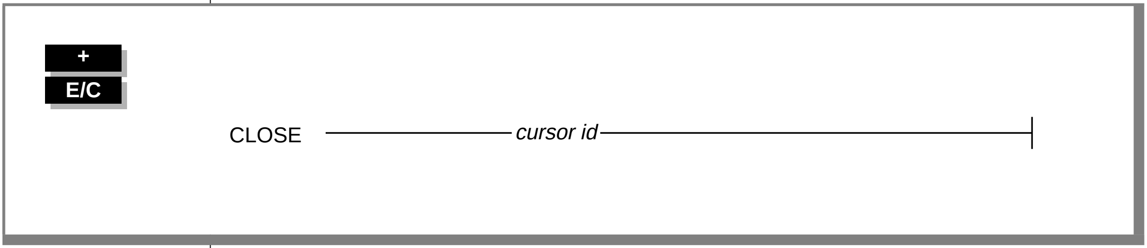
In the *Informix Guide to SQL: Tutorial*, see the discussion of transactions and locking in Chapter 4 and Chapter 7, respectively.

CLOSE

Use the CLOSE statement to close a cursor in the following situations:

- You no longer need to refer to the rows that a select or function cursor produced.
- You want to flush and close an insert cursor.
- You no longer need to access a collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	The name of the cursor to be closed	The DECLARE statement must have previously declared the cursor.	Identifier, p. 1-962

Usage

The CLOSE statement deallocates resources that have been allocated to a cursor when it was opened with the OPEN statement. 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 (an insert cursor) differently than one that is associated with a SELECT statement (a select cursor) or an EXECUTE FUNCTION statement (a function cursor).

You can close a cursor that was never opened or that has already been closed. No action is taken in these cases.

ANSI

You get an error if you close a cursor that was not open. No other action occurs. ♦

Closing a Select or Function Cursor

When *cursor id* is associated with a SELECT statement (select cursor) or an EXECUTE FUNCTION statement (function cursor), the CLOSE statement terminates the SELECT 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 *cursor id* is associated with an INSERT statement (insert cursor), 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 in the **sqlca** structure (**sqlca.sqlerrd[2]**).For information on using SQLERRD to count the total number of rows that were inserted, see the PUT statement on page 1-552.

he **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, the database server sets **SQLCODE** to zero. If an error is encountered, the database server sets SQLCODE 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 might exist. See the GET DIAGNOSTICS statement for information about the **SQLSTATE** status variable.*

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 CLOSE statement failed in this case, the cursor is not closed. A second CLOSE statement can be successful because no buffered rows exist. A subsequent OPEN statement should also be successful because the OPEN statement performs a successful implicit close. For example, a CLOSE statement can fail if insufficient disk space prevents some of the rows from being inserted.

Closing a Collection Cursor

You can declare both select and insert cursors on **collection** variables. Such cursors are called collection cursors. (For more information, see the DECLARE statement on page 1-300.) To close a collection cursor, use the CLOSE statement. The CLOSE statement deallocates resources that have been allocated for the collection cursor.

For more information on the use of OPEN with a collection cursor, see the following sections in the FETCH statement: "Fetching From a Collection Cursor" on page 1-419 and "Inserting into a Collection Cursor" on page 1-560.

Using End of Transaction to Close a Cursor

The COMMIT WORK and ROLLBACK WORK statements close all cursors except hold cursors (those that are declared with the WITH HOLD option of DECLARE). 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 the use of insert cursors and the WITH HOLD clause, see the DECLARE statement on page 1-300.

References

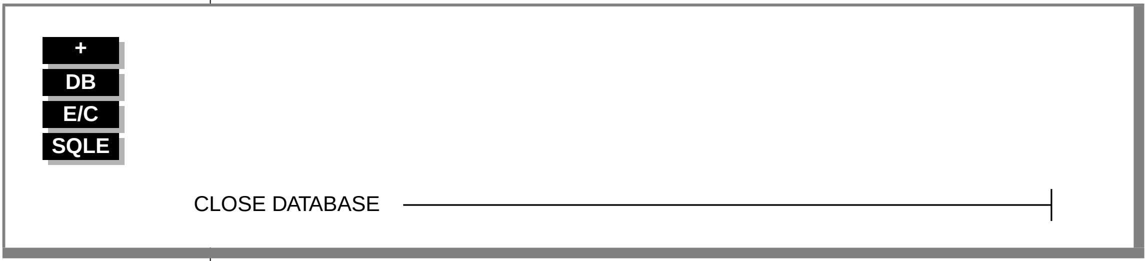
See the CLOSE, DECLARE and FREE statements in this manual for general information about cursors. See the PUT and FLUSH statements in this manual for information about insert cursors. See the FETCH statement in this manual for information about select and function cursors.

In the *Informix Guide to SQL: Tutorial*, see the discussion of cursors in Chapter 5.

CLOSE DATABASE

Use the CLOSE DATABASE statement to close the current database.

Syntax



Usage

Following the CLOSE DATABASE statement, you can use only the DATABASE, CREATE DATABASE, and DROP DATABASE statements. A DISCONNECT statement can also follow a CLOSE DATABASE statement, but only if an explicit connection existed before you issue the CLOSE DATABASE statement. A CONNECT statement can follow a CLOSE DATABASE statement without any restrictions.

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

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

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

```
DATABASE stores7
.
.
.
CLOSE DATABASE
DROP DATABASE stores7
```

ESQL

The CLOSE DATABASE statement cannot appear in a multistatement PREPARE operation.

If you use the CLOSE DATABASE statement within a routine 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, declared cursors are no longer valid. You must redeclare any cursors that you want to use. ♦

References

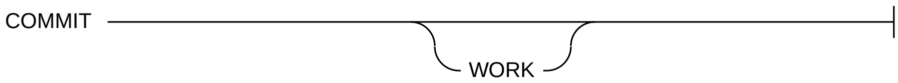
See the CONNECT, DATABASE, CREATE DATABASE, DISCONNECT, and DROP DATABASE statements in this manual.

COMMIT WORK

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

Syntax

DB
E/C
SQLE



Usage

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

The COMMIT WORK statement releases all row and table locks.

ESQL

The COMMIT WORK statement closes all open cursors except those declared with hold. ♦

Issuing COMMIT WORK in a Database That Is Not ANSI Compliant

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

If you are using a database that is not ANSI compliant, and you do not issue a BEGIN WORK statement, the database server executes each statement within its own transaction. These single-statement transactions do not require either a BEGIN WORK statement or a COMMIT WORK statement.

Issuing COMMIT WORK in an ANSI-Compliant Database

ANSI

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

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

References

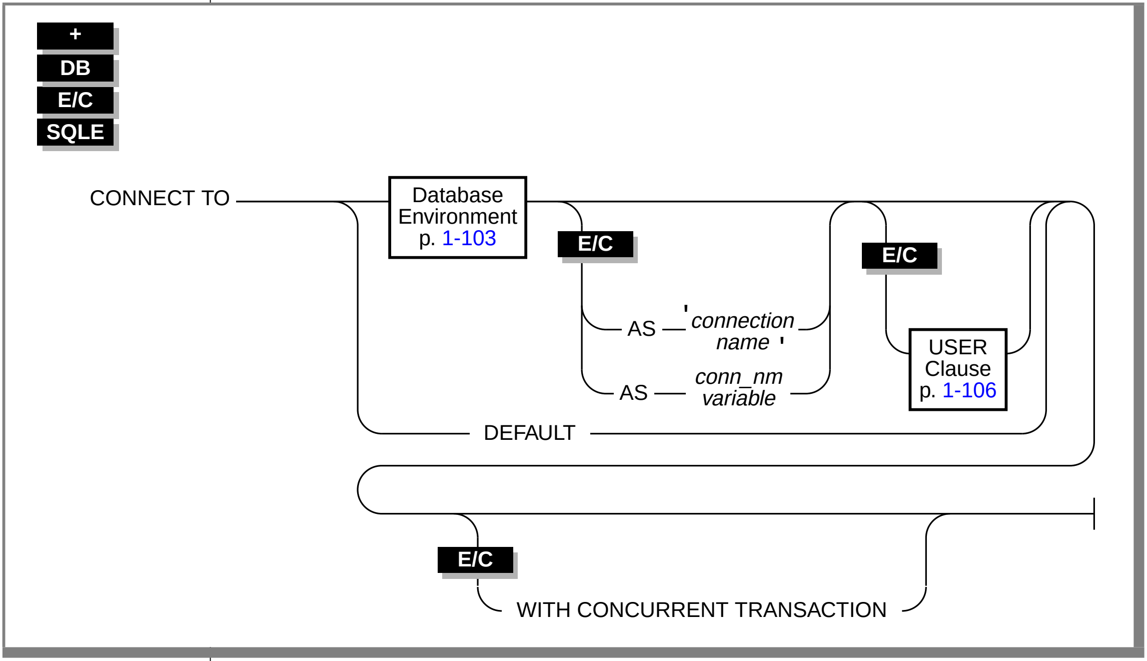
See the BEGIN WORK, ROLLBACK WORK, and DECLARE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of transactions in Chapter 4.

CONNECT

Use the CONNECT statement to connect to a database environment.

Syntax



Element	Purpose	Restrictions	Syntax
connection name	Quoted string that assigns a name to the connection	If your application makes multiple connections to the same database environment, you must specify a unique connection name for each connection.	Quoted String, p. 1-1010
conn_nm variable	Host variable that holds the value of *connection name*	Variable must be a fixed-length character data type.	Variable name must conform to language-specific rules for variable names.

Usage

The CONNECT statement connects an application to a *database environment*. The database environment 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 no current connection exists between an application and a database server. If you specify a database name, the database server opens the database.You cannot use the CONNECT statement in 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. The only restriction on this 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. (See the *INFORMIX-Universal Server Administrator's Guide* for more information.)

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 transaction becomes dormant. You can make a dormant connection current with the SET CONNECTION statement. See "SET CONNECTION" on page 1-682.

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 using 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 1-106.

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 *connection name* (or a *conn_nm* 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.

The value of *connection name* is case sensitive.

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 on dormant connections, see "Making a Dormant Connection the Current Connection" on page 1-683.)

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 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, the CREATE DATABASE statement, or the START DATABASE statement to open or create a database in the default database environment.

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 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 "Locating the Database" on page 1-105.

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 server is the default specified by the **INFORMIXSERVER** environment variable. This default allows the application to refer to the implicit connection if additional explicit connections are made, because the implicit connection does not have an 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 command because the implicit connection is considered to be 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.

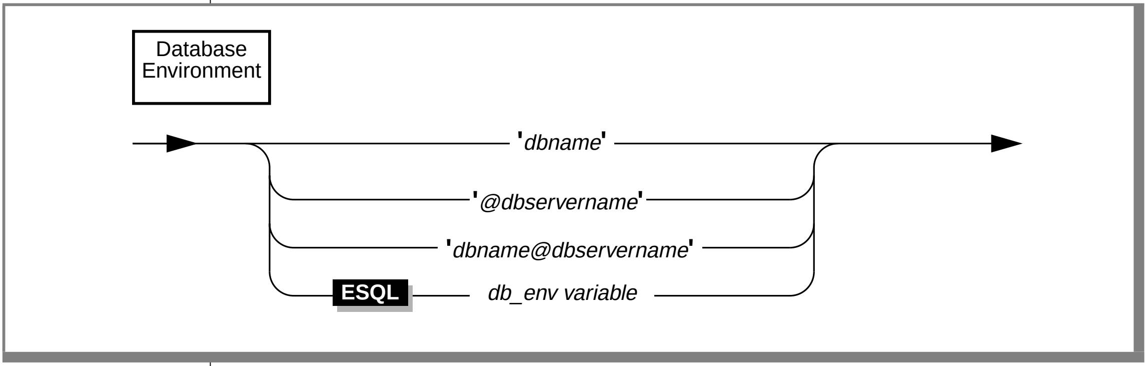
The WITH CONCURRENT TRANSACTION clause supports the concept of multiple concurrent transactions, where each connection can have its own transaction and the COMMIT WORK and ROLLBACK WORK statements affect only the current connection.The WITH CONCURRENT TRANSACTION clause does not support global transactions 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. A deadlock condition occurs when one transaction obtains a lock on a table, and a concurrent transaction tries to obtain a lock on the same table, resulting in the application waiting for itself to release the lock.*

Database Environment



Element	Purpose	Restrictions	Syntax
db_env variable	Host variable that contains a value representing a database environment	Variable must be a fixed-length character data type. The value stored in this host variable must have one of the database-environment formats listed in the syntax diagram.	Variable name must conform to language-specific rules for variable names.
dbname	Quoted string that identifies the name of the database to which a connection is made	Specified database must already exist. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-1010
dbservername	Quoted string that identifies the name of the database server to which a connection is made	Specified database server must match the name of a server in the **sqlhosts** file. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-1010
dbname@ dbservername	Quoted string that identifies the name of the database and database server to which a connection is made	Specified database must already exist. Specified database server must match the name of a server in the **sqlhosts** file. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-1010

Specifying the Database Environment

Using the options shown in the syntax diagram, you can specify either a server and a database, a database server only, or a database only.

Specifying a Database Server Only

The *@dbservername* option establishes a connection to the named 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.

Specifying a Database Only

The *dbname* option establishes connections to the default 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_env variable* option if it specifies only a database name. See “Locating the Database” for the order in which an application connects to different database servers to locate a database.

Locating the Database

How a database is located and opened depends on whether you specify a database server name in the database environment expression

Database Server and Database Specified

If you specify both a database server and a database in the CONNECT statement, your application connects to the database server, which locates and opens the database. For the Universal Server database server, it uses parameters that are specified in the ONCONFIG configuration file to locate the database.

If the database server that you specify is not on-line, you get an error.

Only Database Specified

If you specify only a database in your CONNECT statement, not a database server, the application obtains the name of a database server from the **DBPATH** environment variable. The database server in the **INFORMIXSERVER** environment variable is always added in front of the **DBPATH** value specified by the user. Set environment variables as the following example shows:

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

The resulting **DBPATH** used by your application is shown in the following example:

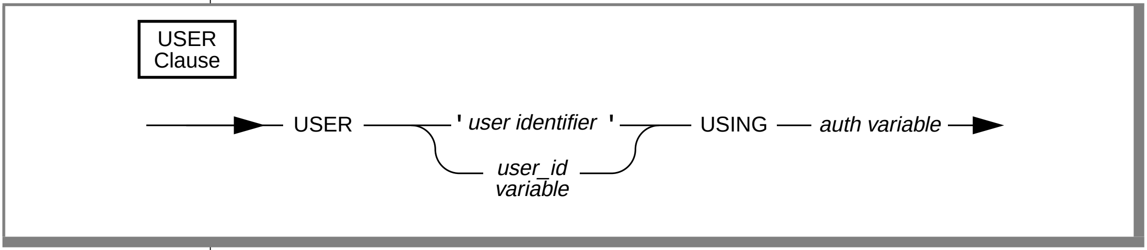
```
//srvA://srvB://srvC
```

The application first establishes a connection to the database server specified by **INFORMIXSERVER**. For the Universal Server database server, it uses parameters that are specified 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 on-line, the application connects to the next database server in **DBPATH**. In the previous example, this server would be **srvB**.

If a directory in **DBPATH** is an NFS-mounted directory, it is expanded to contain the host name of the NFS computer and the complete pathname of the directory on the NFS host. In this case, the host name must be listed in your **sqlhosts** file as a dbservername, and an **sqlexecd** daemon must be running on the NFS host.

USER Clause



Element	**Purpose**	**Restrictions**	**Syntax**
auth variable	Host variable that holds the valid password for the login name specified in *user identifier* or *user_id variable*	Variable must be a fixed-length character data type. The password stored in this variable 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.	Variable name must conform to language-specific rules for variable names.
user_id variable	The name of an ESQL/C host variable that holds the value of *user identifier*	Variable must be a fixed-length character data type. The login name stored in this variable is subject to the same restrictions as the *user identifier* variable.	Variable name must conform to language-specific rules for variable names.
user identifier	Quoted string that is a valid login name for the application	Specified login name 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.	Quoted String, p. 1-1010

The User clause specifies information that is used to determine whether the application can access the target computer 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.

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 *auth variable* phrase.

ESQL

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 *auth variable* phrase. The connection is rejected at runtime by Informix database servers, however, if the *auth variable* is not present. ♦

If you do not supply the USER clause, the connection is attempted using the default user ID. The default Informix user ID is the login name of the user running the application. In this case, network permissions are obtained using the standard UNIX authorization procedures (for example, checking the **/etc/hosts.equiv** file).

Connecting to INFORMIX-OnLine Dynamic Server Before Version 6.0

The CONNECT statement syntax described in this chapter is valid for a Version 6.0 or later application connecting to database servers earlier than Version 6.0. As with Version 6.0 or later database servers, an implicit connection can be made to a database server earlier than Version 6.0, provided that no existing implicit connections exist and no implicit connections have been previously terminated.

Connections to pre-Version 6.0 OnLine database servers differ from connections to Version 6.0 or later OnLine and Universal Server in the following respects:

- The CLOSE DATABASE statement causes a connection to a pre-Version 6.0 database server to be dropped. The same statement, applied to a connection to a Version 6.0 or later database server, causes the database to close, but the connection remains.
- If an application makes a connection to a pre-Version 6.0 database server without using the WITH CONCURRENT TRANSACTION clause, you must close the database (effectively dropping the connection) before you switch to a different connection. Otherwise, Version 6.0 and later OnLine and Universal Server return error message -1800.

References

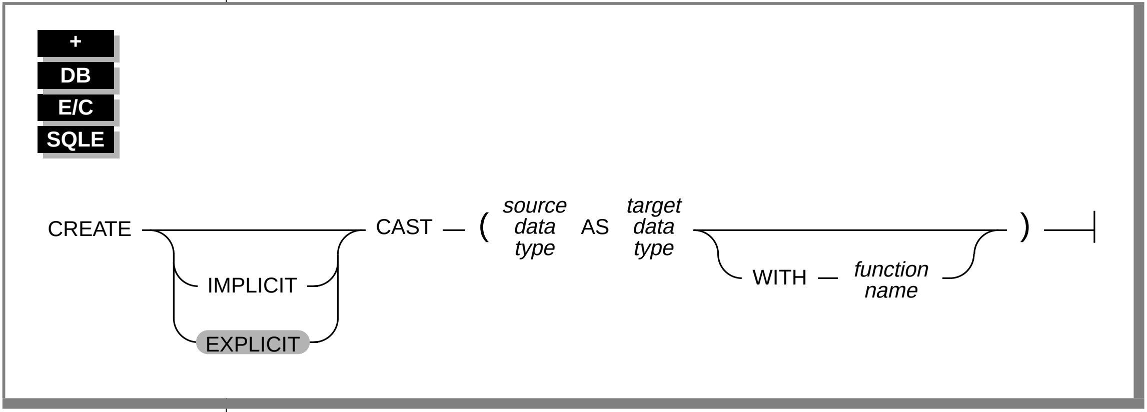
See the DISCONNECT, SET CONNECTION, DATABASE, START DATABASE, and CREATE DATABASE statements in this manual.

For information on the contents of the **sqlhosts** file, refer to the *INFORMIX-Universal Server Administrator’s Guide*.

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
source data type	The data type to be converted	The type must exist in the database at the time the cast is registered. Either the *source data type* or the *target data type*, but not both, can be a built-in type. Neither type can be a distinct type of the other. The type cannot be a collection data type.	Data Type, p. 1-855

(1 of 2)

Element	Purpose	Restrictions	Syntax
target data type	The data type that results from the conversion	The type must exist in the database at the time the cast is registered. Either the *source data type* or the *target data type*, but not both, can be a built-in type. Neither type can be a distinct type of the other. The type cannot be a collection data type.	Data Type, p. 1-855
function name	The name of the function that you register to implement the cast	See "WITH Clause" on page 1-113.	Function Name, p. 1-959

(2 of 2)

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 a 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. However, to create a cast to or from an opaque type, distinct type, or named row type requires the Usage privilege on that 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 *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 yyy column (of type **percent**):

```
SELECT bond_rate FROM bond
WHERE bond_rate::percent > 15
```

Implicit Casts

The database server invokes system-defined 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 system-defined cast.

An implicit cast is a cast that the database server can invoke automatically when it encounters data types that cannot be compared with system-defined casts. This type of cast enables the database server to handle automatically 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 when it needs to convert from the CHAR data type to a distinct data type, **percent**:

```
CREATE IMPLICIT CAST (CHAR AS percent) WITH prcnt_to_char
```

This cast only provides the database server with the ability to automatically convert *from* the CHAR data type *to* **percent**. For the database server to convert *from* **percent** *to* CHAR, you need to define another implicit cast, as follows:

```
CREATE IMPLICIT CAST (percent AS CHAR) WITH char_to_prcnt
```

The database server would automatically invoke 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 casting function, see "Explicit Casts" on page 1-111.

When a system-defined cast does not exist for conversion between data types, you can create user-defined casts to make the necessary conversion. Universal Server supports the following types of casts:

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 casting 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 casting 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 casting function exists. Each time a user invokes the cast explicitly or implicitly, the database server verifies that the user has Execute privilege on the casting function.

References

See the CREATE FUNCTION statement in this manual for information about registering the functions that are used to implement casts. See the CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, and CREATE ROW TYPE statements in this manual for information about creating new data types. See the DROP CAST statement in this manual for information about removing a cast from a database.

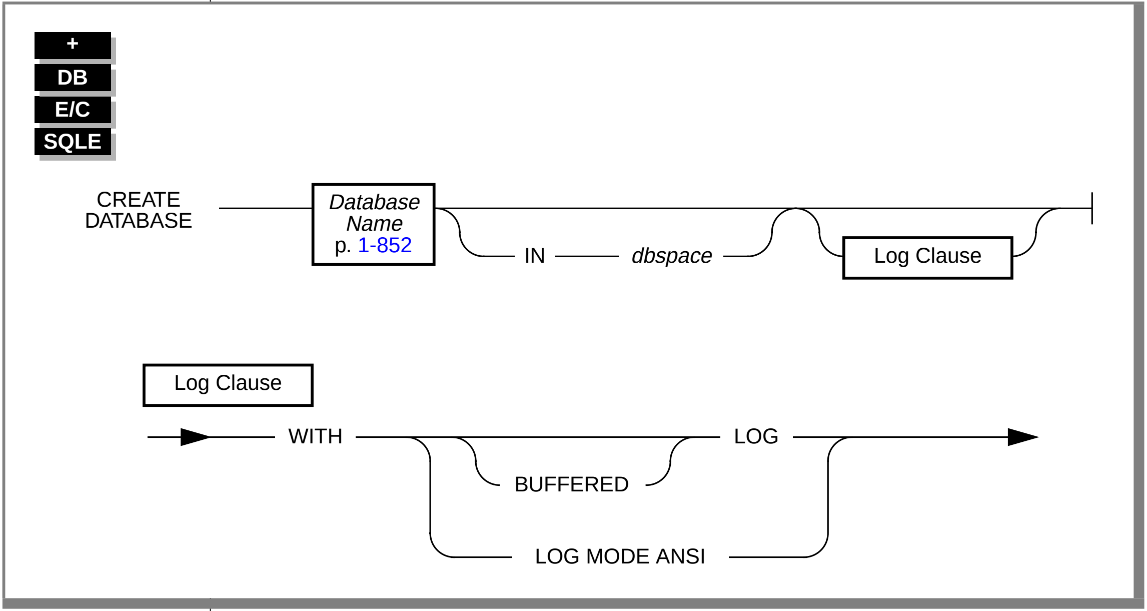
See the Data Types segment in this manual and Chapter 2 in the *Informix Guide to SQL: Reference* for more information about data types, casting, and conversion.

In the *Informix Guide to SQL: Tutorial*, see Chapter 13 for examples that show how to create and use casts.

CREATE DATABASE

Use the CREATE DATABASE statement to create a new database.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where you want to store the data for this database; default is the root dbspace	The dbspace must already exist.	Identifier, p.1-962
pathname	The full pathname, including the file name, for the log file	You cannot specify an existing file.	The pathname and filename must conform to the conventions of your operating system.

Usage

The database that you create 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 contain the data dictionary, which describes the structure of the database in the dbspace. If you do not specify the dbspace, The database server creates the system catalog tables in the **root** dbspace.

When you create a database, you alone have access to it. The database remains inaccessible to other users until you, as DBA, grant database privileges. For information on granting database privileges, see the GRANT statement on page 1-458.

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
```

ESQL

In SQL APIs, the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation. ♦

ANSI-Compliant Databases

ANSI

You have the option of creating an ANSI-compliant database. ANSI-compliant databases differ from databases that are not ANSI compliant in the following ways:

- All statements are automatically contained in transactions. All databases on the database server 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 on the database server, the default isolation level available is Repeatable Read.
- Default privileges on objects differ from those in databases that are not ANSI compliant. Users do not receive the 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 as appropriate with the related SQL statement. ♦

Logging on INFORMIX-Universal Server

In the event of a failure, INFORMIX-Universal Server uses the log to re-create all committed transactions in your database.

If you do not specify the WITH LOG clause, you cannot use transactions or the statements that are associated with databases that have logging (BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET LOG, and SET ISOLATION).

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 Chapter 9 of the *Informix Guide to SQL: Tutorial*.)

ANSI

An ANSI-compliant database does not use buffered logging. ♦

Designating an ANSI-Compliant INFORMIX-Universal Server Database

The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG MODE ANSI
```

Creating an ANSI-compliant database does not mean that you receive ANSI warnings when you run the database. You must use the **-ansi** flag or the **DBANSIWARN** environment variable to receive warnings.

For additional information about **-ansi** and **DBANSIWARN**, see Chapter 3 in the *Informix Guide to SQL: Reference*.

References

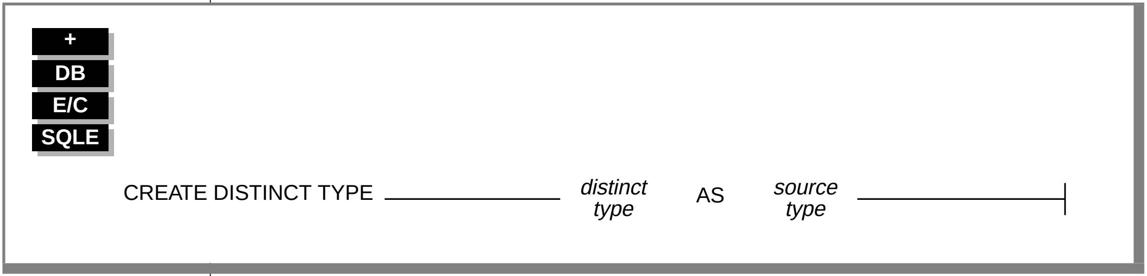
See the CLOSE DATABASE, CONNECT TO, DATABASE, DROP DATABASE, and START DATABASE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating a database in Chapter 9.

CREATE DISTINCT TYPE

Use the CREATE DISTINCT TYPE statement to create a new distinct type. A *distinct type* is a data type based on a built-in type or an existing opaque type, a named row type, or another distinct type. Distinct types are strongly typed. Although the distinct type has the same physical representation as data of its source type, the two types cannot be compared without an explicit cast from one type to the other.

Syntax



Element	Purpose	Restrictions	Syntax
distinct type	The name of the new 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 of the data type must be unique within the database.	Data Type, p. 1-855
source type	The name of an existing data type on which the new type is based	The type must be either a built-in type or a type created with the CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, or CREATE ROW TYPE statement.	Data Type, p. 1-855

Usage

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 are owned by user **informix**.

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 type or have the Usage privilege on the 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 data type, DATE:

```
CREATE DISTINCT TYPE birthday AS DATE
```

INFORMIX-Universal Server uses the same storage method for the distinct type as it does for the source type of the distinct type. However, a distinct type and its source type cannot be compared in an operation unless one type is explicitly cast to the other type.

Support Functions and Casts

When you create a distinct type, Universal 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 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. However, to create an implicit cast, you must first drop the default explicit cast 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. However, casts and functions that are defined on the distinct type are not available to the source type.

Manipulating Distinct Types

When you manipulate data of the distinct type and its source type, you must explicitly cast one type to the other. This means that to insert or update a column of one type with values of the other type, you must explicitly cast the data to be inserted or updated. In addition, you cannot 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;
```

References

For information and examples that show how to use and cast distinct types, see Chapter 13 of the *Informix Guide to SQL: Tutorial*.

See the CREATE OPAQUE TYPE and CREATE ROW TYPE statements in this manual for information about how to create opaque types and row types.

See the CREATE FUNCTION statement in this manual for information about registering support functions for a type. See the CREATE CAST statement in this manual for information about registering these functions as casts.

For information about how to remove opaque types and row types from a database, see the DROP TYPE and DROP ROW TYPE statements in this manual.

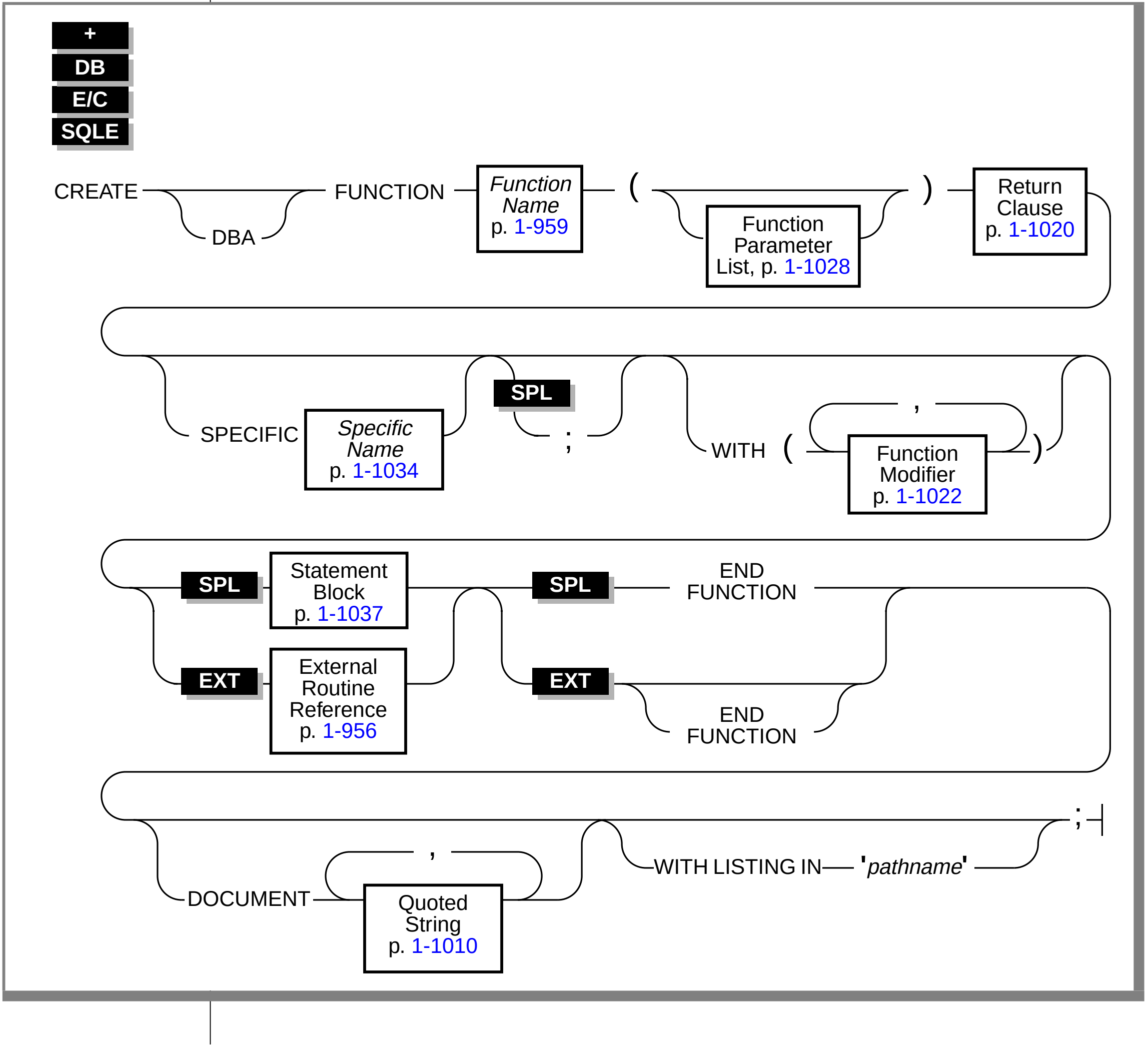
For information about how to create a table that references a data type, see the CREATE TABLE statement in this manual.

For information about built-in data types, user-defined types, and named row types, see the Data Types segment in this manual.

CREATE FUNCTION

Use the CREATE FUNCTION statement to register an external function or to write and register an SPL function.

Syntax



Element	Purpose	Restrictions	Syntax
pathname	The pathname to a file in which compile-time warnings are stored	The specified pathname must exist on the computer where the database resides.	The pathname and filename must conform to the conventions of your operating system.

Usage

A *function* is a user-defined routine that can accept arguments and returns one or more values. INFORMIX-Universal Server supports functions written in the following languages:

- Stored Procedure Language (*SPL functions*)

  An SPL function can return one or more values.
- One of the external languages (such as C) that INFORMIX-Universal Server supports (*External functions*)

  An external function can return only one value.

The entire length of a CREATE FUNCTION statement must be less than 64 kilobytes. This length is the literal length of the statement, including blank space and tabs.

Routines, Functions, and Procedures

The generic term *routine* includes both procedures and functions. A *procedure* is a routine that can accept arguments but does not return any values. A *function* is routine that can accept arguments and returns one or more values. INFORMIX-Universal Server treats any routine that includes a Return clause as a function.

Legacy Procedures

In earlier Informix products, the term *stored procedure* was used for both SPL procedures and SPL functions. However, the database server distinguishes between procedures and functions, even when they are written in SPL. When you use CREATE FUNCTION to write an SPL routine, you create an SPL function.

SPL

SPL Functions

SPL functions are routines written in Stored Procedure Language (SPL) that return one or more values.

Use one CREATE FUNCTION statement, with SQL and SPL statements embedded between CREATE FUNCTION and END FUNCTION, to write and register an SPL function. Unlike external functions, you do not need to write the function and register it in separate steps.

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 Chapter 1, "System Catalog," in the *Informix Guide to SQL: Reference*.

You must use the END FUNCTION keywords with an SPL function.

Place a semicolon after the Return clause or the Modifier clause, whichever comes last. Place another semicolon at the end of the entire statement, after the END FUNCTION, DOCUMENT, or WITH LISTING IN clause.

Examples

The following example creates a 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 writing SPL functions, see Chapter 14, "Creating and Using SPL Routines," in the *Informix Guide to SQL: Tutorial*. ♦

EXT

External Functions

External functions are functions you write in an external language that INFORMIX-Universal Server supports. For this release, INFORMIX-Universal Server supports external functions written in C. To create external functions, follow these steps:

1. Write the function in an external language, such as C, that INFORMIX-Universal Server supports.
2. Compile the function and store the compiled code in a shared library.
3. Register the function in the database server with the CREATE FUNCTION statement.

When INFORMIX-Universal Server executes an external function, the database server invokes the external object code.

The database server does *not* store the body of an external function directly in the database, as it does for SPL functions. Instead, the database server stores only a pathname to the compiled version of the function. You specify this pathname in the External Routine Reference clause.

The database server does store information about an external function in several system catalog tables, including **sysprocbody** and **sysprocauth**. For more information on these system catalog tables, see Chapter 1, "System Catalog," in the *Informix Guide to SQL: Reference*.

With external functions, the END FUNCTION keywords are optional.

Example

The following example registers an external C function named **equal()** in the database. This function takes two arguments of the type **basetype1** and returns a single value of type BOOLEAN. 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 function **basetype1_equal()**, which is invoked during execution of the **equal()** function.

```
CREATE FUNCTION equal ( arg1 opaquetype1, arg2 opaquetype1)
RETURNING BOOLEAN;
EXTERNAL NAME
"/usr/lib/opaquetype1/lib/libbtype1.so(opaquetype1_equal)"
LANGUAGE C
END FUNCTION;
```

♦

DBA Option

The level of privilege necessary to execute a routine depends on whether the routine is created with the DBA keyword.

If you create a function with the DBA option, it is known as a DBA-privileged function. You need the DBA privilege to create or execute a DBA-privileged function.

If you do not use the DBA option, the function is known as an owner-privileged function. If the function is owner privileged, and if the database is ANSI compliant, anyone can execute the function.

If you create an owner-privileged routine in a database that is not ANSI compliant, the **NODEFDAC** environment variable prevents privileges on that routine from being granted to PUBLIC. See the *Informix Guide to SQL: Reference* for further information on the **NODEFDAC** environment variable.

Function Name

Because INFORMIX-Universal 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 if you are defining a type hierarchy or a system of distinct types or casts. When you overload functions, you can create a function for each new data type that you define.

The process of overloading routines and the routine resolution rules are described briefly in "Routine Resolution" on page 1-130.

The syntax of the Function Name segment is described in "Function Name" on page 1-959.

Parameter List

SPL

To define the parameters for an SPL function, specify a parameter name and a data type for each parameter. For more information about defining parameters, see "Routine Parameter List" on page 1-1028. ♦

EXT

To define the parameters for an external routine, you can specify a name and you must specify a data type for each parameter. For more information on the syntax of the parameter list, see "Routine Parameter List" on page 1-1028. ♦

With both SPL functions and external functions, you can specify an OUT parameter, so that the function can be used with a Statement Local Variable in SQL statements. The OUT parameter is described in more detail in "Routine Parameter List" on page 1-1028.

Return Clause

INFORMIX-Universal Server considers any routine that is created with a Return clause to be a function. Informix recommends that you use the CREATE FUNCTION statement, not CREATE PROCEDURE, to create functions. For external routines, this rule is strictly enforced.

The syntax of the Return clause is described in "Return Clause" on page 1-1020.

Specific Name

You can specify a specific name for an SPL procedure or an external procedure. A specific name is a name that is unique in the database. A specific name is useful because more than one procedure can have the same name due to routine overloading.

The syntax of the specific name is described in "Specific Name" on page 1-1034.

Function Modifier

SPL

When you write an SPL function, you can specify the modifier NOT VARIANT with a WITH clause. Both modifiers apply to Boolean functions. The function modifiers are described in “Routine Modifier” on page 1-1022.♦

EXT

In the CREATE FUNCTION statement, you can specify any of a list of function modifiers with a WITH clause. For more information on the function modifiers, see “Routine Modifier” on page 1-1022. ♦

Statement Block

SPL

In an SPL function, you must specify an SPL statement block instead of an External Routine Reference clause. The syntax of the statement block is described in “Statement Block” on page 1-1037. ♦

External Routine Reference

EXT

When you register an external function, you must specify an External Routine Reference clause. The External Routine Reference clause specifies the pathname to the procedure object code, which is stored in a shared library. The External Routine Reference clause also specifies the name of the language in which the procedure is written. For more information on the External Routine Reference clause, see “External Routine Reference” on page 1-956. ♦

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of the routine. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the routine.

To find the description of the SPL procedure **update_by_pct**, shown in “SPL Functions” on page 1-124, enter a query such as the following:

```
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
ORDER BY b.seqno;
```

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
```

An SPL routine, external routine, 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 END FUNCTION. ♦

WITH LISTING IN Clause

The WITH LISTING IN option specifies a filename where compile-time warnings are sent. This listing file is created on the database server when you compile an SPL or external routine.

If you specify a filename but not a directory in the WITH LISTING IN clause, INFORMIX-Universal Server uses the home directory on the database server as the default directory. If you do not have a home directory on the server, the file is created in the root directory.

If you do not use the WITH LISTING IN option, the compiler does not generate a list of warnings.

Privileges Necessary for Using CREATE FUNCTION

You must have the Resource privilege on a database to create a function within that database.

The owner of a privilege grants the Execution privilege for that function to other users. If a function has a commutator function, any user who executes the function must have Execute privilege on both the function and its commutator. If a function has a negator function, any user who executes the function must have Execute privilege on both the function and its negator.

Routine Resolution

In Universal Server, you can have more than one instance of a routine with the same name but different parameter lists, as in the following situations:

- You create a routine 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 functions from their source types.

Routine resolution is the process of determining which instance of a function to execute, given the name of a routine and a list of arguments. For more information on routine resolution, refer to the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

PREPARE Statement

E/C

You can use a CREATE FUNCTION statement only within a PREPARE statement. If you want to create a 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. For more information, see the CREATE FUNCTION FROM statement on page 1-131. ♦

References

See the CREATE PROCEDURE, CREATE FUNCTION FROM, DROP FUNCTION, DROP ROUTINE, GRANT, EXECUTE FUNCTION, PREPARE, UPDATE STATISTICS, and REVOKE statements in this manual.

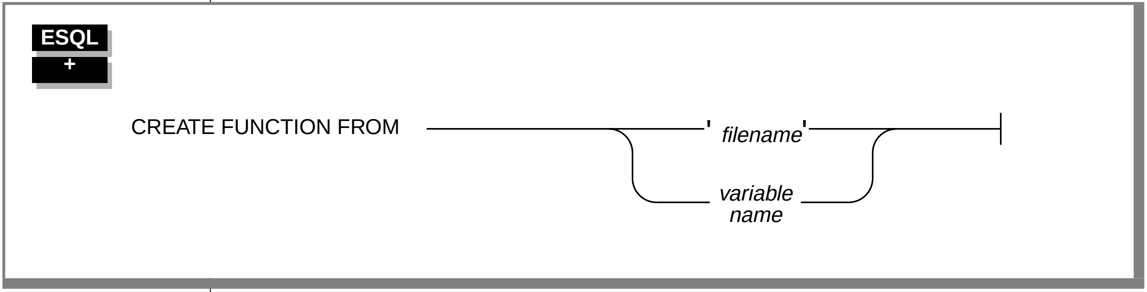
In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using SPL routines in Chapter 14.

In the *Extending INFORMIX-Universal Server: User-Defined Routines* manual, see the discussion of how to create and use external functions.

CREATE FUNCTION FROM

Use the CREATE FUNCTION FROM statement to create a new function. The actual text of the CREATE FUNCTION statement resides in a separate file.

Syntax



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of the file that contains the full text of a CREATE FUNCTION statement. The default pathname is the current directory.	The specified file must exist.	The pathname and filename must conform to the conventions of your operating system.
variable name	The name of a program variable that holds the value of *filename*	The file that is specified in the program variable must exist.	The name must conform to language-specific rules for variable names.

Usage

An INFORMIX-ESQL/C program cannot directly create a stored function or an external function. That is, it cannot contain the CREATE FUNCTION statement. However, you can create these functions within an ESQL/C program with the following steps:

- Create a source file with the CREATE FUNCTION statement.
- Use the CREATE FUNCTION FROM statement to send the contents of this source file to the database server for execution.

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 create the **delete_order()** stored function with the following CREATE FUNCTION FROM statement:

```
EXEC SQL create function from 'del_ord.sql';
```

The filename that you provide is relative. If you provide a simple filename (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. However, to improve readability of the code, Informix recommends that you match these two statements. If you are not sure whether the routine is a function or a procedure, use the CREATE ROUTINE FROM statement in the ESQL/C program.*

References

See the CREATE FUNCTION, CREATE PROCEDURE, CREATE PROCEDURE FROM, and CREATE ROUTINE FROM statements in this manual.

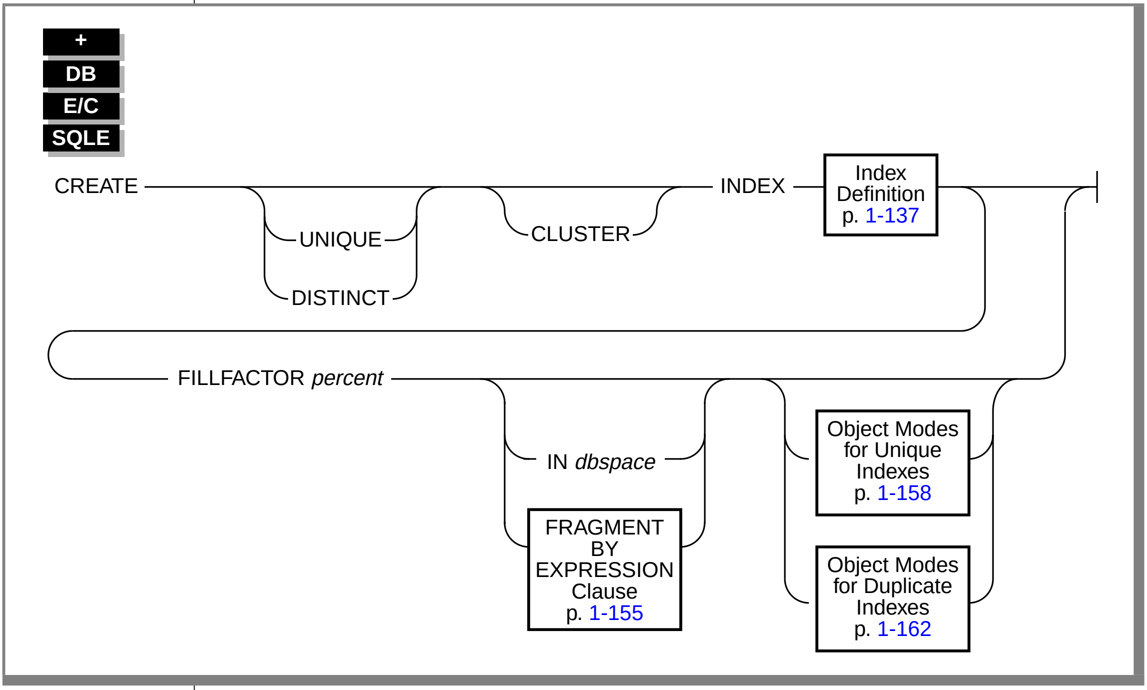
In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored functions in Chapter 14.

CREATE INDEX

Use the CREATE INDEX statement to create a new index for one or more columns in a table, a functional value on one or more columns, and, optionally, to cluster the physical table in the order of the index.

When more than one columns or functions are listed, the concatenation of the set of columns is treated as a single composite column for indexing. The indexes can be fragmented into separate dbspaces. You can create a unique or duplicate index, and you can set the object mode of either type of index.

Syntax


+
DB
E/C
SQLE
CREATE
UNIQUE
DISTINCT
CLUSTER
INDEX
Index Definition p. 1-137
FILLFACTOR *percent*
IN *dbspace*
FRAGMENT BY EXPRESSION Clause p. 1-155
Object Modes for Unique Indexes p. 1-158
Object Modes for Duplicate Indexes p. 1-162

Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace in which you want to place the index	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-962
percent	The percentage of each index page that is filled by index data when the index is created. The default value is 90.	Value must be in the range 1 to 100. Fillfactor does not apply to an R-tree secondary access method.	Literal Number, p. 1-997

Usage

A *secondary access method* (sometimes referred to as an *index access method*) is a set of 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. Typically, a secondary access method speeds up the retrieval of data.

Use CREATE INDEX to create the following types of indexes:

- Column index
- Functional index

  You can create a *functional index* on the resulting values of a function on one or more columns. For more information, see "Function Specification" on page 1-141.

When you issue the CREATE INDEX statement, the table is locked in exclusive mode. If another process is using the table, the database server cannot execute the CREATE INDEX statement and returns an error.

For the different secondary access methods that Universal Server provides, see "USING Clause" on page 1-148.

UNIQUE and DISTINCT Options

The following example creates a unique index:

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
```

A unique index prevents duplicates 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 would accomplish the same task:

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num)
```

The index in either example is maintained in ascending order, which is the default order.

If you do not specify the UNIQUE or DISTINCT keywords in a CREATE INDEX statement, a duplicate index is created. A duplicate index allows duplicate values in the indexed column.

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. See the CREATE TABLE or ALTER TABLE statements for more information on creating unique constraints.

How Unique and Referential Constraints Affect Indexes

The database server creates internal B-tree indexes for unique and referential constraints. If a unique or referential constraint is added after the table is created, the user-created indexes are used, if appropriate. An appropriate index is one that indexes the same columns that are used in the referential or unique constraint. If an appropriate index is not available, a nonfragmented index is created in the database dbspace.

CLUSTER Option

Use the CLUSTER option to reorder the physical table in the order designated by the index. The CREATE CLUSTER INDEX statement fails if a CLUSTER index already exists.

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode)
```

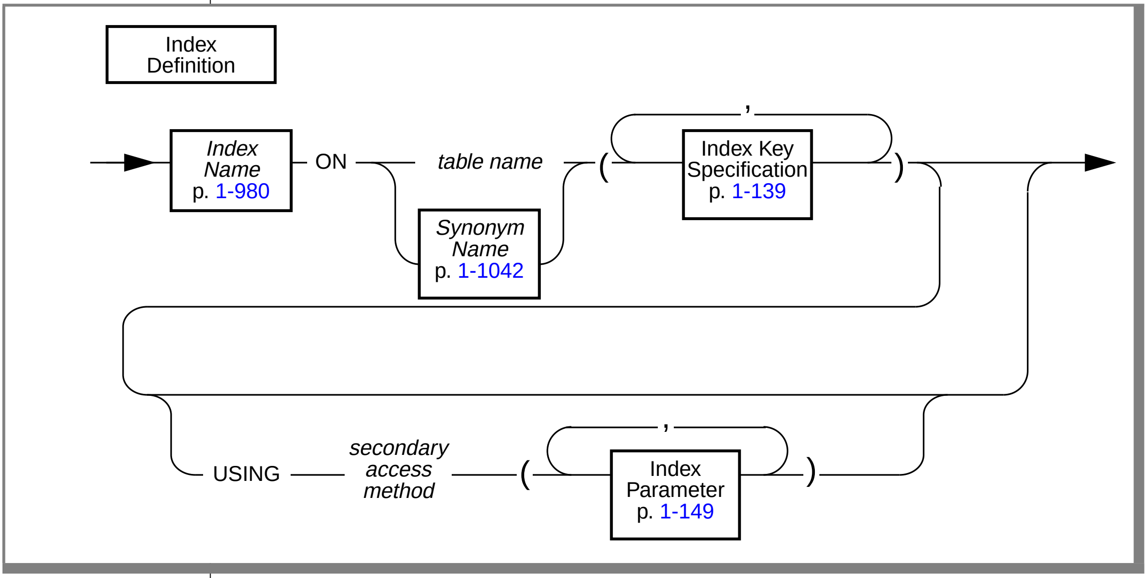
This statement creates an index on the **customer** table that orders the table physically by zip code.

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.

Warning: *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.*

Index Definition

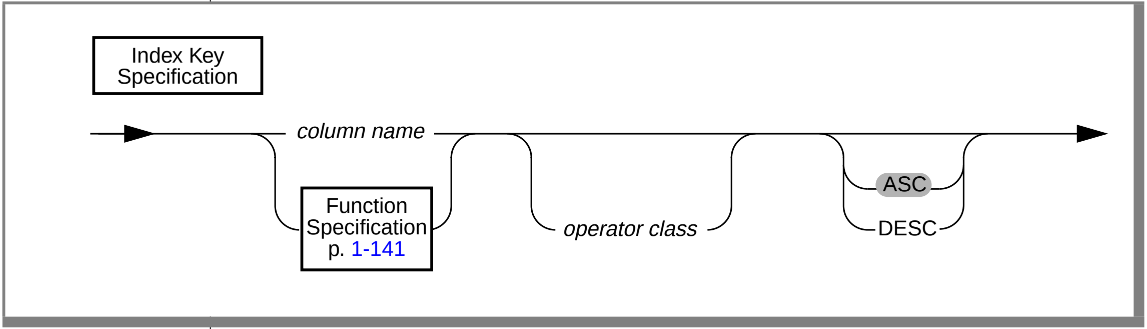
Use the Index Definition portion of the CREATE INDEX statement to give a name to the index, to specify the table on which the index is created, the value or values to use for the index key, and, optionally, the secondary access method.



Element	Purpose	Restrictions	Syntax
table name	The name of the table on which the index is created	The table must exist. The table can be a regular database table or a temporary table. This table cannot be an external table.	Table Name, p. 1-1044
secondary access method	The name of the secondary access method used with the index you are creating.	The access method can be a B-tree, R-tree, or an access method that has been defined by a DataBlade module. The access method 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 restriction does not apply to other secondary access methods.	Identifier, p. 1-962

Index Key Specification

Use the Index Key Specification clause 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 name	The name of the column or columns that you want to index	You must observe restrictions on the location of the columns, the maximum number of columns, the total width of the columns, existing constraints on the columns, and the number of indexes allowed on the same columns. See "Restrictions on the Column Name Variable in CREATE INDEX" on page 1-140.	Identifier, p. 1-962
operator class	The operator class associated with this column or function of the index	If you specify a secondary access method in the USING clause that does not have a default operator class, you must specify an operator class here. If you use an alternative access method, and if the access method has a default operator class, you can omit the operator class here. 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.	Identifier, p. 1-962

The index key value can be one of the following values:

- One or more columns that contain built-in data types
- 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

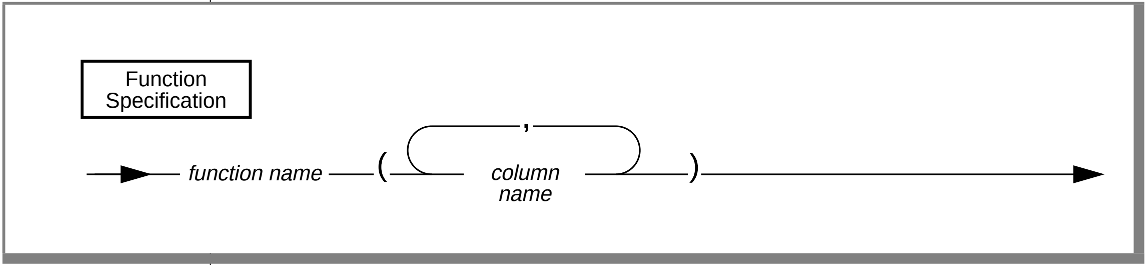
Restrictions on the Column Name Variable in CREATE INDEX

Observe the following restrictions when you specify the *column name* variable:

- All the columns you specify must exist and must belong to the same table—the table being indexed.
- You cannot create an index on a column that belongs to an external table.
- The column you specify cannot be a column whose data type is a collection.
- The maximum number of arguments (columns) you can specify is 16. See "Composite Indexes" on page 1-150.
- You cannot add an ascending index to a column or column list that already has a unique constraint on it. See "ASC and DESC Keywords" on page 1-142.
- The number of indexes you can create on the same column or same sequence of columns is restricted. See "Number of Indexes Allowed" on page 1-151.

Function Specification

This clause specifies the user-defined function whose return value is the key for a functional index.



Element	Purpose	Restrictions	Syntax
function name	The name of the function used as a key to this index	This must be a non-variant function. The return type of the function cannot be BYTE or TEXT. You cannot create an index on built-in algebraic, exponential, log, or hex functions.	Function Name, p. 1-959
column name	The name of the column or columns on which the function acts	See "Restrictions on the Column Name Variable in CREATE INDEX" on page 1-140.	Identifier, p. 1-962

You can create an index on an external function or an SPL function. You can also create functional indexes within an SPL routine.

A functional index can be a B-tree index or a user-defined index type provided by a DataBlade module.

Functional indexes are indexed on the value returned by the specified function rather than on the value of a column.

For example, the following statement creates a functional index on table **zones** using the value returned by the function **Area()** as the key:

```
CREATE INDEX zone_func_ind ON zones (Area(length,width));
```

Operator Class

An *operator class* is the set of operators that Universal Server associates with a secondary access method for query optimization and building the index.

Specify an operator class when you create an index if you have one of the following situations:

- There is no default operator class for the secondary access method. For example, some of the DataBlade modules do not provide a default operator class.
- You want to use an operator class that is different from the default operator class that the secondary access method provides.

For more information, see "Default Operator Classes" on page 1-176. 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);
```

ASC and DESC Keywords

Use the ASC option to specify an index that is maintained in ascending order. The ASC option is the default ordering scheme. Use the DESC option to specify an index that is maintained in descending order. 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.

The ASC and DESC options can be used with B-trees only.

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 allowed:

```
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 cathtmp ON customer (customer_num DESC)
CREATE INDEX c_temp2 ON customer (customer_num, zipcode)
```

Bidirectional Traversal of Indexes

When you create an index on a column but do not specify the ASC or DESC keywords, the database server stores the key values in ascending order by default. If you specify the ASC keyword, the database server stores the key values in ascending order. If you specify the DESC keyword, the database server stores the key values in descending order.

Ascending order means that the key values are stored in order from the smallest key to the largest key. For example, if you create an ascending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Albertson, Beatty, Currie.`

Descending order means that the key values are stored in order from the largest key to the smallest key. For example, if you create a descending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Currie, Beatty, Albertson.`

However, 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.

Example of Bidirectional Traversal of an Index

An example can help to illustrate the bidirectional traversal of indexes by the database server. Suppose that you want to enter the following two queries:

```
SELECT lname, fname FROM customer ORDER BY lname ASC;
SELECT lname, fname FROM customer ORDER BY lname DESC;
```

When you specify the ORDER BY clause in SELECT statements such as these, you can improve the performance of the queries by creating an index on the ORDER BY column. Because of the bidirectional traversal capability of the database server, you only need to create a single index on the **lname** column.

For example, you can create an ascending index on the **lname** column with the following statement:

```
CREATE INDEX lname_bothways ON customer (lname ASC)
```

The database server will use the ascending index **lname_bothways** to sort the results of the first query in ascending order and to sort the results of the second query in descending order.

In the first query, you want to sort the results in ascending order. So the database server traverses the pages of the **lname_bothways** index from left to right and retrieves key values from the smallest key to the largest key. The query result is as follows.

lname	fname
Albertson	Frank
Beatty	Lana
Currie	Philip
. . .	
Vector	Raymond
Wallack	Jason
Watson	George

Traversing the index from left to right means that the database server starts at the leftmost leaf node of the index and continues to the rightmost leaf node of the index.

In the second query, you want to sort the results in descending order. So the database server traverses the pages of the **lname_bothways** index from right to left and retrieves key values from the largest key to the smallest key. The query result is as follows.

lname	fname
Watson	George
Wallack	Jason
Vector	Raymond
. . .	
Currie	Philip
Beatty	Lana
Albertson	Frank

Traversing the index from right to left means that the database server starts at the rightmost leaf node of the index and continues to the leftmost leaf node of the index. For an explanation of leaf nodes in indexes, see the *INFORMIX-Universal Server Administrator's Guide*.

Choosing an Ascending or Descending Index

In the preceding example, you created an ascending index on the **lname** column of the **customer** table by specifying the ASC keyword in the CREATE INDEX statement. Then the database server used this index to sort the results of the first query in ascending order of **lname** values and to sort the results of the second query in descending order of **lname** values. However, you could have achieved exactly the same results if you had created the index as a descending index.

For example, the following statement creates a descending index that the database server can use to process both queries:

```
CREATE INDEX lname_bothways2 ON customer (lname DESC)
```

The resulting **lname_bothways2** index stores the key values of the **lname** column in descending order, from the largest key to the smallest key. When the database server processes the first query, it traverses the index from right to left to perform an ascending sort of the results. When the database server processes the second query, it traverses the index from left to right to perform a descending sort of the results.

So 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.

Use of the ASC and DESC Keywords in Composite Indexes

If you want to place an index on a single column of a table, you do not need to specify the ASC or DESC keywords because the database server can traverse the index in either ascending or descending order. The database server will create the index in ascending order by default, but the database server can traverse this index in either ascending or descending order when it uses the index in a query.

However, if you create a composite index on a table, 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 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);
```

Now, when you execute the query, the database server uses the index that you created (either **stock_idx1** or **stock_idx2**) to sort the query results in ascending order by the value of the **manu_code** column and then in descending order by the value of the **unit_price** column. If you created the **stock_idx1** index, the database server traverses the index from left to right when it executes the query. If you created the **stock_idx2** index, the database server traverses the index from right to left when it executes the query.

Regardless of which index you created, the query result is as follows.

stock_num	manu_code	description	unit_price
8	ANZ	volleyball	$840.00
205	ANZ	3 golf balls	$312.00
110	ANZ	helmet	$244.00
304	ANZ	watch	$170.00
301	ANZ	running shoes	$95.00
310	ANZ	kick board	$84.00
201	ANZ	golf shoes	$75.00
313	ANZ	swim cap	$60.00
6	ANZ	tennis ball	$48.00
9	ANZ	volleyball net	$20.00
5	ANZ	tennis racquet	$19.80
309	HRO	ear drops	$40.00
302	HRO	ice pack	$4.50
. . .			
113	SHM	18-spd, assmbld	$685.90
1	SMT	baseball gloves	$450.00
6	SMT	tennis ball	$36.00
5	SMT	tennis racquet	$25.00

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);
```

USING Clause

Use the USING clause to specify the secondary access method to use for the new index. A *secondary access method* is a set of routines that perform all of the operations needed to make an index available to a server, such as create, drop, insert, delete, update, and scan.

Universal 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 of a database.
- The R-tree secondary access method is a registered secondary access method.

  An R-tree index is good for searches on multi-dimensional data (such as box, circle, and so forth). The database server registers this secondary access method as **rtree** in the system catalog tables of a database.

Important: *To use an R-tree index, you must install a spatial DataBlade module such as the 2D DataBlade module, Geodetic DataBlade, or any other 3rd party DataBlade modules that implement the R-tree index. These DataBlade modules implement the R-tree secondary access method.*

DataBlade modules might provide other types of secondary access methods. For more information on these other secondary access methods, refer to the DataBlade user guides.

By default, the CREATE INDEX statement creates a generic B-tree index. If you want to create an index with an secondary access method other than B-tree, you must specify that name of the secondary access method in the USING clause.

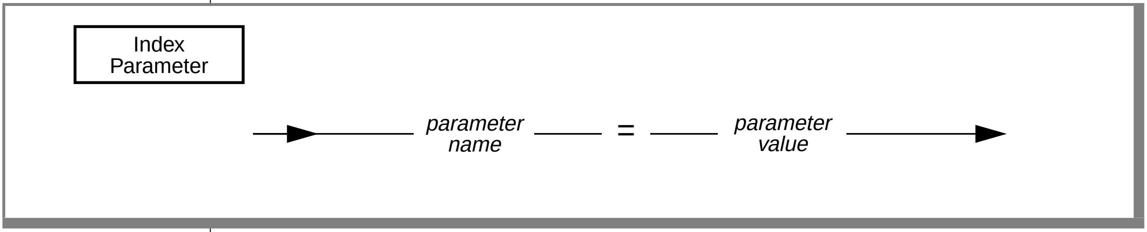
The following example assumes that the database implements the R-tree index. It creates an R-tree index on the **location** column that contains a opaque data type, **point**.

```
CREATE INDEX loc_ix ON TABLE emp (location)
    USING rtree;
SELECT name FROM emp
    WHERE location N_equator_equals point('500, 0');
```

The sample query has a filter on the **location** column.

Index Parameter

Some DataBlade modules provide indexes that require specific parameters when you create them.



Element	Purpose	Restrictions	Syntax
parameter name	Name of the secondary access method parameter used with this index	The parameter name must be one of the strings allowed for this secondary access method. For more information, refer to the DataBlade module user guide.	Quoted String, p. 1-1010
parameter value	Value of the specified parameter	The parameter value must be one of the quoted strings or literal numbers allowed for this secondary access method.	Quoted String, p. 1-1010 or Literal Number, p. 1-997

Example of an Index with Parameters

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’);
```

Composite Indexes

A composite index can have up to 16 key parts. An *index key part* is either a table column or, if the index is a functional index, the result of a function on one or more table columns. A composite index can have any of the following 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 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 index prevents any duplicates of a given combination of **stock_num** and **manu_code**. The index is in ascending order by default.

The total width of all key parts in a single CREATE INDEX statement cannot exceed 390 bytes. Place key parts in the composite index in the order from most frequently used to least frequently used.

Number of Indexes Allowed

Restrictions exist on the number of indexes that you can create on the same column or the same sequence of columns.

Restrictions on the Number of Indexes on a Single Column

You can create only one ascending index and one descending index on a single column. For example, if you wanted to create all possible indexes on the **stock_num** column of the **stock** table, you could create the following indexes:

- The **stock_num_asc** index on the **stock_num** column in ascending order
- The **stock_num_desc** index on the **stock_num** column in descending order

Because of the bidirectional traversal capability of the database server, you do not need to create both indexes in practice. You only need to create one of the indexes. Both of these indexes would achieve exactly the same results for an ascending or descending sort on the **stock_num** column. For further information on the bidirectional traversal capability of the database server, see "Bidirectional Traversal of Indexes" on page 1-143.

Restrictions on the Number of Indexes on a Sequence of Columns

You can create multiple indexes on a sequence 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 the following 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 in practice. You only need to create two indexes:

- The **ix1** and **ix2** indexes achieve exactly the same results for sorts in which the user specifies the same sort direction (ascending or descending) for both columns. Therefore, you only need to create one index of this pair.
- The **ix3** and **ix4** indexes achieve exactly 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). Therefore, you only need to create one index of this pair.

For further information on the bidirectional-traversal capability of the database server, see "Bidirectional Traversal of Indexes" on page 1-143.

FILLFACTOR Clause

Use the FILLFACTOR clause to provide for expansion of a B-tree index at a later date or to create compacted indexes. You provide a percent value ranging from 1 to 100, inclusive. The default percent value is 90.

When the B-tree index is created, Universal Server initially fills only that percentage of the nodes specified with the FILLFACTOR value. If you provide a low percentage value, such as 50, you allow room for growth in your B-tree index. The nodes of the B-tree 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 growth. A low percentage value can result in faster inserts and can be used for indexes that you expect to grow.

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.

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 with **ONCONFIG**, see the *INFORMIX-Universal Server Administrator's Guide*.

Indexes on Fragmented and Nonfragmented Tables

When you fragment a table and, at a later time, create an index for that table, the index uses the same fragmentation strategy as the table unless you specify otherwise with the FRAGMENT BY EXPRESSION clause or the IN *dbspace* clause. Any changes to the table fragmentation result in a corresponding change to the index fragmentation.

In Universal Server, all indexes are detached. When indexes are created with a fragmentation strategy or a dbspace is specified in the IN *dbspace* clause, the indexes are stored in separate dbspaces from the table. If there is no fragmentation scheme and no dbspace is specified in the IN *dbspace* clause, the index is created in the same dbspace as the table.

For information on the IN *dbspace* clause, see "IN dbspace Clause". For information on the FRAGMENT BY EXPRESSION clause, see page 1-155.

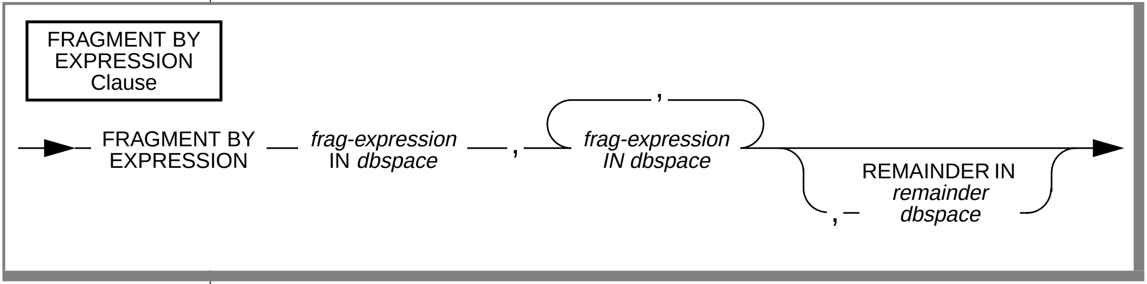
IN dbspace Clause

Use the IN *dbspace* clause to specify the dbspace where you want your index to reside. With this clause, you create a detached index, even though the index is not fragmented. The dbspace that you specify must already exist. If you do not specify the IN *dbspace* clause, the index is created in the dbspace where the table was created. In addition, if you do not specify the IN *dbspace* clause, but the underlying table is fragmented, the index is created as a detached index, subject to all the restrictions on fragmented indexes. See page 1-155 for more information about fragmented indexes.

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
```

FRAGMENT BY EXPRESSION Clause


FRAGMENT BY EXPRESSION Clause
FRAGMENT BY EXPRESSION
frag-expression IN dbspace
,
,
frag-expression IN dbspace
REMAINDER IN remainder dbspace
,

Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that will contain an index fragment that *frag-expression* defines	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces. The dbspaces must exist at the time you execute the statement.	Identifier, p. 1-962
frag-expression	An expression that defines a fragment where an index key is to be stored using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values 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. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in a fragment expression.	Expression, p. 1-876, and Condition, p. 1-831

(1 of 2)

Element	Purpose	Restrictions	Syntax
remainder dbspace	The dbspace that contains index keys that do not meet the conditions defined in any fragment expression	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-962

(2 of 2)

You use the FRAGMENT BY EXPRESSION clause to define the expression-based distribution scheme.

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a dbspace. Each fragment expression within the rule isolates data and aids the database server in searching for index keys. You can specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which index keys are placed in a fragment, as the following example shows:

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

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

  ```
  .. . .
     FRAGMENT BY EXPRESSION
     MOD(id_num, 3) = 0 IN dbsp1,
     MOD(id_num, 3) = 1 IN dbsp2,
     MOD(id_num, 3) = 2 IN dbsp3;
  ```

- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of 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: *When you specify a date value in a fragment expression, 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 the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Creating Index Fragments

When you fragment a table, all indexes for the table become fragmented the same as the table, unless you specify a different fragmentation strategy.

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 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.

Fragmentation of Indexes on Temporary Tables

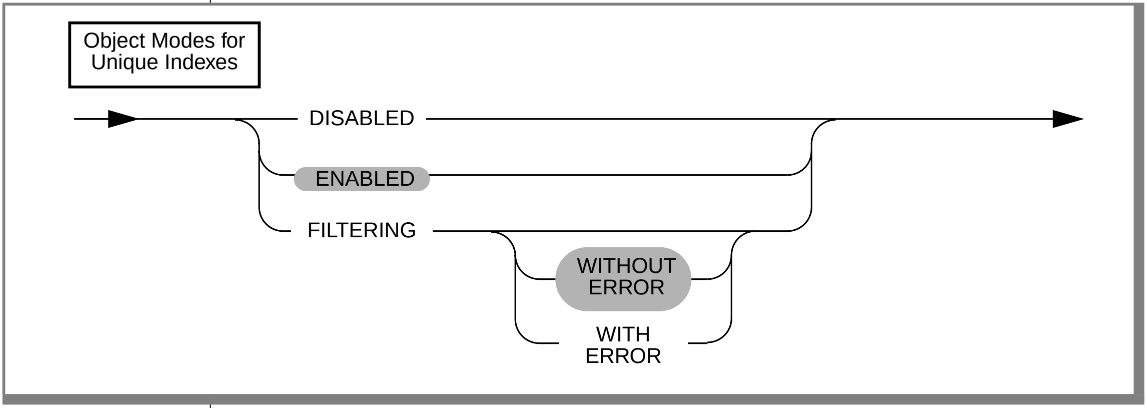
You can create explicit temporary tables with the TEMP TABLE clause of the CREATE TABLE statement or with the INTO TEMP clause of the SELECT statement. If you specified more than one dbspace in the **DBSPACETEMP** environment variable, but you did not specify an explicit fragmentation strategy, the database server fragments the temporary table round-robin across the dbspaces that **DBSPACETEMP** specifies.

If you then try to create a unique index on the temporary table, but you do not specify a fragmentation strategy for the index, the index is not fragmented in the same way as the table. You can fragment a unique index only if the underlying table uses an expression-based distribution scheme, but the temporary table is fragmented according to a round-robin distribution scheme.

Instead of fragmenting the unique index on the temporary table, the database server creates the index in the first dbspace that the **DBSPACETEMP** environment variable specifies. To avoid this result, use the FRAGMENT BY EXPRESSION clause to specify a fragmentation strategy for the index.

For more information on the **DBSPACETEMP** environment variable, see the *Informix Guide to SQL: Reference*.

Object Modes for Unique Indexes



You can set unique indexes in the following modes: disabled, enabled, and filtering. The following list explains these modes.

Object Mode	Effect
disabled	A unique index created in disabled mode is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.
enabled	A unique index created in enabled mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a unique enabled index, the statement fails.
filtering	A unique index created in filtering mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the filtering mode index are up to date, the optimizer uses the index during the execution of queries. 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 mode, you can also specify one of the following error options.

Error Option	Effect
WITHOUT ERROR	When a unique-index violation occurs during an insert or update operation, no integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.
WITH ERROR	When a unique-index violation occurs during an insert or update operation, an integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.

Specifying Object Modes for Unique Indexes

You must observe the following rules when you specify object modes for unique indexes in CREATE INDEX statements:

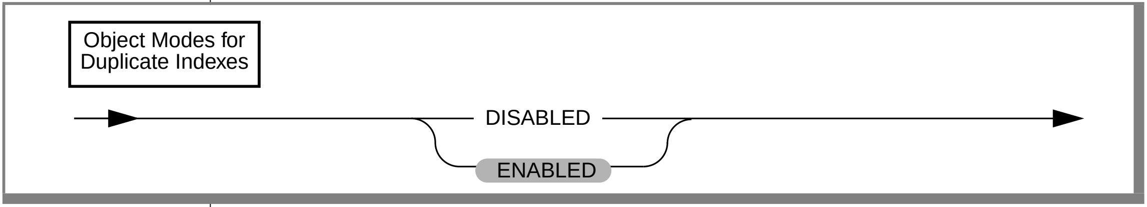
- You can set a unique index to the enabled, disabled, or filtering modes.
- If you do not specify the object mode of a unique index explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering-mode unique index, the default error option is WITHOUT ERROR.
- When you add a new unique index to an existing base table and specify the disabled object 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-object 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 statement to switch the object mode of the index to the enabled mode. 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 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.

Object Modes for Duplicate Indexes



If you create a duplicate index, you can set the object mode of the index to the disabled or enabled mode. The following table explains these modes.

Object Mode	Effect
disabled	A duplicate index is created in disabled mode. The disabled index is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.
enabled	A duplicate index is created in enabled mode. The enabled index is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a duplicate enabled index, the statement does not fail.

Specifying Object Modes for Duplicate Indexes

You must observe the following rules when you specify object modes for duplicate indexes in CREATE INDEX statements:

- You can set a duplicate index to the enabled or disabled mode, but you cannot set a duplicate index to the filtering mode.
- If you do not specify the object mode of a duplicate index explicitly, the default mode is enabled.

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 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. So 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 (not disabled) unique, foreign-key, or primary-key constraint on a column or set of columns if the indexes needed by the active constraint exist and are disabled.

References

See the ALTER INDEX, CREATE OPCLASS, DROP INDEX, and CREATE TABLE statements in this manual.

For a more detailed description of the different types of indexes, refer to Chapter 3 of the *INFORMIX-Universal Server Performance Guide*. For information about when to use the different types of indexes and other performance issues with indexes, refer to Chapter 4 of the *INFORMIX-Universal Server Performance Guide*.

For information about operator classes, refer to the CREATE OPCLASS statement and the *Extending INFORMIX-Universal Server: Data Types* manual.

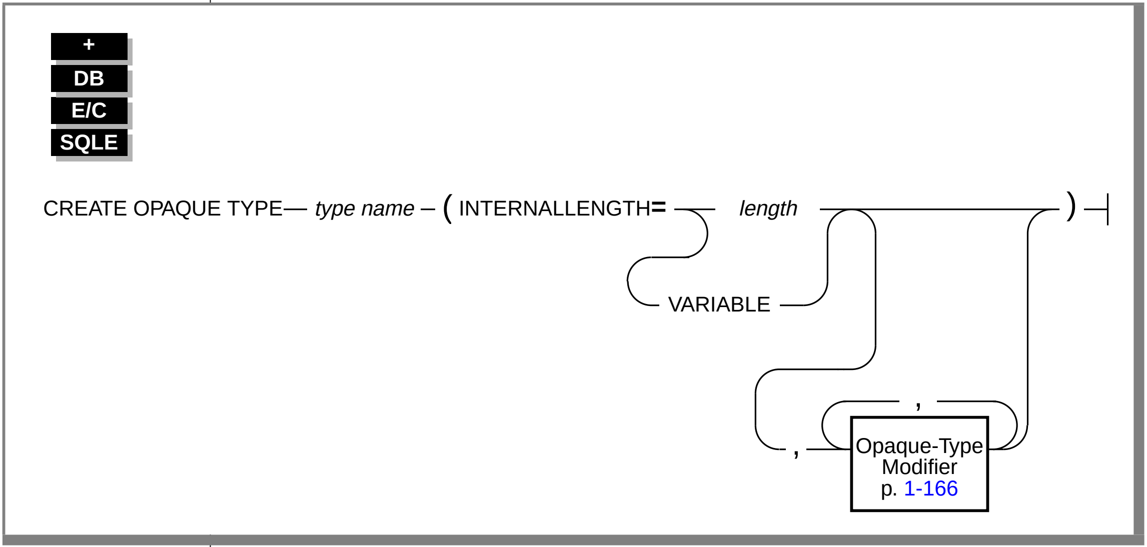
For information about the GLS aspects of the CREATE INDEX statement, refer to the *Guide to GLS Functionality*.

For information about the indexes provided by DataBlade modules, refer to your DataBlade module user's guide.

CREATE OPAQUE TYPE

Use the CREATE OPAQUE TYPE statement to create an opaque data type.

Syntax



Element	Purpose	Restrictions	Syntax
type name	The name of the new opaque data type	The name you specify must follow the conventions of SQL identifiers. In an ANSI-compliant database, the combination *owner.type* must be unique within the database. In a database that is not ANSI compliant, the *type name* must be unique within the database.	Identifier, p. 1-962 Data Type, p .1-855
length	The number of bytes needed by the database server to store a value of a fixed-length opaque data type	The number must match the positive integer reported when the C language **sizeof()** directive is applied to the type structure.	Literal Number, p. 1-997

Usage

The CREATE OPAQUE TYPE statement registers a new opaque data type in the database. Universal Server stores information on extended data types, including opaque types, in the **sysxtdtypes** system catalog table.

Naming an Opaque Data Type

The actual name of an opaque data type is an SQL identifier. When you create an opaque data type, the *type name* must be unique within a database. The *type name* cannot be the same as any distinct-type names or named row-type names.

ANSI

When you create an opaque data type in an ANSI-compliant database, *owner.type_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 opaque-type owner is stored in uppercase letters. ♦

Privileges on an Opaque Data Type

To create a new opaque type within a database, you must have the Resource privilege on the database. The CREATE OPAQUE TYPE statement creates a new opaque type with Usage privilege granted to the owner of the opaque type and the DBA. To use the opaque data type in an SQL statement, you must have Usage privilege. The owner can grant Usage privilege to other users with the USAGE ON TYPE clause of the GRANT statement. For more information, see the GRANT statement on page 1-458.

INTERNALLENGTH Modifier

The CREATE OPAQUE TYPE statement must indicate the name of the opaque type and its internal length. The INTERNALLENGTH modifier specifies the size of an opaque data type. The way you specify the internal length defines whether the opaque data type is fixed length or varying length.

Fixed-Length Opaque Data Types

A fixed-length opaque type has an internal structure that has a fixed size. To create a fixed-length opaque data type, specify the size of the internal structure, in bytes, for the INTERNALLENGTH modifier. The following statement creates a fixed-length opaque type called **fixlen_typ**. The database server allocates 8 bytes for this type.

```
CREATE OPAQUE TYPE fixlen_typ(INTERNALLENGTH=8, CANNOTHASH)
```

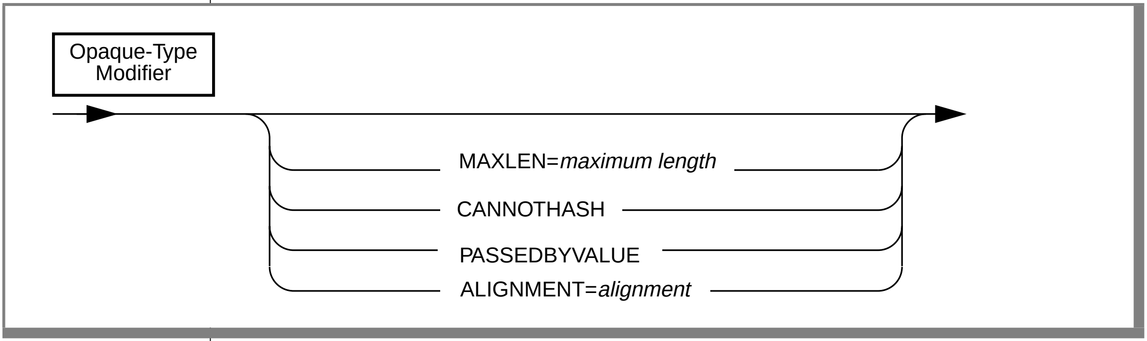
Varying-Length Opaque Data Types

A varying-length opaque data type has an internal structure whose size might vary from one instance of the opaque type 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 for 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
maximum length	For varying-length opaque types, the maximum length in bytes, to allocate for instances of the type. Values that exceed this length return errors.	The length must be a positive integer less than or equal to 32K. Do not specify for fixed-length types. If maximum length is not specified for a variable-length type, the system default is 2 Kilobytes.	Literal Number, p.1-997
alignment	The byte boundary on which the database server aligns the opaque type when passing it to a user-defined routine	The alignment must be 1, 2, 4, or 8, depending upon the C definition of the opaque type and the hardware and compiler used to build the object file for the type. If alignment is not specified, the system default is 4 bytes.	Literal Number, p.1-997

Use modifiers to specify the following optional information:

- MAXLEN specifies the maximum length for varying-length opaque data types.
- CANNOTHASH specifies that the database server cannot use a hash function on the opaque type.

  You must provide an appropriate hash function for the database server to evaluate GROUP BY clauses on the type.
- PASSEDBYVALUE specifies that an opaque type of four bytes or fewer is passed by value.

  By default, opaque types are passed to user-defined routines by reference.
- ALIGNMENT specifies the byte boundary on which the database server aligns the opaque type.

Defining an Opaque Data Type

To define the opaque data type to the database server, you must provide the following information in the C language:

- A data structure that serves as the internal storage of the opaque data type

  The internal storage details of the data type are hidden, or opaque. Once you define a new opaque type, the database server can manipulate it without knowledge of the C 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 programming language.

- Optional additional routines that can be called by other support functions or by end users to operate on the opaque data type

  Possible additional functions include operator functions and casts that operate on the opaque data type. You can also write SQL functions for an opaque data type; SQL functions can appear within an SQL statement.

The following table summarizes the support functions for an opaque data type.

Function	Purpose	When Invoked
input	Converts the opaque data 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 data 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.

(1 of 3)

Function	Purpose	When Invoked
receive	Converts the opaque data 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.
send	Converts the opaque data 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.
import	Performs any tasks need to convert from the external (character) representation of an opaque type to the internal representation for a bulk copy.	When DB-Access (LOAD) or the High Performance Loader initiates a bulk copy from a text file to a database.
export	Performs any tasks need to convert from the internal representation of an opaque type to the external (character) representation 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 need 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.
exportbinary	Performs any tasks need 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()	Does any processing required before storing the opaque type to disk. This function must be named **assign()**.	When the database server executes an INSERT, UPDATE, and LOAD statement, *before* it stores the opaque type to disk.
destroy()	Does any processing necessary before removing a row that contains the opaque type. This function must be named **destroy()**.	When the database server executes the DELETE and DROP TABLE statements, *before* it removes the opaque type from disk.

(2 of 3)

Function	Purpose	When Invoked
lohandles()	Returns a list of the LO-pointer structures (pointers to smart large objects) in an opaque data type.	Whenever the database server must search opaque types for references to smart large objects: when the **oncheck** utility runs, when 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 it executes the CREATE INDEX statement to create a B-tree index.

(3 of 3)

Once 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.

When you have written the necessary source code to define the opaque data type, you then use the CREATE OPAQUE TYPE statement to register the opaque type in the database.

References

See the CREATE CAST, CREATE DISTINCT TYPE, CREATE FUNCTION, CREATE ROW TYPE, CREATE TABLE, and DROP TYPE statements in this manual.

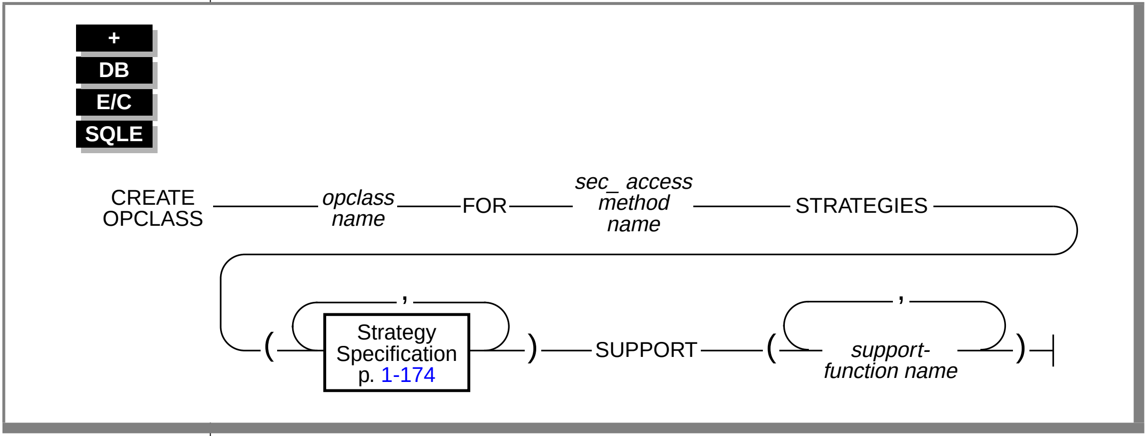
For a summary of an opaque data type, see Chapter 2 of the *Informix Guide to SQL: Reference*. For information on how to define an opaque data type, see the *Extending INFORMIX-Universal Server: Data Types* manual.

For information about the GLS aspects of the CREATE OPAQUE TYPE statement, refer to the *Guide to GLS Functionality*.

CREATE OPCLASS

Use the CREATE OPCLASS statement to create an *operator class* for a *secondary access method*.

Syntax



Element	Purpose	Restrictions	Syntax
opclass name	Name of the operator class being created	The operator class name must be unique within the database. In an ANSI-compliant database, the combination *owner.opclassname* must be unique within the database.	Identifier, p. 1-962
sec_access method name	Name of the secondary access method with which the specified operator class is being associated	The secondary access method must already exist and must be registered in the **sysams** system catalog table. The database server provides the B-tree and R-tree secondary access method.	Identifier, p. 1-962
support-function name	Name of a support function required by the specified secondary access method	The support functions must be listed in the order expected by the specified access method.	Identifier, p. 1-962

Usage

An *operator class* is the set of operators that Universal Server associates with the *sec_ access method name* 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 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.

You define a new operator class when you want:

- an index to use a different order for the data than the sequence provided by the default operator class.
- 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)
```

For more information on the **btree** secondary access method, see “Default Operator Classes” on page 1-176.

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 statement, the **abs_btree_ops** operator class has five strategy functions.

- Support functions

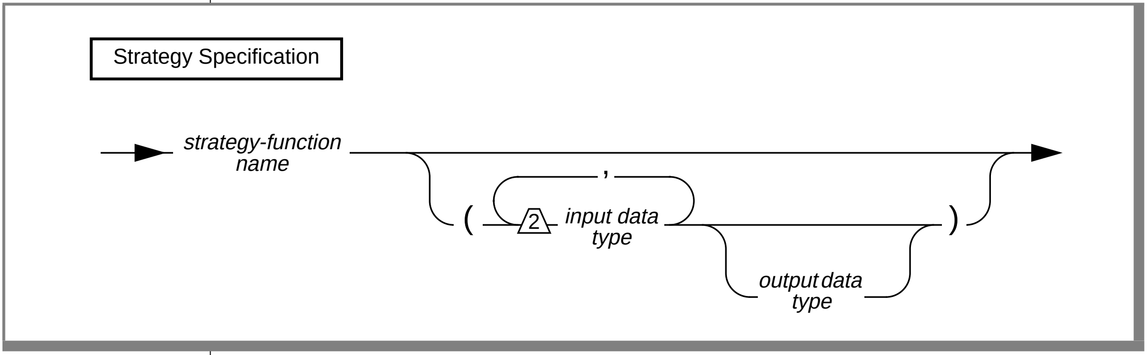
  Specify support functions of an operator class in the SUPPORT clause of the CREATE OPCLASS statement. In the preceding CREATE OPCLASS statement, the **abs_btree_ops** operator class has one support function.

STRATEGY 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 the *INFORMIX-Universal Server Performance Guide*.

When you create a new operator class, you specify the strategy functions for the secondary access method in the STRATEGY 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, refer to the *Extending INFORMIX-Universal Server: Data Types* manual.

Strategy Specification



Element	Purpose	Restrictions	Syntax
input data type	Data type of the input argument for the strategy function	This is the data type 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. 1-855
output data 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. 1-855
strategy-function name	The name of an strategy function to associate with the specified operator class	The operators must be listed in the order expected by the specified secondary access method. For more information, refer to the user's guide of the DataBlade module that provides the secondary access method.	Identifier,1-962

The *strategy_function name* function is an external function. The CREATE OPCLASS statement does not verify that a user-defined function of *strategy_function name* exists. However, for the secondary access method to use the *strategy_function name* function, this function must be:

- compiled in a shared library.
- registered in the database with the CREATE FUNCTION statement.

You can optionally the signature of an strategy function in addition to its name. A strategy function can only take two input parameters and an optional output parameter. To specify the function signature, you specify:

- an *input data type* for each of the two input parameter 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 user-defined data types as well as built-in 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.

Side-Effect Indexes

Side-effect data is additional data that a strategy function returns when Universal Server executes a query containing 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 the side effect data, along with the qualifying images.

SUPPORT Clause

Support functions are functions that the secondary access method uses internally to build and search the index. You specify the support functions for the secondary access method in the SUPPORT clause of the CREATE OPCLASS statement. 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 1-176.

The *support_function name* function is an external function. The CREATE OPCLASS statement does not verify that a user-defined function of *support_function name* exists. However, for the secondary access method to use the *support_function name* function, this 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 creates 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 Universal 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 of a database. 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 Spatial DataBlade module, Geodetic DataBlade, or any other third-party DataBlade module that implements the R-tree index. These DataBlade modules implement the R-tree operator-class functions.*

For information on the operator-class functions of these operator classes, refer to the chapter on operator classes in the *Extending INFORMIX-Universal Server: Data Types* manual.

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 the DataBlade user guides.

References

See the CREATE FUNCTION and DROP OPCLASS statements in this manual. For more information on how to specify a secondary access method or an operator class for an index, see the CREATE INDEX statement in this manual.

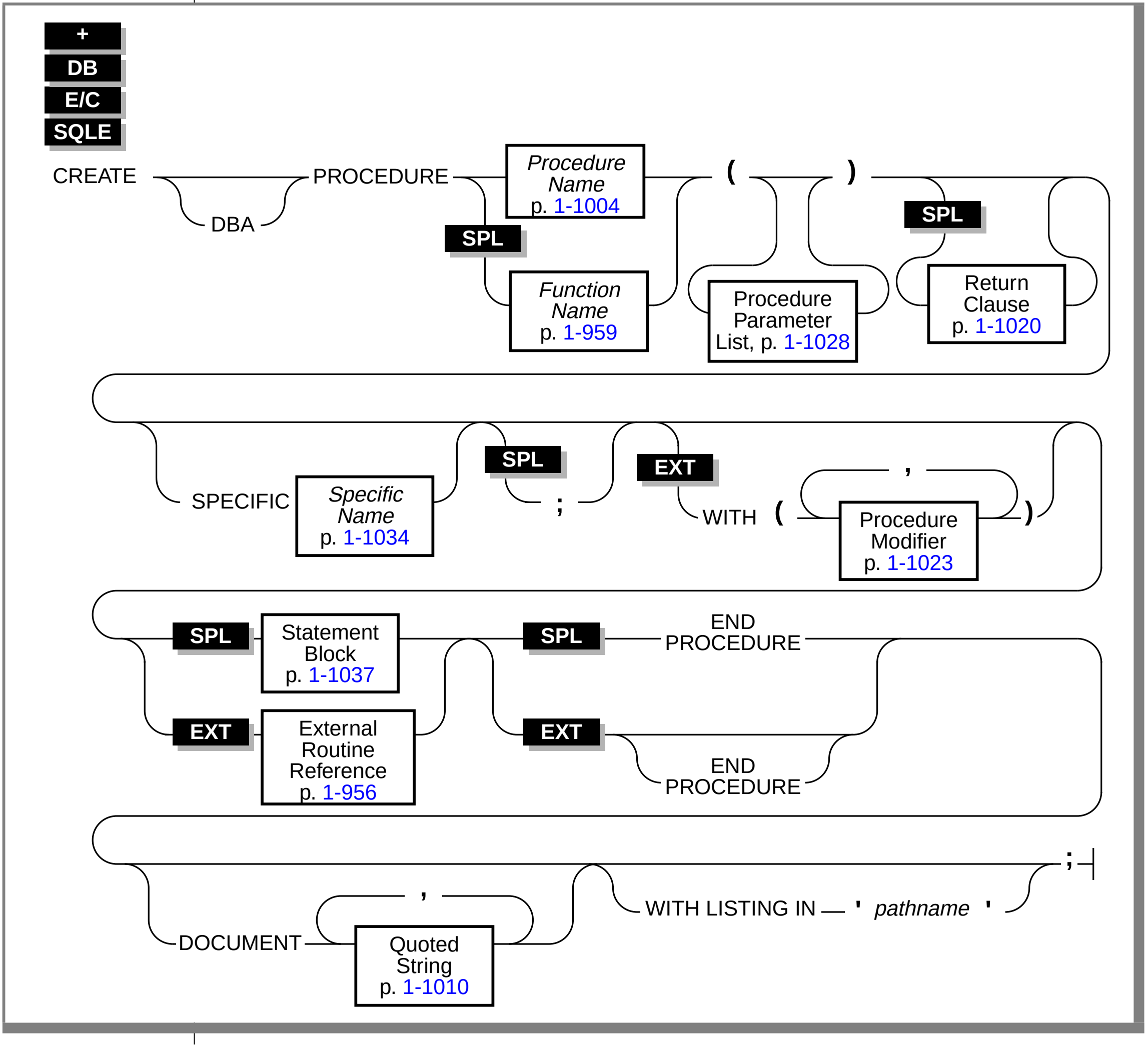
For information on how to create and extend an operator class, see the *Extending INFORMIX-Universal Server: Data Types* manual.

For information about the GLS aspects of the CREATE OPCLASS statement, refer to the *Guide to GLS Functionality*.

CREATE PROCEDURE

Use the CREATE PROCEDURE statement to register an external procedure or to write and register an SPL procedure.

Syntax



Element	Purpose	Restrictions	Syntax
pathname	The pathname to a file in which compile-time warnings are stored	The specified pathname must exist on the computer where the database resides.	The pathname and filename must conform to the conventions of your operating system.

Usage

A *procedure* is a user-defined routine that can accept arguments but does not return a value. INFORMIX-Universal Server supports procedures written in the following languages:

- Stored Procedure Language (*SPL procedures*)
- One of the external languages (such as C) that INFORMIX-Universal Server supports (*External procedures*)

The entire length of a CREATE PROCEDURE statement must be less than 64 kilobytes. This length is the literal length of the statement, including blank space and tabs.

Routines, Functions, and Procedures

In INFORMIX-Universal Server, *routine* is a generic term that includes both procedures and functions. A *procedure* is a routine that can accept arguments but does not return any values. A *function* is routine that can accept arguments and returns one or more values. Universal Server treats any routine that includes a Return clause as a function.

Legacy Procedures

SPL

In earlier Informix products, the term *stored procedure* was used for both SPL procedures and SPL functions. As a result, you may have created functions with CREATE PROCEDURE in the past. For backward compatibility with earlier products, you can continue to create SPL functions with CREATE PROCEDURE. However, with Universal Server, Informix recommends that you use CREATE PROCEDURE only with procedures and CREATE FUNCTION only with functions.

For more information on CREATE FUNCTION, see page 1-122. ♦

SPL

SPL Procedures

SPL procedures are routines written in Stored Procedure Language (SPL) that do not return a value.

Use one CREATE PROCEDURE statement, with SQL and SPL statements embedded between CREATE PROCEDURE and END PROCEDURE, to write and register an SPL procedure. Unlike external procedures, you do not need to write the procedure and register it in separate steps.

SPL procedures are parsed, optimized (as far as possible), and stored in the system catalog tables in executable format. The body of an SPL procedure is stored in the **sysprocbody** system catalog table. Other information about the procedure is stored in other system catalog tables, including **sysprocedures**, **sysprocplan**, and **sysprocauth**. For more information about these system catalog tables, see Chapter 1, "System Catalog," in the *Informix Guide to SQL: Reference*.

You must use the END PROCEDURE keywords with an SPL procedure.

If you use a Return clause or a Specific Name clause, place a semicolon after the clause immediately before the SPL statement block. If you do not use a Return clause or a Specific Name clause, do not place a semicolon after the CREATE PROCEDURE statement. Always place a semicolon at the end of the entire statement, after the END PROCEDURE, DOCUMENT, or WITH LISTING IN clause.

Example

The following example creates a 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';
```

For more information on writing SPL procedures, see Chapter 14, "Creating and Using SPL Routines," in the *Informix Guide to SQL: Tutorial*. ♦

EXT

External Procedures

External procedures are procedures you write in an external language that the Universal Server supports. To create external procedures, follow these steps:

1. Write the procedure in an external language, such as C, that Universal Server supports.
2. Compile the procedure and store the compiled code in a shared library.
3. Register the procedure in the database server with the CREATE PROCEDURE statement.

When Universal Server executes an external procedure, the database server invokes the external object code.

Universal Server does *not* store the body of an external procedure directly in the database, as it does for SPL procedures. Instead, the database server stores only a pathname to the compiled version of the procedure. You specify this pathname in the External Routine Reference clause.

The database server does store information about an external procedure in several system catalog tables, including **sysprocbody** and **sysprocauth**. For more information on these system catalog tables, see Chapter 1, "System Catalog," in the *Informix Guide to SQL: Reference*.

With external procedures, the END PROCEDURE keywords are optional.

Example

The following example registers an external C procedure named **check_owner()** in the database. This procedure 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 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;
```

♦

Using the DBA Keyword

The level of privilege necessary to execute a routine depends on whether the routine is created with the DBA keyword. The DBA keyword limits execution of the procedure to those users who have the DBA privilege.

You need the DBA privilege to create a procedure using the DBA keyword. You need the DBA privilege to execute a procedure that is created with the DBA keyword.

If you do not use the DBA option, the procedure is known as an owner-privileged procedure. If the procedure is owner privileged, and if the database is ANSI compliant, anyone can execute the procedure.

If you create an owner-privileged routine in a database that is not ANSI-compliant, the **NODEFDAC** environment variable prevents privileges on that routine from being granted to PUBLIC. See the *Informix Guide to SQL: Reference* for further information on the **NODEFDAC** environment variable.

Procedure Name

Because Universal Server offers *routine overloading*, you can define more than one procedure with the same name but different parameter lists. You may want to overload procedures if you are defining a type hierarchy or a system of distinct types or casts. When you overload procedures, you can create a procedure for the new data types you define.

The process of overloading routines and the routine resolution rules are described briefly in "Routine Resolution" on page 1-186.

The syntax of the Procedure Name segment is described in "Procedure Name" on page 1-1004.

Parameter List

SPL

To define the parameters for an SPL procedure, specify a parameter name and a data type for each parameter. For more information about defining parameters, see "Routine Parameter List" on page 1-1028. ♦

EXT

To define the parameters for an external routine, you can specify a name, and you must specify a data type for each parameter. For more information on the syntax of the parameter list, see "Routine Parameter List" on page 1-1028. ♦

Return Clause

The database server considers any routine that is created with a Return clause to be a function. Informix recommends that you use the CREATE FUNCTION statement, not CREATE PROCEDURE, to create functions. For external routines, this rule is strictly enforced.

The syntax of the Return clause is described in "Return Clause" on page 1-1020.

SPL

In SPL, you can use CREATE PROCEDURE to write and register a routine that returns one or more values (that is, a function). However, this feature is offered only for backward compatibility with earlier Informix products. Informix recommends that you do not use CREATE PROCEDURE to create functions. ♦

EXT

You cannot specify a Return clause for an external procedure. An external procedure does not return a value. ♦

Specific Name

You can specify a specific name for an SPL procedure or an external procedure. A specific name is a name that is unique in the database. A specific name is useful, because due to routine overloading, more than one procedure can have the same name.

The syntax of the Specific Name is described in "Specific Name" on page 1-1034.

Procedure Modifier

SPL

When you write an SPL procedure, you cannot specify a procedure modifier in the CREATE PROCEDURE statement. ♦

EXT

In the CREATE PROCEDURE statement, you can specify any of a list of procedure modifiers with a WITH clause. For more information on the procedure modifiers, see "Routine Modifier" on page 1-1022. ♦

Statement Block

SPL

In an SPL routine, you must specify an SPL statement block instead of an external routine reference. The syntax of the statement block is described in "Statement Block" on page 1-1037. ♦

External Routine Reference

EXT

When you register an external procedure, you must specify an External Routine Reference clause. The External Routine Reference clause specifies the pathname to the procedure object code, which is stored in a shared library. The External Routine Reference Clause also specifies the name of the language in which the procedure is written. For more information on the External Routine Reference clause, see "External Routine Reference" on page 1-956. ♦

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of the routine. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the routine.

To find the description of the SPL procedure **raise_prices**, shown in "SPL Procedures" on page 1-180, enter a query such as the following:

```
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
ORDER BY b.seqno;
```

The preceding query returns the following text:

```
USAGE: EXECUTE PROCEDURE raise_prices( xxx )
xxx = percentage from 1 - 100
```

An SPL routine, external routine, 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 PROCEDURE statement, whether or not you use END PROCEDURE. ♦

WITH LISTING IN Clause

The WITH LISTING IN option specifies a filename where compile-time warnings are sent. This listing file is created on the database server when you compile an SPL or external routine.

If you specify a filename but not a directory in the WITH LISTING IN clause, Universal Server uses the home directory on the database server as the default directory. If you do not have a home directory on the server, the file is created in the root directory.

If you do not use the WITH LISTING IN option, the compiler does not generate a list of warnings.

Privileges Necessary for Using CREATE PROCEDURE

You must have the Resource privilege on a database to create a procedure within that database. The owner of a procedure grants the Execution privilege to on that procedure to other users.

Routine Resolution

In Universal Server, you can have more than one instance of a routine with the same name but different parameter lists, as in the following situations:

- You create a routine 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 functions from their source types.

Routine resolution is the process of determining which instance of a function to execute, given the name of a routine and a list of arguments. For more information on routine resolution, refer to the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

PREPARE Statement

E/C

You can use a CREATE PROCEDURE statement only within a PREPARE statement. If you want to create a procedure for which the text is known at compile time, you must put the text in a file and specify this file with the CREATE PROCEDURE FROM statement. For more information, see the CREATE PROCEDURE FROM statement on page 1-188. ♦

References

See the CREATE FUNCTION, CREATE PROCEDURE FROM, DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, EXECUTE FUNCTION, EXECUTE PROCEDURE, GRANT, PREPARE, UPDATE STATISTICS, and REVOKE statements in this manual.

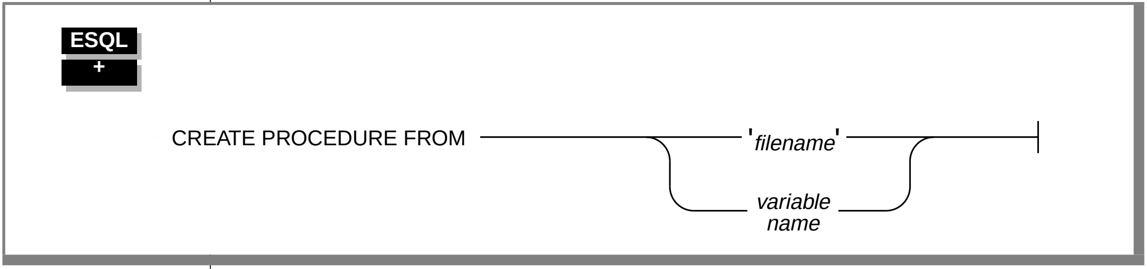
In the *Informix Guide to SQL: Tutorial*, see the discussion of how to create and execute SPL routines on page 14-6.

In the *Extending INFORMIX-Universal Server: User-Defined Routines* manual, see the discussion of how to create and use external procedures

CREATE PROCEDURE FROM

Use the CREATE PROCEDURE FROM statement to create a procedure. The actual text of the CREATE PROCEDURE statement resides in a separate file.

Syntax



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of the file that contains the full text of a CREATE PROCEDURE statement. The default pathname is the current directory.	The specified file must exist.	The pathname and filename must conform to the conventions of your operating system.
variable name	The name of a program variable that holds the value of *filename*	The file that is specified in the program variable must exist.	The name must conform to language-specific rules for variable names.

Usage

An INFORMIX-ESQL/C program cannot directly create a stored procedure or external procedure. That is, it cannot contain the CREATE PROCEDURE statement. However, you can create these functions within an ESQL/C program with the following steps:

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.

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 create the **raise_prices()** stored procedure with the following CREATE PROCEDURE FROM statement:

```
EXEC SQL create procedure from 'raise_pr.sql';
```

The filename that you provide is relative; if you provide a simple filename (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 PROCEDURE FROM actually contains a CREATE PROCEDURE statement. However, to improve readability of the code, Informix recommends that you match these two statements. If you are not sure whether the routine is a function or a procedure, use the CREATE ROUTINE FROM statement in the ESQL/C program.*

References

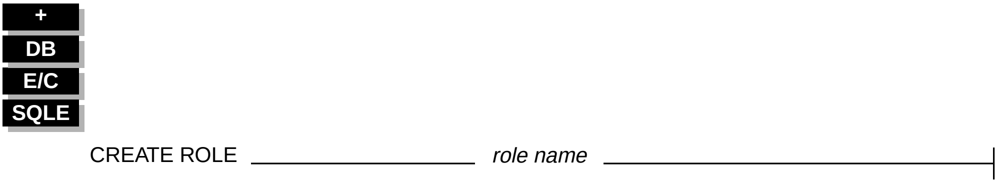
See the CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE, and CREATE ROUTINE FROM statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of how to create and use SPL procedures in Chapter 14.

CREATE ROLE

Use the CREATE ROLE statement to create a new role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name assigned to a role created by the DBA	Maximum number of characters is 8. A role name cannot be a user name known to the database server or the operating system of the database server. A role name cannot be in the **username** column of the **sysusers** system catalog table or in the **grantor** or **grantee** columns of the **systabauth**, **syscolauth**, **sysprocauth**, **sysfragauth**, and **sysroleauth** system catalog tables.	Identifier, p. 1-962

Usage

The database administrator (DBA) uses 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.

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 it is dropped either by the DBA or by a user to whom the role was granted with the WITH GRANT OPTION. Use the DROP ROLE statement to drop a role.

To create the role **engineer**, enter the following statement:

```
CREATE ROLE engineer
```

References

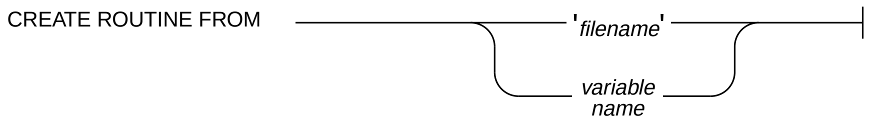
See the DROP ROLE, GRANT, REVOKE, and SET ROLE statements in this manual.

CREATE ROUTINE FROM

Use the CREATE ROUTINE FROM statement to create a routine. The actual text of the CREATE FUNCTION or CREATE PROCEDURE statement resides in a separate file.

Syntax

ESQL
+



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of the file that contains the full text of a CREATE PROCEDURE or CREATE FUNCTION statement. The default pathname is the current directory.	The specified file must exist.	The pathname and filename must conform to the conventions of your operating system.
variable name	The name of a program variable that holds the value of *filename*	The file that is specified in the program variable must exist.	The name must conform to language-specific rules for variable names.

Usage

An INFORMIX-ESQL/C program cannot directly define a routine. That is, it cannot contain the CREATE FUNCTION or CREATE PROCEDURE statement. However, you can create these functions within an ESQL/C program with the following steps:

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 filename that you provide is relative. If you provide a simple filename (as in the preceding example), the client application looks for the file in the current directory.

If you know at compile time whether the routine in the file is a function or a procedure, use the CREATE ROUTINE FROM statement in the ESQL/C program. However, if you do know whether the routine is a function or procedure, Informix recommends that you use the matching statement to create the file:

- The CREATE FUNCTION FROM to create stored or external functions
- The CREATE PROCEDURE FROM to create stored or external procedures

Use of the matching statements improves the readability of the code.

References

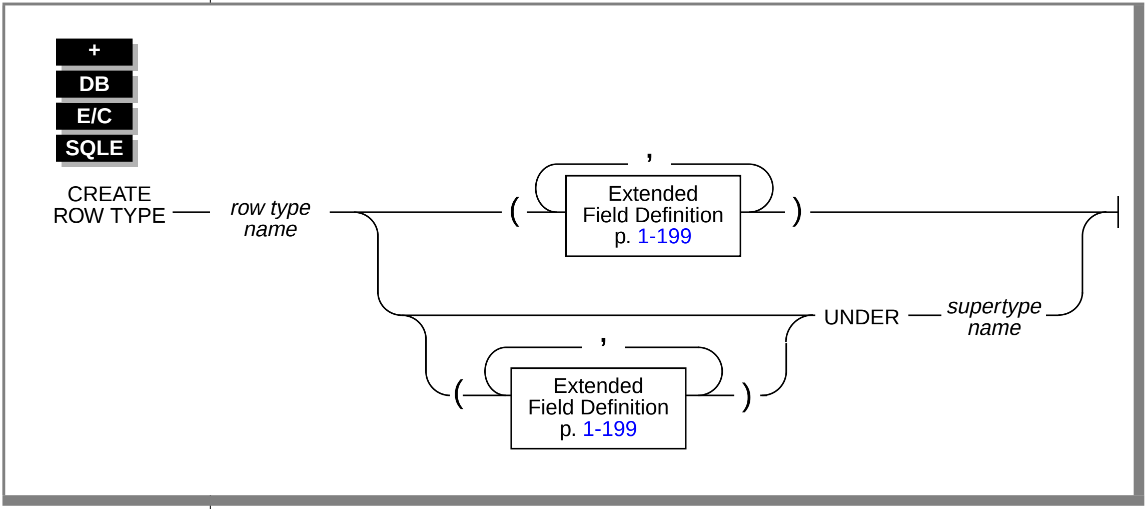
See the CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE, and CREATE PROCEDURE FROM statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures in Chapter 14.

CREATE ROW TYPE

Use the CREATE ROW TYPE statement to create a named row type.

Syntax


+
DB
E/C
SQLE
CREATE ROW TYPE
row type name
(
,
Extended Field Definition p. 1-199
)
UNDER
supertype name
(
,
Extended Field Definition p. 1-199
)

Element	Purpose	Restrictions	Syntax
row type name	The name of the named row type that you create. If you create a named row type under an existing supertype, this is the name of the subtype.	The name you specify for the named row type must follow the conventions for SQL identifiers. In an ANSI-compliant database, the combination *owner. type* must be unique within the database. In a database that is not ANSI compliant, the type name must be unique within the database. You must have the Resource privilege to create a named row type.	Identifier, p. 1-962 Data type, p. 1-855
supertype name	The name of the supertype in an inheritance hierarchy	The supertype must already exist and must be a named row type. The same restrictions apply for the supertype name as for the type name. In addition, you must have the Under privilege on this supertype to create a subtype under it, and the Resource privilege.	Identifier, p. 1-962 Data type, p. 1-855

Usage

The CREATE ROW TYPE statement creates a named row type. You can assign a named row type to a table or view to create a typed table or typed view. You can also assign a named row type to a column. Although you can assign a row type to a table to define the structure 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, which are defined using the Extended Field Definition syntax. For a full discussion of named row types and typed tables, see Chapter 10, “Understanding Complex Data Types,” in the *Informix Guide to SQL: Tutorial*.

You can use a named row type anywhere you can use any other type. Named row types are strongly typed. Any two named row types are not considered 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 on named row types and unnamed row types, see the section "Complex Data Type" on page 1-868 of this manual and Chapter 10, "Understanding Complex Data Types" in the *Informix Guide to SQL: Tutorial*.

Privileges on Named Row Types

The following table indicates which privileges you must have to create a row type.

Task	Privileges Required
Create a named row type	The Resource privilege on the database
Create a named row type as a subtype under a supertype	The Under privilege on the supertype, as well as the Resource privilege

To find out what privileges you have on a particular data type, check the **sysxtdtypes** system catalog table. This table is described in Chapter 1 of the *Informix Guide to SQL: Reference*.

See the reference pages for GRANT, beginning on page 1-458, for information about the RESOURCE, UNDER, and ALL privileges.

Privileges on a typed table (a table that is assigned a named row type) are the same as privileges on any table. Refer to the CREATE TABLE statement on page 1-208 and the "Table-Level Privileges" section of the GRANT statement on page 1-458.

To find out what privileges you have on a particular table, check the **systabauth** system catalog table. This table is described in Chapter 1 of the *Informix Guide to SQL: Reference*.

Privileges on Named Row Type Columns

Privileges on named row type columns are the same as privileges on any column. For more information, see the "Table-Level Privileges" section of the GRANT statement on page 1-458.

To find out what privileges you have on a particular column, check the **syscolauth** system catalog table. This table is described in Chapter 1 of the *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. You 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.

When you create a named row type as a subtype, the subtype inherits the following properties:

- All fields of the supertype
- All functions that are defined on the supertype

In addition, you can add new fields to the subtype that you create and define functions on the subtype. The new fields and functions are specific to the subtype alone.

You cannot substitute a row type in an inheritance hierarchy for its supertype or its subtype. For example, suppose you define 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 a another named row type (supertype). To create the fields of a named row type, you use the field definition clause that is shown on page 1-200.

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**. However, the **person_t** type is not altered.

Tip: *A subtype inherits all the fields and functions that are defined on the supertype as well as any additional fields and routines that you define on the subtype.*

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 and functions of **person_t**. If you create another named row type, **salesrep_t** under **customer_t**, **salesrep_t** inherits all the fields and functions of **customer_t**. More specifically, **salesrep_t** inherits all the fields and functions that **customer_t** inherited from **person_t** as well as all the fields and functions defined specifically for **customer_t**. For a full discussion of type inheritance, refer to Chapter 10 of the *Informix Guide to SQL: Tutorial*.

Procedure for Creating a Subtype

Before you create a named row type as a subtype in an inheritance hierarchy, do the following:

- 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 schema.

  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 you want to use 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.

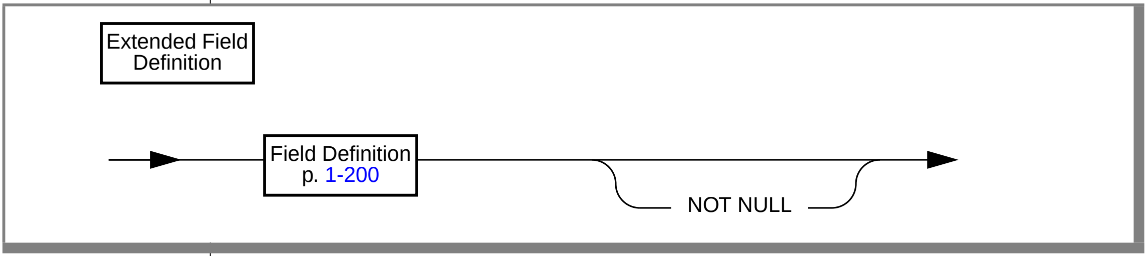
Important: *When you create a subtype, do not redefine fields that the subtype inherited for its supertype. If you attempt to redefine these fields, the database server returns an error.*

Constraints on Named Row Types

You cannot apply constraints to named row types directly. Specify the constraints for the tables that use named row types when you create or alter the table.

Extended Field Definition

Use the *extended field definition* to define new fields in a named row type.

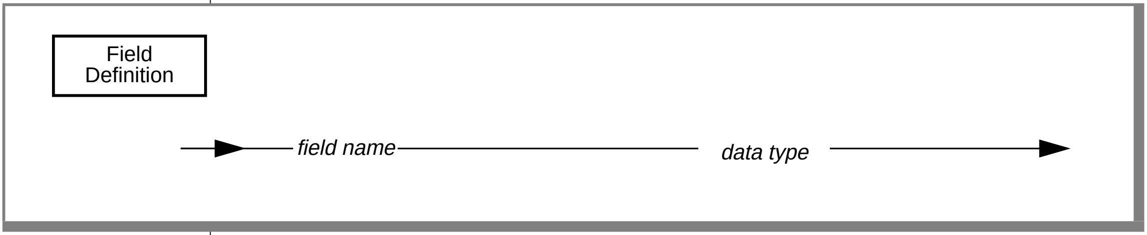


Each field has its own field definition, as described in the "Field Definition" section.

Important: *The* NOT NULL *constraints that you specify on the fields of a named row type also apply to corresponding columns of a table when the named row type is used to create a typed table.*

Field Definition

To define a field, you must specify a name and a data type for each field.



Element	Purpose	Restrictions	Syntax
field name	Name of a field in the row	Name must be unique within the row type and its supertype.	Identifier, p. 1-962
data type	Data type of the field	If a named row type is used to define a column, the fields of the row type cannot be the SERIAL, SERIAL8, BYTE, or TEXT data type. If a named row type is assigned to a table, the fields of the row type cannot be the SERIAL or SERIAL8 data type.	Data type, p. 1-855

References

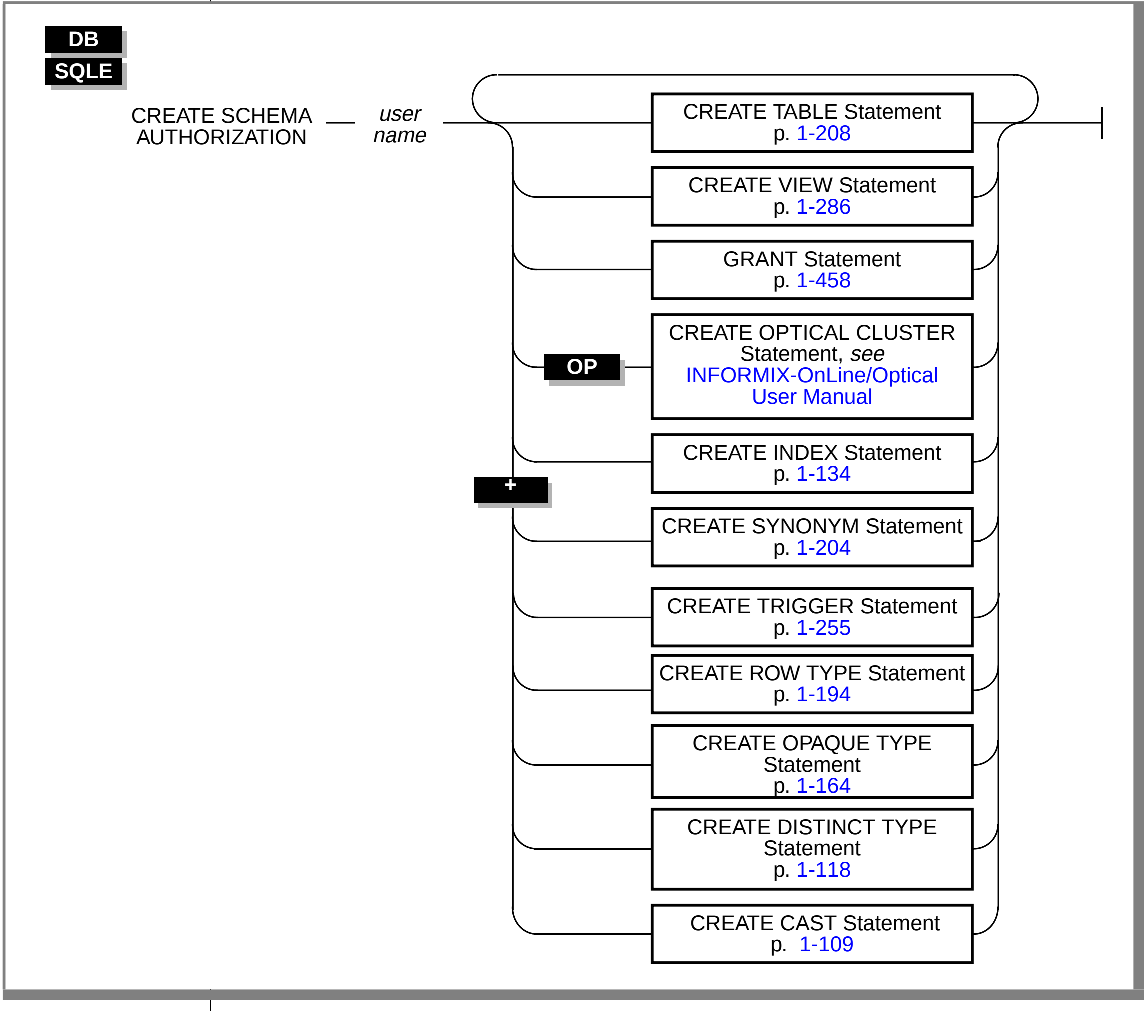
See the DROP ROW TYPE, CREATE TABLE, CREATE CAST, GRANT, and REVOKE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of named row types in Chapter 10, “Understanding Complex Data Types.” In the *Informix Guide to SQL: Reference*, see Chapter 2, “Data Types.”

CREATE SCHEMA

Use the CREATE SCHEMA statement to issue a block of CREATE and GRANT statements as a unit. This statement allows you to specify an owner of your choice for all objects that the CREATE SCHEMA statement creates.

Syntax


DB
SQLE
CREATE SCHEMA AUTHORIZATION
user name
CREATE TABLE Statement p. 1-208
CREATE VIEW Statement p. 1-286
GRANT Statement p. 1-458
OP
CREATE OPTICAL CLUSTER Statement, see INFORMIX-OnLine/Optical User Manual
CREATE INDEX Statement p. 1-134
+
CREATE SYNONYM Statement p. 1-204
CREATE TRIGGER Statement p. 1-255
CREATE ROW TYPE Statement p. 1-194
CREATE OPAQUE TYPE Statement p. 1-164
CREATE DISTINCT TYPE Statement p. 1-118
CREATE CAST Statement p. 1-109

Element	Purpose	Restrictions	Syntax
user name	The name of the user who will own the objects that the CREATE SCHEMA statement creates	If the user who issues the CREATE SCHEMA statement has the Resource privilege, *user name* must be the name of this user. If the user who issues the CREATE SCHEMA statement has the DBA privilege, *user name* can be the name of this user or another user.	Identifier, p. 1-962

Usage

You cannot issue the CREATE SCHEMA statement until you create the affected database.

Users with the Resource privilege can create a schema for themselves. In this case, *user name* 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 name* 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 within the statement, 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 > '12/31/1993'
    CREATE INDEX idxtime ON mytable (mytime);
```

Creating Objects Within CREATE SCHEMA

All objects that a CREATE SCHEMA statement creates are owned by *user name*, even if you do not explicitly name each object. If you are the DBA, you can create objects for another user. If you are not the DBA, and you try to create an object for an owner other than yourself, you receive an error message.

Granting Privileges Within CREATE SCHEMA

You can only grant privileges with the CREATE SCHEMA statement; you cannot revoke or drop privileges.

Creating Objects or Granting Privileges Outside CREATE SCHEMA

If you create an object or use the GRANT statement outside a CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**.

References

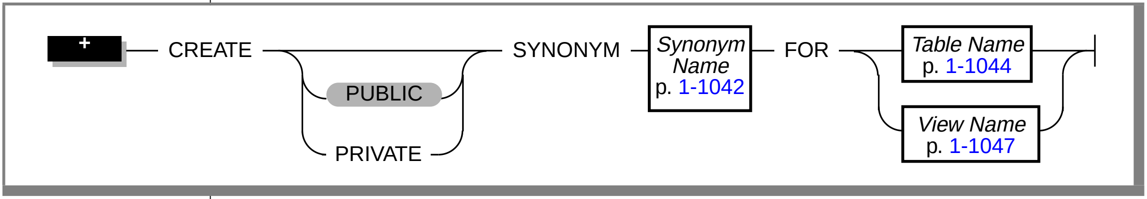
See the CREATE INDEX, CREATE SYNONYM, CREATE TABLE, CREATE VIEW, and GRANT statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating the database in Chapter 9.

CREATE SYNONYM

Use the CREATE SYNONYM statement to provide an alternative name, called a synonym, for a table or view.

Syntax



Usage

Users have the same privileges for a synonym that they have for the table to which the synonym applies.

The synonym name must be unique; that is, the synonym name cannot be the same as another database object, such as a table, view, or temporary table.

Once a synonym is created, it persists until the owner executes the DROP SYNONYM statement. This property distinguishes a synonym from an alias that you can use in the FROM clause of a SELECT statement. The alias persists for the existence of the SELECT statement. If a synonym refers to a table or view in the same database, the synonym is automatically dropped if you drop the referenced table or view.

You cannot create a synonym for a synonym in the same database.

ANSI

The owner of the synonym (*owner.synonym*) qualifies the name of a synonym. The identifier *owner.synonym* must be unique among all the synonyms, tables, temporary tables, and views in the database. You must specify *owner* when you refer to a synonym that another user owns. The following example shows this convention:

```
CREATE SYNONYM emp FOR accting.employee
```

♦

You can create a synonym for any table or view in any database on your database server. Use the *owner*. convention if the table is part of an ANSI-compliant database. The following example shows a synonym for a table outside the current database. It assumes that you are working on the same database server that contains the **payables** database.

```
CREATE SYNONYM mysum FOR payables:jean.summary
```

You can create a synonym for any table or view that exists on any networked database server as well as on the database server that contains your current database. The database server that holds the table must be on-line when you create the synonym. In a network, INFORMIX-Universal Server verifies that the object of the synonym exists when you create the synonym.

The following example shows how to create a synonym for an object that is not in the current database:

```
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. Note that 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`.

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 a synonym is public, a user does not need to know the name of the owner of the synonym. Any synonym in a database that is not ANSI compliant *and* was created before Version 5.0 of the database server is a public synonym.

ANSI

Synonyms are always private. If you use the PUBLIC or PRIVATE keywords, you receive a syntax error. ♦

If you use the PRIVATE keyword, the synonym can be used only by the owner of the synonym or if the owner's name is specified explicitly with the synonym. More than one private synonym with the same name can exist in the same database. However, a different user must own each synonym with that name.

You can own only one synonym with a given name; you cannot create both private and public synonyms with 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!!!
```

Synonyms with the Same Name

If you own a private synonym, and a public synonym exists with the same name, when you use the synonym by its unqualified name, the private synonym is used.

If you use DROP SYNONYM with a synonym, and multiple synonyms exist with the same name, the private synonym is dropped. If you issue the DROP SYNONYM statement again, the public synonym is dropped.

Chaining Synonyms

If you create a synonym for a table that is not in the current database, and this table is dropped, the synonym stays in place. You can create a new synonym for the dropped table, with the name of the dropped table as the synonym name, which points to another external or remote table. In this way, you can move a table to a new location and chain synonyms together so that the original synonyms remain valid. (You can chain as many as 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 **stores7** 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 stores7@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 stores7@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 stores7@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.

References

See the DROP SYNONYM statement in this manual.

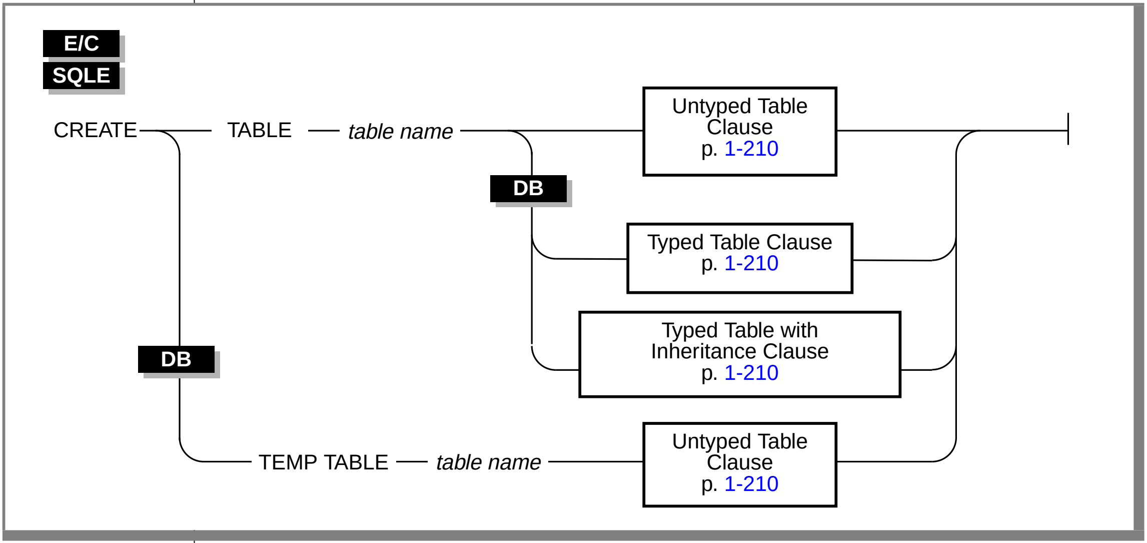
In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 5.

CREATE TABLE

Use the CREATE TABLE statement to create a new table in the current database, place data-integrity constraints on its columns or on a combination of its columns, designate the size of its initial and subsequent extents, and specify how to lock each table. You can also use this statement to fragment tables into separate dbspaces.

You can use CREATE TABLE to create *untyped table*s (traditional relational-database tables), *typed tables* (object-relational tables), typed tables with inheritance, or temporary tables.

Syntax



E/C
SQLE
CREATE
TABLE
table name
Untyped Table Clause p. 1-210
DB
Typed Table Clause p. 1-210
Typed Table with Inheritance Clause p. 1-210
DB
TEMP TABLE
table name
Untyped Table Clause p. 1-210

Element	Purpose	Restrictions	Syntax
table name	The name assigned to the table. Every table must have a name.	Name must be unique within a database. It must not be used for any other tables or for any views or synonyms within the current database.	Identifier, p. 1-962

Some of the syntax diagrams in this chapter include branches that are valid only for certain types of tables. The diagrams use the following icons to indicate which types of tables can use the limited branches:

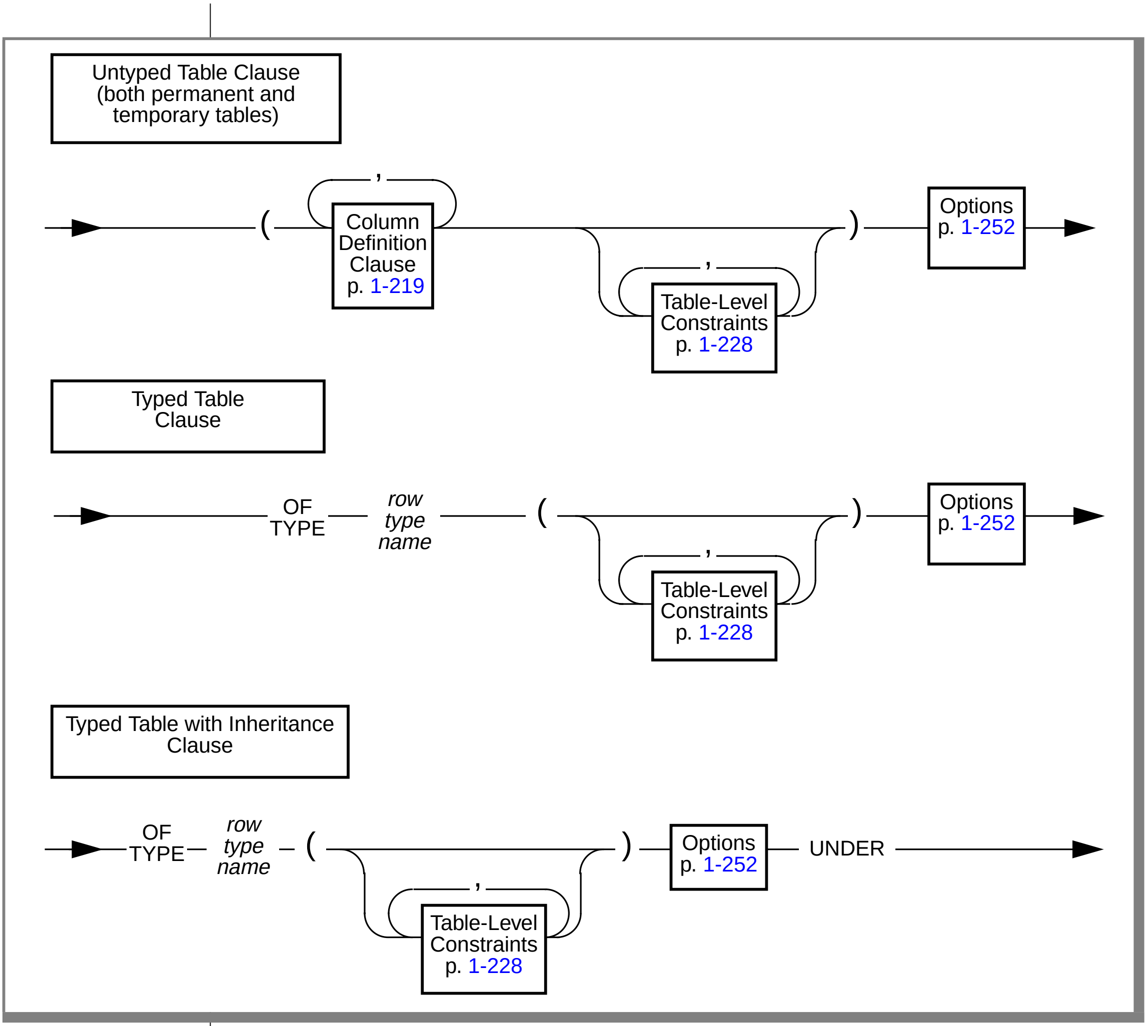
UnT Actions in this branch can appear in (permanent) untyped tables.

Typ Actions in this branch can appear in typed tables.

Tmp Actions in this branch can appear in temporary (untyped) tables.

Syntax Clauses for Typed and Untyped Tables

The following syntax diagrams show the syntax for both typed and untyped tables. Typed tables and inheritance are new features introduced by INFORMIX-Universal Server. Earlier releases of Informix products support only untyped tables, both permanent and temporary.


Untyped Table Clause
(both permanent and
temporary tables)
(
,
Column
Definition
Clause
p. 1-219
,
Table-Level
Constraints
p. 1-228
)
Options
p. 1-252
Typed Table
Clause
OF
TYPE
row
type
name
(
,
Table-Level
Constraints
p. 1-228
)
Options
p. 1-252
Typed Table with Inheritance
Clause
OF
TYPE
row
type
name
(
,
Table-Level
Constraints
p. 1-228
)
Options
p. 1-252
UNDER

Element	Purpose	Restrictions	Syntax
row type name	The name of row type used as this table's type	This type must already exist and must be a named row type.	Data type, p.1-855 Identifier, p. 1-962 *Informix Guide to SQL: Reference*: Chapter 3, "Environment Variables"
supertable name	The name of the parent table of which this table is a child.	This parent table must already exist as a typed table. A type hierarchy must already exist in which the named row type of this table is a subtype of the named row type of the supertable.	

Usage

When you create a table, the table and columns within that table must have unique names and every table column must have a data type associated with it. However, although temporary table names must be different from existing table, view, or synonym names in the current database, they need not be different from temporary table names used by other users.

ANSI

In an ANSI-compliant database, the combination *owner.tablename* must be unique within the database. ♦

DB

In DB-Access, using the CREATE TABLE statement outside the CREATE SCHEMA statement generates warnings if you set **DBANSIWARN**. ♦

ESQL

The CREATE TABLE statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN** environment variable. ♦

For information about **DBANSIWARN** environment variable, refer to the *Informix Guide to SQL: Reference*.

Typed and Untyped Tables

Untyped tables are the only kinds of tables that are available in Informix products prior to INFORMIX-Universal Server. Typed tables use *named row types*. For a detailed discussion of row types and typed and untyped tables, refer to Chapter 10 of the *Informix Guide to SQL: Tutorial*.

Important: *Informix recommends that you use the BLOB or CLOB data types instead of the TEXT or BYTE 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 TEXT or BYTE fields and use that type to recreate an existing (untyped) table as a typed table. However, 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 CLOB or BLOB fields in both typed tables and columns.*

Typed Tables

A *typed table* is a table that has a named row type assigned to it. The columns of a typed table correspond to the fields of the named row type.

For example, suppose you create a named row type, **student_t** as follows:

```
CREATE ROW TYPE student_t
    (name       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**.

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
```

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. You cannot add additional columns to a typed table.

Important: *Typed tables do not take the default values or null/not null specification of the row type whose type they adopt.*

For more information about row types, refer to the CREATE ROW TYPE statement on page 1-194.

Typed Tables with Inheritance

A typed table can inherit properties from a typed *supertable* and add new columns and properties. The table that inherits is a *subtable*. The subtable must use a row type that is derived from the row type of the supertable.

Continuing the example shown in "Typed Tables" on page 1-212, the following statements created a typed table, **grad_students**, that inherits all of the columns of the **students** table and in addition has columns for **adviser** and **field_of_study**:

```
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 create a typed table as a subtable, the subtable inherits the following properties:

- All columns in the immediate supertable
- All constraint definitions defined on its supertable
- Fragmentation. If a subtable does not define fragments, and if its supertable has fragments defined, then the subtable inherits the fragments of the supertable.
- All indexes defined by its supertable
- Referential integrity

- The access method
- The WITH options
- The storage option
- 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. It is not necessary to add all heritable attributes to a supertable before creating its subtables.*

Inheritance occurs in one direction only—from supertable to subtable. Properties of subtables are *not* inherited by supertables.

Constraints, indices, and triggers are recorded in the system catalog for the supertable, but not for subtables that inherit them. Fragmentation information is recorded for both supertables and subtables.

No two tables in a table hierarchy can have the same type. For example, the final line of the following code sample is illegal 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;
Illegal --> create table tab3 of type rowtype2 under tab1;
```

For more information about inheritance, refer to Chapter 10 of the *Informix Guide to SQL: Tutorial*.

Untyped Tables

Tables that have not been assigned a named row type are *untyped tables*. Untyped tables, both permanent and temporary, are traditional relational-database tables. For simplicity, this discussion refers to permanent untyped tables as *untyped tables* and temporary untyped tables as *temporary tables*.

The following CREATE TABLE statement creates an untyped table:

```
CREATE TABLE students
    (name       VARCHAR(30),
     average    REAL,
     birthdate  DATETIME YEAR TO DAY)
```

Temporary Tables

Temporary tables are always untyped tables. The following CREATE TABLE statement creates a temporary table:

```
CREATE TEMP TABLE transient
    (col1   integer,
     col2   char(20))
```

After a temporary table is created, you can build indexes on the table. However, you are the only user who can see the temporary table.

Temporary tables that you create with the CREATE TEMP TABLE statement are *explicit* temporary tables. You can also create explicit temporary tables with the SELECT ... INTO TEMP statement. Temporary tables that the database server creates as a part of processing are called *implicit* temporary tables. Implicit temporary tables are discussed in the *INFORMIX-Universal Server Administrator's Guide*.

When an application creates an explicit temporary table, the table exists until one of the following situations occur:

- The application terminates.
- The application closes the database where the table was created. In this case, the table is dropped only if the database does transaction logging, and the temporary table was not created with the WITH NO LOG option.
- The application closes the database where the table was created and opens a database in a different database server.

When any of these events occur, the temporary table is deleted.

DB

You cannot use the INFO statement and the Info Menu Option with temporary tables. ♦

Temporary table names must be different from existing table, view, or synonym names in the current database. However, they need not be different from other temporary table names used by other users.

You can specify where temporary tables are created with the CREATE TEMP TABLE statement, environment variables, and ONCONFIG parameters. If you do not specify a storage location, the temporary tables are created in the same dbspace as the database. The database server stores temporary tables in the following order:

1. The IN *dbspace* clause

   You can specify the dbspace where you want the temporary table stored with the IN *dbspace* clause of the CREATE TABLE statement.

2. The dbspaces you specify when you fragment temporary tables

   Use the FRAGMENT BY clause of the CREATE TABLE statement to fragment regular and temporary tables.

3. The **DBSPACETEMP** environment variable

   The **DBSPACETEMP** environment variable lists dbspaces where temporary tables can be stored. This list can include standard dbspaces, temporary dbspaces, or both. If the environment variable is set, the database server assigns each temporary table to a dbspace in round-robin sequence.

4. The ONCONFIG parameter DBSPACETEMP

   You can specify a location for temporary tables with the ONCONFIG parameter DBSPACETEMP.

Tip: *Use the PUT clause to specify a separate storage area for smart large objects.*

For additional information about the **DBSPACETEMP** environment variable, see Chapter 3 in the *Informix Guide to SQL: Reference*. For additional information about the ONCONFIG parameter DBSPACETEMP, see the *INFORMIX-Universal Server Administrator's Guide*.

Differences Among Tables

Tables created with the CREATE TABLE statement are similar in most ways, but have a few notable differences. The following table summarizes the major differences among tables.

Table	Description	Characteristics
Untyped table	A permanent database table See "Untyped Tables" on page 1-214.	Allows column-level constraints as well as table-level constraints.
Temporary Table	A table that exists only until the application either terminates or, under certain conditions, closes the database. See "Temporary Tables" on page 1-215.	Allows column-level constraints as well as table-level constraints. Can use the **DBSPACETEMP** environment variable or the DBSPACETEMP configuration parameter to specify storage location.
Typed table	A permanent database table See "Typed Tables" on page 1-212.	Allows only table-level constraints. Does not allow SERIAL, and SERIAL8 data types. Does not allow the WITH ROWIDS clause.
Table with Inheritance	A permanent database table that is a subtable in an inheritance hierarchy. See "Typed Tables with Inheritance" on page 1-213.	Both subtable and supertable must be typed, and their types must be named row types. The type of the subtable must be a subtype directly under the type of the supertable. Allows only table-level constraints. Does not allow SERIAL, and SERIAL8 data types. Does not allow the WITH ROWIDS clause.

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 information about the privileges required for creating a table, refer to the GRANT statement on page 1-458. For additional information about privileges, refer to Chapter 11, "Granting and Limiting Access to Your Database," in the *Informix Guide to SQL: Tutorial*.

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 on a new table in a database that is not ANSI compliant from being granted to PUBLIC. For information about preventing privileges from being granted to PUBLIC, see the **NODEFDAC** environment variable in the *Informix Guide to SQL: Reference*.

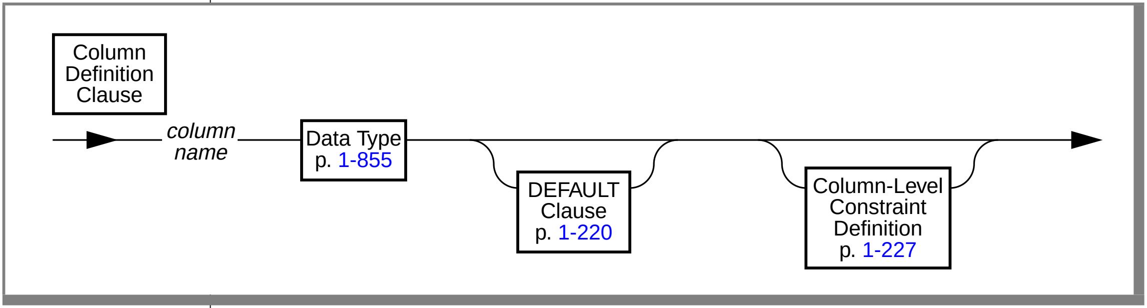
System Catalog Information

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** table. The **sysblobs** 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 required for various CREATE TABLE options. The **systables**, **sysxtdtypes**, and **sysinherits** system catalog tables provide information about table types.

For information about the system catalog tables, refer to the *Informix Guide to SQL: Reference*. For information about sysmaster database, refer to the *INFORMIX-Universal Server Administrator's Guide*.

Column Definition Clause

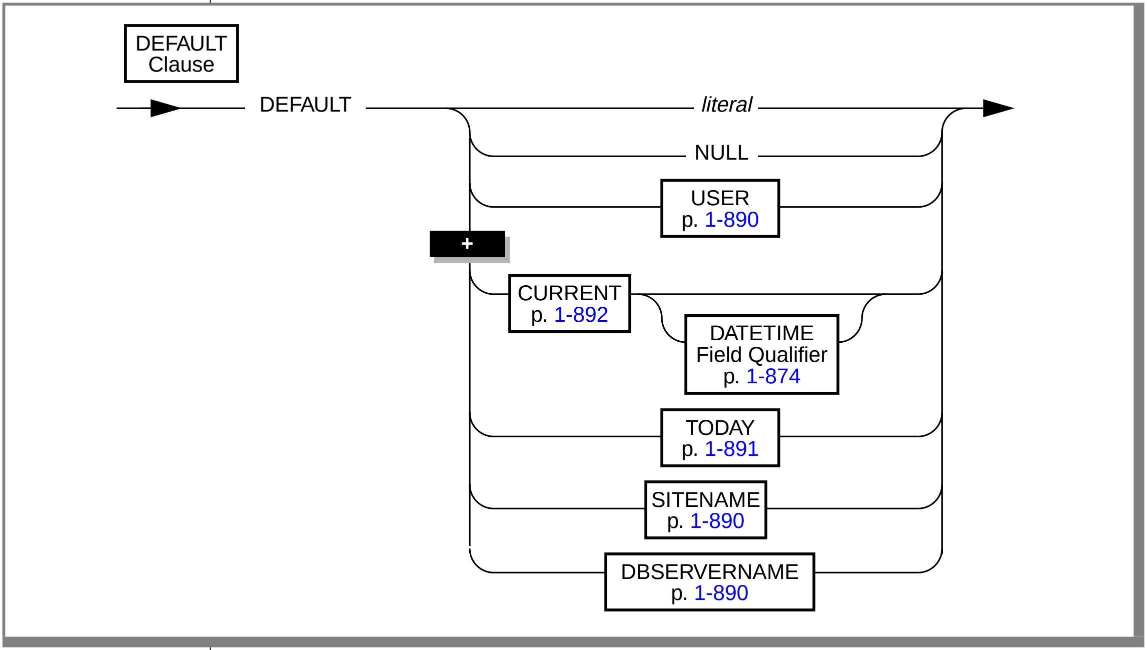


Element	Purpose	Restrictions	Syntax
column name	The name of a column in the permanent table	Name must be unique within a table, but you can use the same names in different tables in the same database.	Identifier, p. 1-962

Use the column definition portion of the CREATE TABLE statement to list the name, data type, default values, and constraints *of a single column* of an untyped table (permanent or temporary) as well as to specify constraints on the column.

The Untyped Table clause on page 1-210 refers to the Column Definition Clause.

DEFAULT Clause



Element	Purpose	Restrictions	Syntax
literal	A literal term that defines alphabetic or numeric constant characters to be used as the default value for the column	Term must be appropriate type for the column. See "Literal Terms as Default Values" on page 1-221.	Expression, p. 1-876

The default value is inserted in the column when an explicit value is not specified. If a default is not specified, and the column allows nulls, the default is NULL.

The column definition clause on page 1-219 refers to DEFAULT clause.

Important: *If you use a named row type as one of the columns in an untyped table, the table does not adopt any constraints of the named row.*

Literal Terms as Default Values

You can designate *literal terms* as default values. A literal term is a string of character or numeric constant characters that you define. To use a literal term as a default value, you must adhere to the following rules.

Use a Literal	With Columns of Data Type
INTEGER	INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, SMALLFLOAT, INT8
DECIMAL	DECIMAL, MONEY, FLOAT, SMALLFLOAT
CHARACTER	CHAR, VARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, DATE
INTERVAL	INTERVAL
DATETIME	DATETIME
CHARACTER	Opaque data types

Characters must be enclosed in quotation marks. Date literals must be of the format specified with the **DBDATE** environment variable. If **DBDATE** is not set, the format *mm/dd/yyyy* is assumed.

Opaque data types support only string literals for default values. The default value must be specified at the column level and not at the table level.

For information on using a literal INTERVAL, refer to the Literal INTERVAL segment on page 1-994. For more information on using a literal DATETIME, refer to the Literal DATETIME segment on page 1-991.

NULL as the Default Value

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 this case, no default value exists for the column.

If you designate NULL as the default value for a column, you cannot specify a not null constraint as part of the column definition.

If the column is TEXT or BYTE data type, you can designate *only* NULL as the default value.

Data Type Requirements for Certain Columns

The following table indicates the data type requirements for columns that specify the CURRENT, DBSERVERNAME, SITENAME, TODAY, or USER functions as the default value.

Function Name	Data Type Requirement
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 18 characters long
SITENAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 18 characters long
TODAY	DATE column
USER	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column at least 8 characters long

Limitations on Default Values

You cannot designate default values for serial columns.

You cannot designate a server-defined function (that is, CURRENT, USER, TODAY, SITENAME or DBSERVERNAME) as the default value for opaque or distinct data types.

You cannot designate NULL as a default value for a column that is part of a primary key.

Examples of Default Values in Column Definitions

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 user's login name.

```
CREATE TABLE accounts (
    acc_num INTEGER DEFAULT 0001,
    acc_type CHAR(1) DEFAULT 'A',
    acc_descr CHAR(20) DEFAULT 'New Account',
    acc_id CHAR(8) DEFAULT USER)
```

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))
```

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.

Constraints

Putting a constraint on a column is similar to putting an index on a column (using the CREATE INDEX statement). However, if you use constraints instead of indexes, you can also implement data-integrity constraints and turn effective checking off and on. For information on data-integrity constraints, refer to the *Informix Guide to SQL: Tutorial*. For information on effective checking, see the SET statement on page 1-644.

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.*

Limits on Constraint Definitions

You can include 16 columns in a list of columns for a table-level constraint. The total length of all columns in the constraint list cannot exceed 390 bytes.

You cannot place a constraint on a violations or diagnostics table. For further information on violations and diagnostics tables, see the START VIOLATIONS TABLE statement on page 1-744.

Restrictions for Unique Constraints

Use the UNIQUE keyword to require that a single column or set of columns accepts only unique data. You cannot insert duplicate values in a column that has a unique constraint.

When you define a unique constraint (UNIQUE or DISTINCT keywords), a column cannot appear in the constraint list more than once. You cannot place a unique constraint on a column on which you have already placed a primary-key constraint. You cannot place a unique constraint on a BYTE or TEXT column.

Opaque types support a unique constraint only where there is a secondary access method that supports the uniqueness for that type. The built-in (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 unique constraint.

Restrictions for Primary-Key Constraints

You can define a primary-key constraint (PRIMARY KEY keywords) on only one column or one set of columns in a table.You cannot define a column or set of columns as a primary key if you have already defined another column or set of columns as the primary key.

You cannot define a primary-key constraint on a BYTE or TEXT column.

Opaque types support a primary key constraint only where there is a secondary access method that supports the uniqueness for that type. The built-in 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

Restrictions for Referential Constraints

When you specify a referential constraint, the data type of the referencing column (the column you specify after the FOREIGN KEY keywords) must match the data type of the referenced column (the column you specify in the REFERENCES clause). The only exception is that the referencing column must be INTEGER if the referenced column is SERIAL, or INT8 if the column is SERIAL8.

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

Adding or Dropping Constraints

After you have used the CREATE TABLE statement to place constraints on a column or set of columns in an *untyped or temporary* table, you can use the ALTER TABLE statement to modify the constraints. You cannot use ALTER TABLE with a typed table.

Enforcing Primary-Key, Unique, and Referential Constraints

When a primary-key, unique, and referential constraint is placed on a column, the database server performs the following functions:

- Creates a unique, ascending index for a unique or primary-key constraint
- Creates a nonunique, ascending index for the columns specified in the referential constraint

However, if a constraint already was created on the same column or set of columns, another index is not built for the constraint. Instead, the existing index is *shared* by the constraints. If the existing index is non-unique, it is *upgraded* to a unique index if a unique or primary-key constraint is placed on that column.

Because these constraints are enforced through indexes, you cannot create an index (using the CREATE INDEX statement) for a column that is of the same type as the constraint placed on that column. For example, if a unique constraint exists on a column, you can create neither an ascending unique index for that column nor a duplicate ascending index.

Constraint Names

Whenever you place a data restriction on a column or specify a table-level constraint, the database server creates a constraint. If you wish, you can specify a name for the constraint. The name of the constraint must be unique within the database.

The database server adds a row to the **sysconstraints** system catalog table for each constraint. 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. For example, the constraint name for a unique constraint might look like this: **u111_14**. If the name conflicts with an existing identifier, the database server returns an error, and you must then supply a constraint name.

ANSI

When you create a constraint of any type, the *owner.constraint_name* (the combination of the owner name and constraint name) must be unique within the database. ♦

In addition, the database server adds a row to the **sysindices** system catalog table for each new primary-key, unique, or referential constraint that does not share an index with an existing constraint. The index name in **sysindices** is created with the following format:

```
[space]<tabid>_<constraintid>
```

In this format, *tabid* and *constraintid* are values from the **tabid** and **constrid** columns of the **systables** and **sysconstraints** system catalog tables, respectively. For example, the index name might be something like this: " **121_13**" (quotes used to show the space).

Using Simple Large Object Types in Constraints

You cannot place a unique, primary-key, or referential constraint on BYTE or TEXT columns. However, you can check for null or non-null values if you place a check constraint on a BYTE or TEXT column.

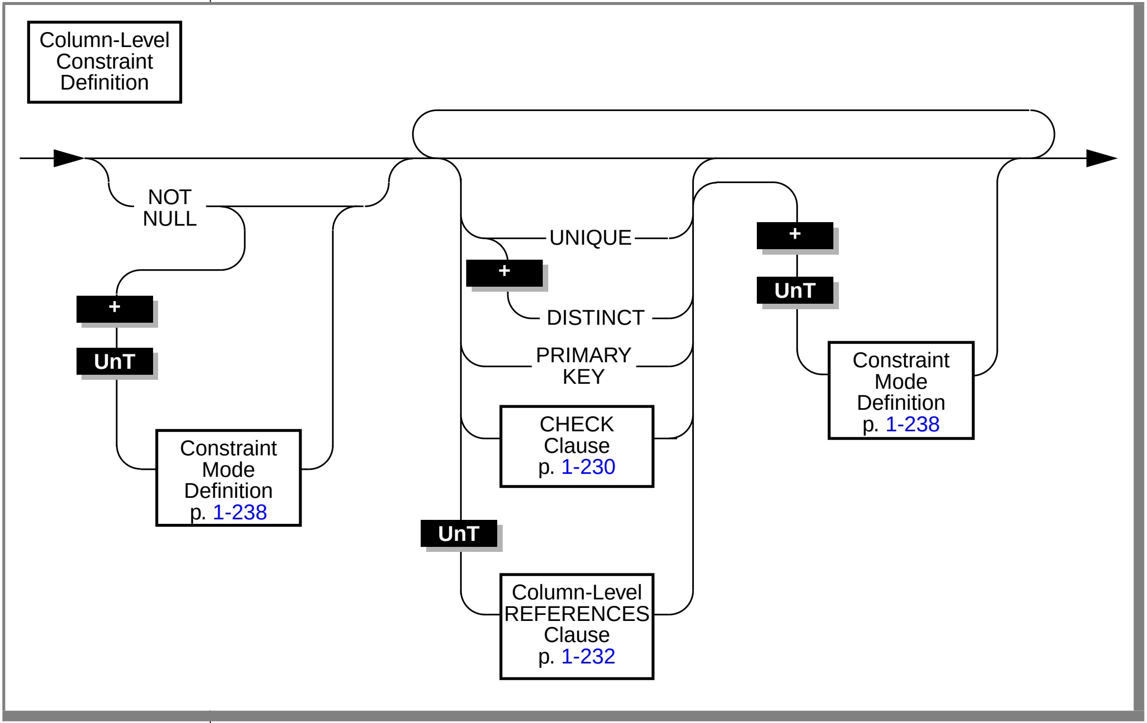
Restrictions on Temporary Table Constraints

The only difference between columns in permanent tables and columns in temporary tables is in the constraint options, as follows:

- You cannot place referential constraints on columns in a temporary table. Temporary columns cannot be referenced or referencing columns.
- You cannot assign a name to a constraint on a temporary-table column.
- You cannot set the constraint mode on a temporary-table column. (See "Constraint Mode Definition" on page 1-238 for information on this option.)

Column-Level Constraint Definition

In untyped and temporary tables, you can define constraints at either the *column level* or *table level*. At the column level, you can indicate that the column has a specific default value or that data entered into the column must be checked to meet a specific data requirement. Constraints at the column level cannot refer to multiple columns. In other words, the constraint created at the column level can apply only to a single column.


Column-Level Constraint Definition
NOT NULL
+
UnT
Constraint Mode Definition p. 1-238
UNIQUE
+
DISTINCT
PRIMARY KEY
CHECK Clause p. 1-230
UnT
Column-Level REFERENCES Clause p. 1-232
+
UnT
Constraint Mode Definition p. 1-238

The following example creates a simple table with two constraints, a primary-key constraint named **num** on the **acc_num** column and a unique constraint named **code** 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 column definition clause on page 1-219 refers to the column-level constraint definition.

Table-Level Constraint Definition

You can define table-level constraints for both typed and untyped tables. When you define a constraint at the table level, the constraint can refer to a single column or to multiple columns. Constraints that refer to a single column are treated the same way whether they are defined at the column level or the table level.

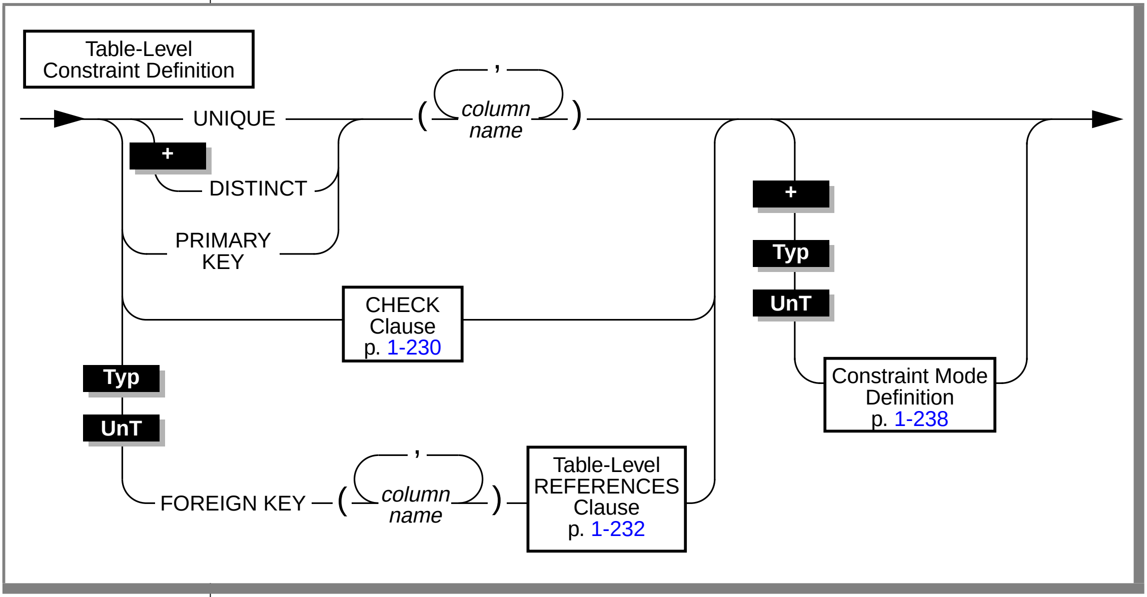


Table-Level Constraint Definition
UNIQUE
+
DISTINCT
PRIMARY KEY
(
,
column name
)
CHECK Clause p. 1-230
Typ
UnT
FOREIGN KEY
(
,
column name
)
Table-Level REFERENCES Clause p. 1-232
+
Typ
UnT
Constraint Mode Definition p. 1-238

Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns on which the constraint is placed	You must observe general restrictions that apply regardless of the type of constraint you are defining. You must also observe specific restrictions that depend on the type of constraint you are defining. See "Constraints" on page 1-223.	Identifier, p. 1-962

Using the UNIQUE and DISTINCT Keywords

Use the UNIQUE keyword to require that a single column or set of columns accepts only unique data. You cannot insert duplicate values in a column that has a unique constraint.

When you define a unique constraint (UNIQUE or DISTINCT keywords), a column cannot appear in the constraint list more than once. You cannot place a unique constraint on a column on which you have already placed a primary-key constraint. You cannot place a unique constraint on a BYTE or TEXT column.

Each column named in a unique constraint must be a column in the table and cannot appear in the constraint list more than once.

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)
```

The following example creates a simple table, but includes the constraint as a table-level constraint instead of a column-level constraint:

```
CREATE TABLE accounts
    (acc_name   CHAR(12),
     acc_num    SERIAL,
     UNIQUE     (acc_num) CONSTRAINT acc_num)
```

Using the PRIMARY KEY Keywords

A primary key is a column or set of columns that contains a non-null unique value for each row in a table. A table can have only one primary key, and a column that is defined as a primary key cannot also be defined as unique.

In the previous two examples, a unique constraint was placed on the column **acc_name**. The following example creates this column as the primary key for the **accounts** table:

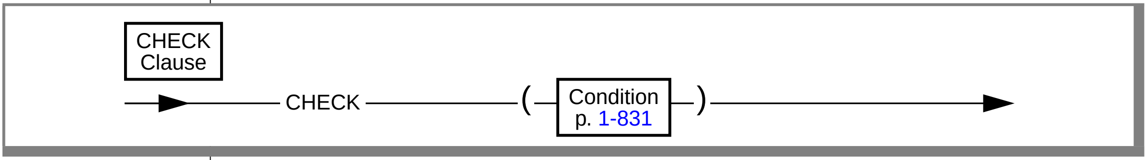
```
CREATE TABLE accounts
    (acc_name   CHAR(12),
     acc_num    SERIAL,
     PRIMARY KEY (acc_num))
```

Using the FOREIGN KEY Keywords

A foreign key *joins* and establishes dependencies between tables. 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 make BYTE or TEXT columns foreign keys.

When you use FOREIGN KEY keywords, you must use the REFERENCES clause, page 1-232, to complete the foreign key dependencies.

CHECK Clause



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. If a row evaluates to *false* for any check constraint defined on a table during an insert or update, the database server returns an error.

Check constraints are defined using *search conditions*. The search condition cannot contain subqueries; aggregates; host variables; rowids; the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions; or stored procedure calls.

Warning: *When you specify a date value in a search condition, 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 search condition. When you specify a 2-digit year, the **DBCENTURY** environment variable can affect how the database server interprets the search condition, so the check constraint might not work as you intended. See the "Informix Guide to SQL: Reference" for more information on the **DBCENTURY** environment variable.*

The Column-Level Constraint definition on page 1-227 and the Table-Level Constraint definition on page 1-228 refer to the CHECK clause.

Defining Check Constraints at the Column Level

If 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. For example, as the following statement shows, the table **my_accounts** has two columns with check constraints:

```
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 wanted to test that **acct1** had a larger balance than **acct2**, you would not be able to create the check constraint at the column level. To create a constraint that checks values in more than one column, you must define the constraint at the table level.

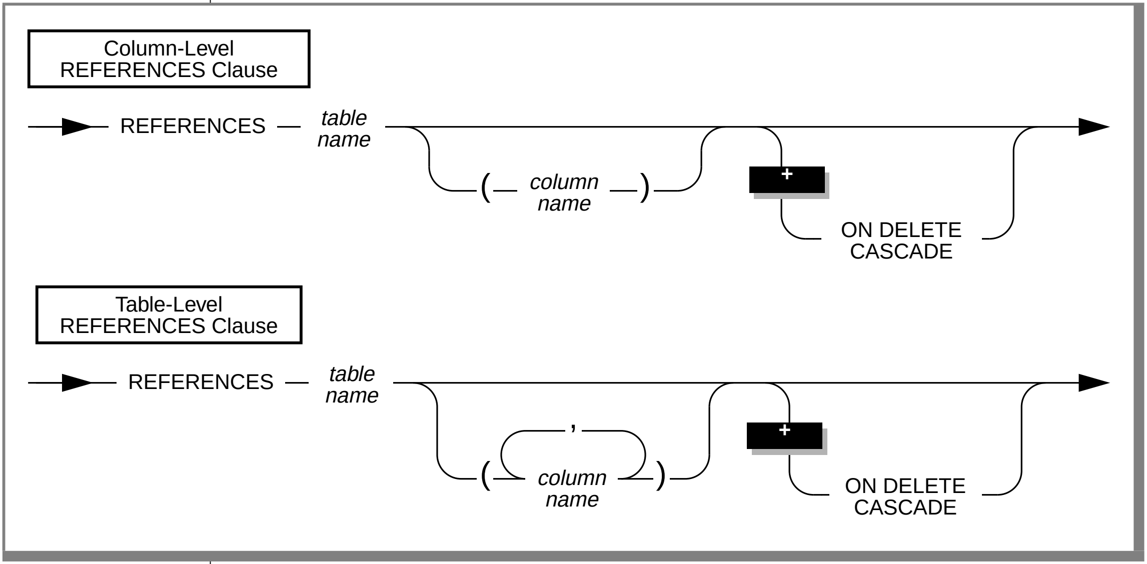
Defining Check Constraints at the Table Level

When you defined a check constraint at the table level, each column in the search condition must be a column in that table. You cannot create a check constraint for columns across tables. The next example builds the same table and columns as the previous example. However, the check constraint now spans two columns in the table.

```
CREATE TABLE my_accounts (
    chk_id      SERIAL PRIMARY KEY,
    acct1       MONEY,
    acct2       MONEY,
    CHECK (acct1 > acct2))
```

In this example, the **acct1** column must be greater than the **acct2** column, or the insert or update fails.

REFERENCES Clause



Element	Purpose	Restrictions	Syntax
column name	A referenced column or columns in the referenced table	You must observe restrictions on the column type and the number and length of columns. See "Restrictions on the Column Name Variable" on page 1-233.	Identifier, p. 1-962
table name	The name of the referenced table	The referenced table must reside in the same database as the referencing table.	Table Name, p. 1-1044

The REFERENCES clause appears in the Column-Level Constraint definition on page 1-227.

Restrictions on the Column Name Variable

You must observe the following restrictions on the *column name* variable in the REFERENCES clause:

- The referenced column must be a unique or primary-key column.

  That is, the referenced column must already include 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 INTEGER if the referenced column is SERIAL or INT8 if the referenced column is SERIAL8.
- You cannot place a referential constraint on a BYTE or TEXT column.
- 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, and the total length of the columns cannot exceed 390 bytes.

Using the REFERENCES Clause

In a referential relationship, the *referenced* column is a column or set of columns within a table that uniquely identifies each row in the table. The referenced column or set of columns must have a unique or primary-key constraint.

The *referencing* column is the column or set of columns that refers to the referenced columns. Unlike a referenced column, the *referencing* column or set of columns can contain null and duplicate values. However, every non-null value in the referencing columns must match a value in the referenced columns. When a referencing column meets this criteria, it is called a foreign key.

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.

You can use the REFERENCES clause to establish a referential relationship between two tables or within the same table. The referenced and referencing tables must be in the same database.

For example, you can have an **employee** table where the **emp_no** column uniquely identifies every employee through an employee number. The **mgr_no** column in that table contains the employee number of the manager who manages that employee. In this case, **mgr_no** is the foreign key (the child) that references **emp_no,** the primary key (the parent).

Using Column-Level REFERENCES Constraints

You can reference only one column when you use the REFERENCES clause at the column level (that is, when you use the REFERENCES clause with the "Column-Level Constraint Definition" on page 1-227).

The following example creates two tables, **accounts** and **sub_accounts**. A referential constraint is created between the foreign key, **ref_num**, in the **sub_accounts** table and the primary key, **acc_num**, 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))
```

The **ref_num** is not explicitly called a foreign key in the column- definition syntax. At the column level, the foreign-key designation is applied automatically.

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

In the preceding example, you can simply reference the **accounts** table without specifying a column. Because **acc_num** is the primary key of the **accounts** table, it becomes the referenced column by default.

Using Table-Level REFERENCES Constraints

You can specify multiple columns when you are using the REFERENCES clause at the table level.

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, then the foreign key also must be a set of columns that corresponds to the primary key. The following example creates two tables. The first table has a multiple-column primary key, and the second table has a referential constraint that references this 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 primary key, **acc_num** and **acc_type**, in the **accounts** table. If, during an insert, 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. Likewise, if you attempt to update **sub_accounts** with values for **ref_num** and **ref_type** that do not correspond to an equivalent set of values in **acc_num** and **acc_type** (from the **accounts** table), the database server returns an error.

If you are referencing a primary key in another table, you do not have to state the primary-key columns in that table explicitly. Referenced tables that do not specify the referenced columns default to the primary-key columns. You can rewrite the references section of the previous example as follows:

```
...
    FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
...
```

Because **acc_num** and **acc_type** is the primary key of the **accounts** table, and no other columns are specified, the foreign key, **ref_num** and **ref_type**, references those columns.

Locking Implications

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 done. 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.

Using ON DELETE CASCADE

Cascading deletes allow you to specify whether you want rows deleted in the child table when rows are deleted in the parent table. Unless you specify cascading deletes, the default prevents you from deleting data in the parent table if child tables are associated with the parent table. With the ON DELETE CASCADE clause, when you delete a row in the parent table, any rows associated with that row (foreign keys) in a child table are also deleted. 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 clause on the foreign key:

```
CREATE TABLE all_candy
    (candy_num  SERIAL PRIMARY KEY,
     candy_makerCHAR(25));

CREATE TABLE hard_candy
    (candy_num     INT,
     candy_flavor  CHAR(20),
     FOREIGN KEY (candy_num) REFERENCES all_candy
     ON DELETE CASCADE)
```

With cascading deletes specified on the child table, in addition to deleting a candy item from the **all_candy** table, the delete cascades to the **hard_candy** table associated with the **candy_num** foreign key. If you indicate cascading deletes, when you delete a row from a parent table, the database server deletes the associated matching rows from the child table.

You specify cascading deletes with the REFERENCES clause on a column-level or table-level constraint. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to perform cascading deletes; however, you do need the Delete privilege on tables referenced in the DELETE statement.

Locking and Logging

During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables. You must turn logging on when you perform the deletes. If logging is turned off in a database, even temporarily, deletes do not cascade. This restriction applies because if logging is turned off, you cannot roll back any actions. For example, if a parent row is deleted, and the system crashes before the child rows are deleted, the database will have dangling child records, which violates referential integrity. However, when logging is turned back on, subsequent deletes cascade.

What Happens to Multiple Children Tables

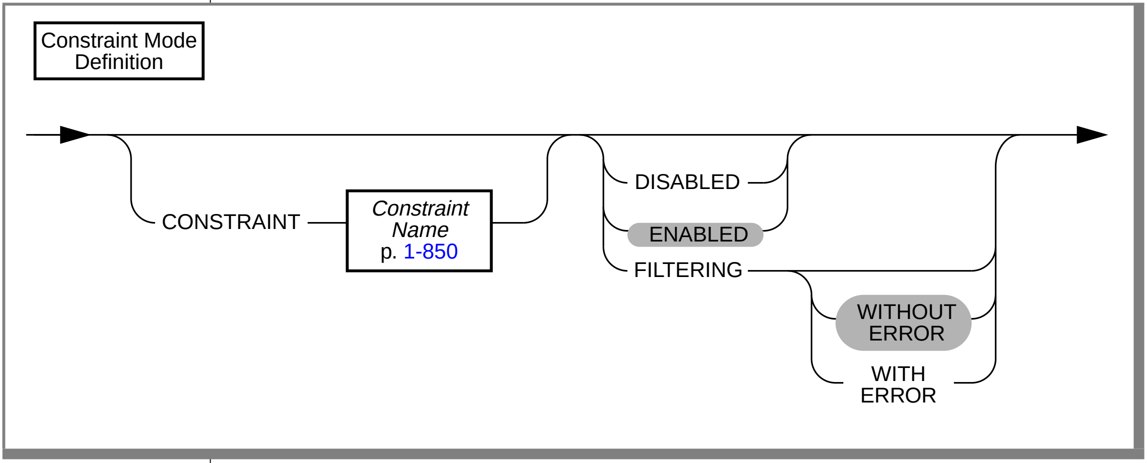
If you have a parent table with two child constraints, one child with cascading deletes specified and one child without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, the delete statement fails, and no rows are deleted from either the parent or child tables.

Restriction on Cascading Deletes

Cascading deletes can be used for most deletes. The only exception is correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a query that contains such a correlated subquery.

See the *Informix Guide to SQL: Tutorial* for a detailed discussion about cascading deletes.

Constraint Mode Definition



You can set the object mode of the constraint to the enabled, disabled, or filtering mode. For a discussion of object modes, see "Terminology for Object Modes" on page 1-645.

You can use the Constraint Mode Definition option for the following purposes:

- To assign a name to a column-level or table-level constraint
- To set any type of column-level constraint or table-level constraint to the disabled, enabled, or filtering object modes

The Column-Level Constraint Definition on page 1-227 and the Table-Level Constraint Definition on page 1-228 refer to the Constraint Mode Definition.

Description of Constraint Modes

You can set constraints in the following modes: disabled, enabled, and filtering. The following list explains these modes and options.

Constraint Mode	Effect
disabled	A constraint created in disabled mode is not enforced during insert, delete, and update operations.
enabled	A constraint created in enabled mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement fails.
filtering	A constraint created in filtering mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement continues processing, but the bad row is written to the violations table associated with the target table. Diagnostic information about the constraint violation is written to the diagnostics table associated with the target table.

If you choose filtering mode, you can specify the WITHOUT ERROR or WITH ERROR options. The following list explains these options.

Error Option	Effect
WITHOUT ERROR	When a filtering-mode constraint is violated during an insert, delete, or update operation, no integrity-violation error is returned to the user.
WITH ERROR	When a filtering-mode constraint is violated during an insert, delete, or update operation, an integrity-violation error is returned to the user.

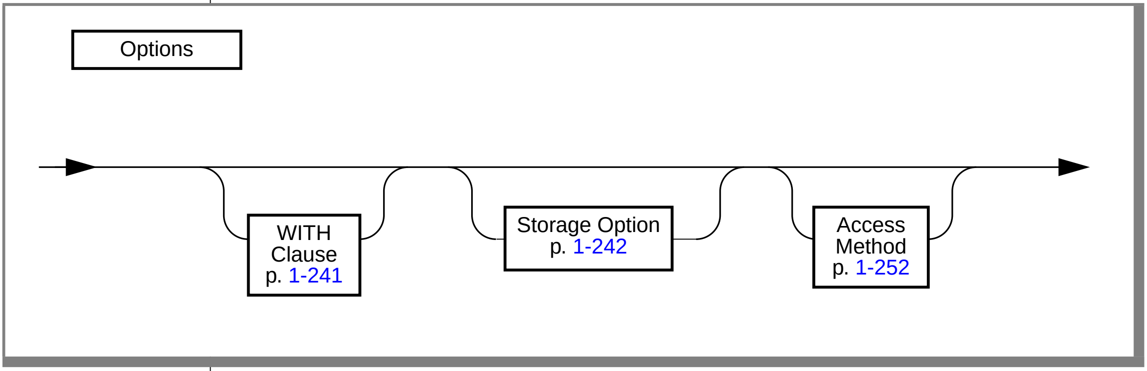
Using Constraint Mode Definitions

You must observe the following rules concerning the use of constraint mode definitions:

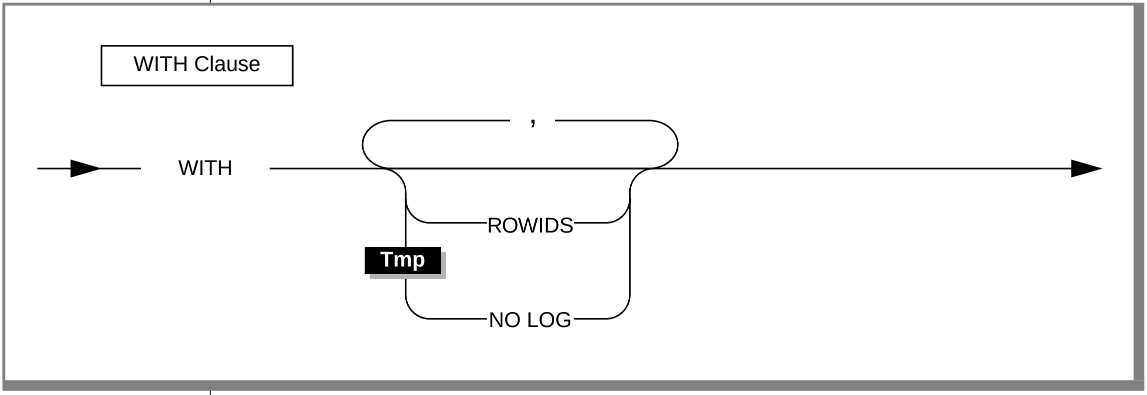
- If you do not specify the object mode of a column-level constraint or table-level constraint explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering-mode constraint, the default error option is WITHOUT ERROR.
- Constraints defined on temporary tables are always in the enabled mode. You cannot create a constraint on a temporary table in the disabled or filtering mode, nor can you use the SET statement to switch the object mode on a temporary table to the disabled or filtering mode.
- You cannot assign a name to a not null constraint on a temporary table.
- You cannot create a constraint on a table that is serving as a violations or diagnostics table for another table.

Options

The CREATE TABLE options let you specify logging and rowid options, optional storage locations, and user-defined access methods.



WITH Clause



Using WITH ROWIDS

Nonfragmented tables contain a hidden column called the rowid column. However, fragmented tables do not contain this column. If a table is fragmented, you can use the WITH ROWIDS clause to add the rowid column to the table. The database server assigns each row in the rowid column a unique number that remains stable for the life of the row. The database server uses an index to find the physical location of the row. After you add the rowid column, each row contains an additional 4 bytes to store the rowid.

You cannot use the WITH ROWIDS clause with typed tables.

Important: *Use the WITH ROWIDS clause only on fragmented tables. In nonfragmented tables, the rowid column remains unchanged. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the rowid column.*

Using WITH NO LOG

You must use the WITH NO LOG keywords on temporary tables created in temporary dbspaces. Using the WITH NO LOG keywords prevents logging of temporary tables in databases started with logging.

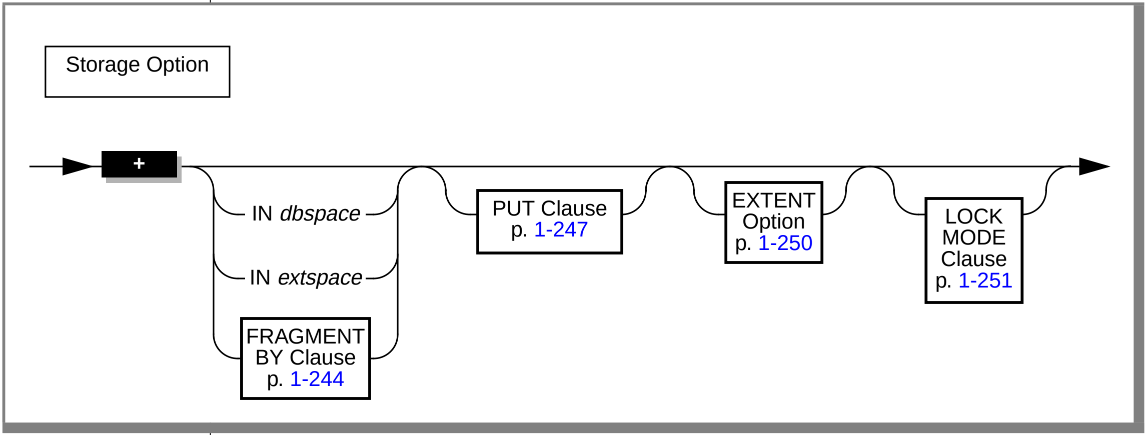
If you use the WITH NO LOG keywords in a CREATE TABLE statement, and the database does not use logging, the WITH NO LOG option is ignored.

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 example shows how to prevent logging temporary tables in a database that uses logging:

```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
    WITH NO LOG
```

Storage Option



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace in which to store the table. The default for database tables is the dbspace in which the current database resides.	Specified dbspace must already exist.	Identifier, p. 1-962
extspace	The name of an external space in which to store a virtual table	Specified extspace must already exist.	

The storage option allows you to specify where the table is stored and the locking granularity for the table. If you use the Access Method Option on page 1-252 to specify an access method, the spaces named must be supported by that access method.

You can specify a dbspace for the table that is different from the storage location specified for the database, or fragment the table into several dbspaces. You can also specify an sbspace for each smart large object (CLOB or BLOB) using the PUT clause.

***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 864.*

The following statement creates the **foo** table. The data for the table is fragmented into the **dbs1** and **dbs2** dbspaces. However, 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 in the **eps** column is assigned to the **blb1** blobspace.

```
create table foo
(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)
```

The Storage Option appears in the Options diagram on page 1-240.

IN dbspace Clause

The IN *dbspace* clause allows you to isolate a table. The dbspace that you specify must already exist. If you do not specify the IN *dbspace* clause, the default is the dbspace where the current database resides. Temporary tables do not have a default dbspace. For further information about storing temporary tables, see the "Temporary Tables" on page 1-215.

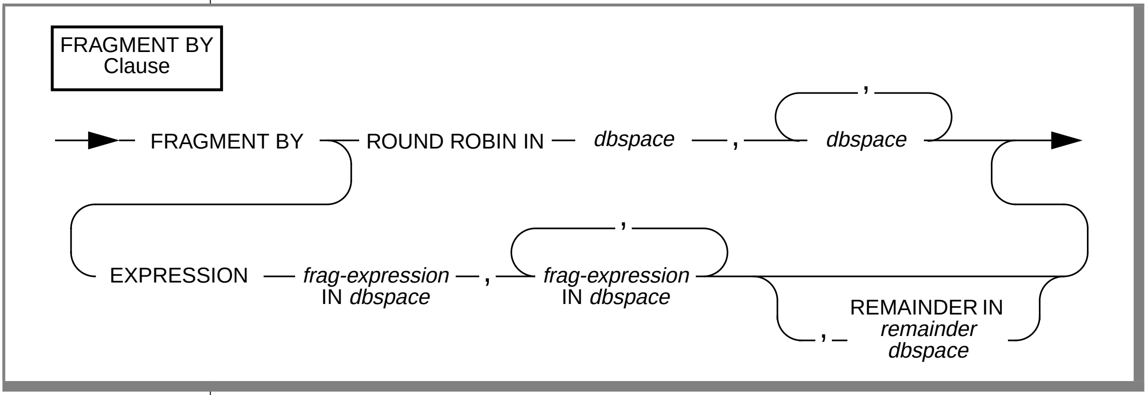
For example, if the **stores7** database is in the **stockdata** dbspace, but you want the **customer** data placed in a separate dbspace called **custdata**, use the following statements:

```
CREATE DATABASE stores7 IN stockdata

CREATE TABLE customer
    (
    customer_num    SERIAL(101),
    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)
        )
     IN custdata EXTENT SIZE 16

        .
        .
        .
```

For more information about storing your tables in separate dbspaces, see the *INFORMIX-Universal Server Administrator's Guide*.

FRAGMENT BY Clause

The FRAGMENT BY clause allows you to create fragmented tables. Fragmentation means that groups of rows within a table are stored together in the same dbspace.


FRAGMENT BY Clause
FRAGMENT BY
ROUND ROBIN IN
dbspace
,
,
dbspace
EXPRESSION
frag-expression IN dbspace
,
,
frag-expression IN dbspace
, REMAINDER IN remainder dbspace

Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that contains a table fragment	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces. The dbspaces must exist when you execute the statement.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
frag-expression	An expression that defines a fragment where a row is to be stored using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in a fragment expression.	Expression, p. 1-876, and Condition, p. 1-831
remainder dbspace	The dbspace that contains table rows that do not meet the conditions defined in any fragment expression	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-962

(2 of 2)

Use the FRAGMENT BY clause to define the distribution scheme, either round-robin or expression-based.

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.

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a dbspace. Each fragment expression within the rule isolates data and aids the database server in searching for rows. 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 following example shows:

  ```
  ...
      FRAGMENT BY EXPRESSION
      c1 < 100 IN dbsp1,
      c1 >= 100 and c1 < 200 IN dbsp2,
      c1 >= 200 IN dbsp3
  ```

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

  ```
  ...
      FRAGMENT BY EXPRESSION
      MOD(id_num, 3) = 0 IN dbsp1,
      MOD(id_num, 3) = 1 IN dbsp2,
      MOD(id_num, 3) = 2 IN dbsp3
  ```

- Arbitrary rule

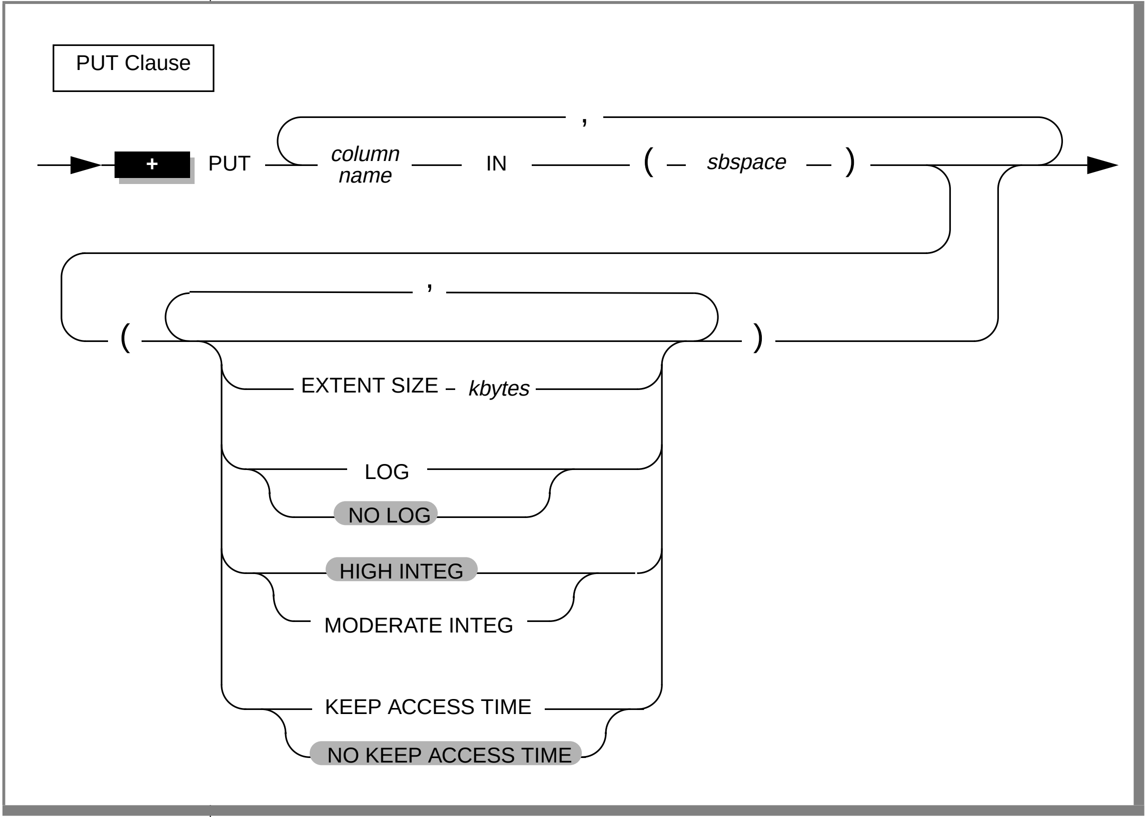
  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as the following example shows:

  ```
  ...
      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: *When you specify a date value in a fragment expression, 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 the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

PUT Clause

The PUT clause specifies storage information for smart large objects (CLOB and BLOB columns).



Element	Purpose	Restrictions	Syntax
column name	The name of the smart-large-object column to store in the specified sbspace	Column must be BLOB or CLOB data type.	Identifier, p. 1-962
kbytes	The number of kilobytes to allocate for the extent size	Number must be an integer value.	
sbspace	An area of storage used for smart large objects	The sbspace must exist.	

A smart large object is contained in a single sbspace. The SB_SPACE_NAME configuration parameter specifies the system default in which smart large objects are created unless you specify another area.

Important: *The PUT clause does not affect the storage of simple large-object data types (BYTE and TEXT).*

The PUT clause appears in the Storage Option on page 1-242.

EXTENT SIZE Option of the PUT Clause

The EXTENT SIZE option of the PUT clause specifies the number of kilobytes in an smart large-object extent. The EXTENT SIZE should be a multiple of the sbspace page size. If it is not, Universal Server rounds up the number to the nearest multiple of the sbspace page size.

If the extent size is not specified, or if no extent of the specified size exists, Universal Server uses the larger of:

- the size of the write request.
- the smallest extent size for the sbspace.

After eight extension operations for a single smart large object, Universal Server automatically doubles the extent size for that smart large object, to avoid having a large number of extents.

LOG and NO LOG Options of the PUT Clause

Use the LOG option of the PUT clause when you want the database server to follow the logging procedure used with the current database log for the corresponding smart large object.

Warning: *Use of the LOG option can generate large amounts of log traffic and increase the risk that the logical log fills up.*

Instead of full logging, you might turn off logging when you load the smart large object initially, and then turn logging back on once the smart large object has been 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 NO LOG option is the default logging behavior for smart large objects.

HIGH INTEG and MODERATE INTEG Option

The HIGH INTEG option of the PUT clause provides user data pages that contain a page header and a page trailer. The database server uses the header and trailer to detect incomplete writes and data corruption. The HIGH INTEG option is the default.

Important: *After you have specified the HIGH INTEG option, you cannot use the ALTER TABLE statement to change to MODERATE INTEG.*

The MODERATE INTEG option provides a lower level of data integrity but is faster. It does not produce page headers or trailers on user data pages, so it preserves the user data alignment on pages. The MODERATE INTEG option is useful for moving large volumes of data through the server when very high data integrity is not required. Audio and video applications may benefit from a MODERATE INTEG option.

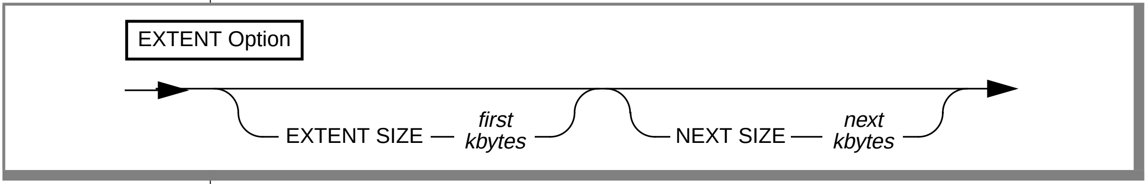
KEEP ACCESS TIME and NO KEEP ACCESS TIME Options

The KEEP ACCESS TIME option of the PUT clause tells the database server to record, in the smart large-object meta data, the system time at which the corresponding smart large object was last read or written. This capability is provided for compatibility with the Illustra interface.

When you specify the NO KEEP ACCESS TIME option, the database server does not track the system time at which the corresponding smart large object was last read or written. This option provides better performance than the KEEP ACCESS TIME option.

The NO KEEP ACCESS TIME option is the default.

EXTENT Option



Element	Purpose	Restrictions	Syntax
first kbytes	The length in kilobytes of the first extent for the table. The default length is eight times the disk page size on your system. For example, if you have a 2-kilobyte page system, the default length is 16 kilobytes.	The minimum length is four times the disk page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is eight kilobytes. The maximum length is equal to the chunk size.	Expression, p. 1-876
next kbytes	The length in kilobytes for the subsequent extents. The default length is eight times the disk page size on your system. For example, if you have a 2-kilobyte page system, the default length is 16 kilobytes.	The minimum length 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 maximum length is equal to the chunk size.	Expression, p. 1-876

See the *INFORMIX-Universal Server Performance Guide* for a discussion about calculating extent sizes.

The following example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

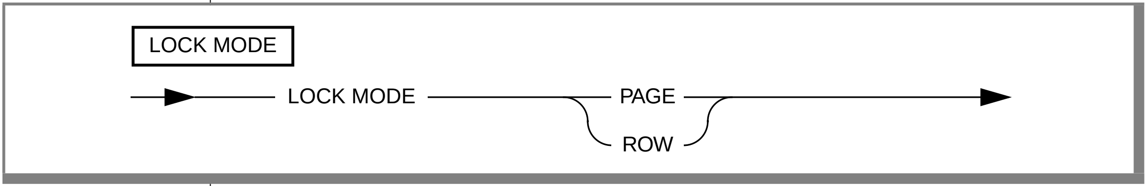
```
CREATE TABLE emp_info
    (
    f_name      CHAR(20),
    l_name      CHAR(20),
    position    CHAR(20),
    start_date  DATETIME YEAR TO DAY,
    comments    VARCHAR(255)
    )
EXTENT SIZE 20
```

Revising Extent Sizes for Unloaded Tables

You can revise the CREATE TABLE statements in generated schema files to revise the extent and next-extent sizes of unloaded tables. See the *INFORMIX-Universal Server Administrator's Guide* for information about revising extent sizes.

The EXTENT option appears in the Storage Option on page 1-242.

LOCK MODE Clause



The default locking granularity is a page.

Row-level locking provides the highest level of concurrency. However, if you are using many rows at one time, the lock-management overhead can become significant. Also, you might exceed the maximum number of locks available, depending on the configuration of your database-server system.

Page locking allows you to obtain and release one lock on a whole page of rows. Page 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.

You can change the lock mode of an existing table with the ALTER TABLE statement.

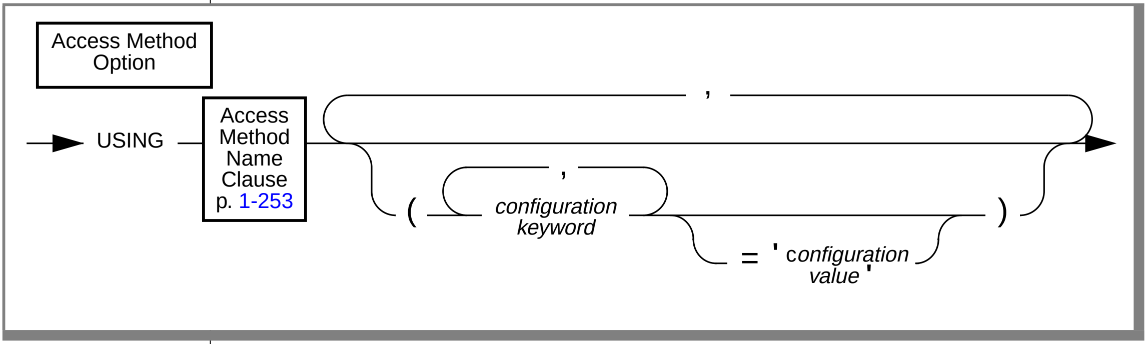
The Lock Mode clause appears in the Storage Option on page 1-242.

Access Method Option

A primary access method is a set of routines that perform all of the operations needed to make a table available to a server, such as create, drop, insert, delete, update, and scan. Universal Server provides a built-in primary access method.

An virtual table is managed outside of the database server but can be accessed by Universal Server users with SQL statements. Access to an virtual table requires a user-defined primary access method.

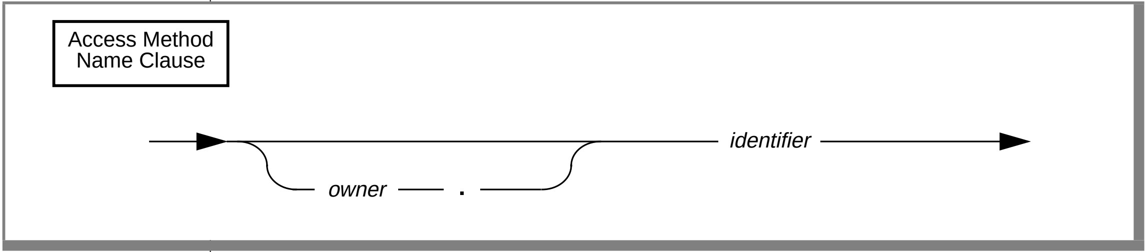
DataBlade Modules can provide other primary access methods to access virtual tables. When you access a virtual table, the database server calls the routines associated with that access method rather than the built-in table routines. For more information on these other primary access methods, refer to the DataBlade user guides.



Element	Purpose	Restrictions	Syntax
access method name	Name of the access method to be used with this table	Access method must already exist.	See "Access Method Name Clause"
configuration keyword	One of the configuration keywords associated with the specified access method name.	Keyword must already exist.	Keywords can be up to 18 bytes in length.
configuration value	Value of the specified configuration keyword. Configuration values are not required with all keywords. 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.	Value must be defined by the access method.	Value must be in quotation marks. Values can be up to 236 bytes in length.

The Access Method Option appears in the Options clause on page 1-240.

Access Method Name Clause



For example, if there was an access method called **textfile**, you could specify that access method in the following Access Method clause:

```
create table mybook
(... )
using textfile (delimiter=':')
```

The Access Method Name clause appears in the Access Method Option on page 1-252.

References

See the ALTER TABLE, CREATE INDEX, CREATE DATABASE, DROP TABLE, and SET statements in this manual. Also see the Condition, Data Type, Identifier, and Table Name segments.

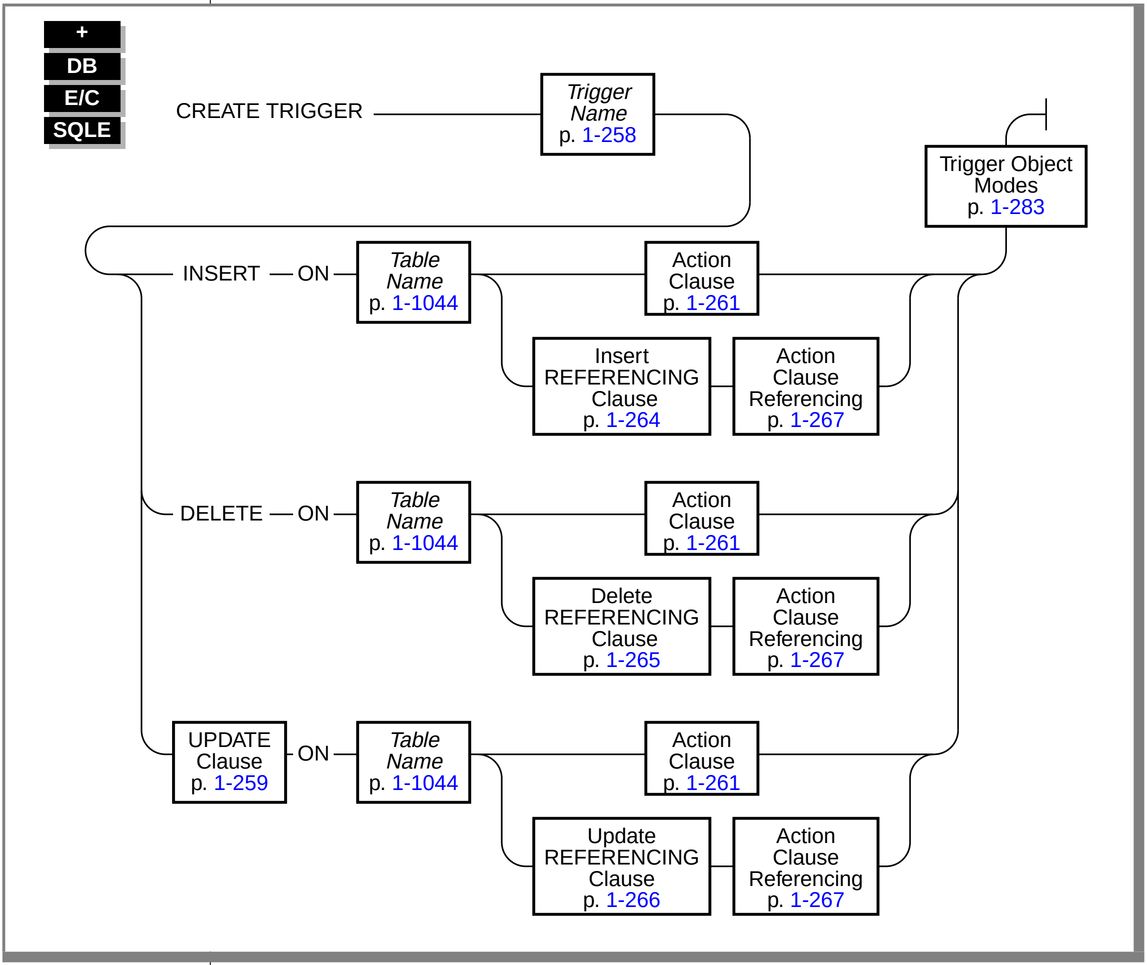
In the *Informix Guide to SQL: Tutorial*, see the discussion of data-integrity constraints and the discussion of the ON DELETE CASCADE clause. Also see the discussion of creating a database and tables in the same book.

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of extent sizing.

CREATE TRIGGER

Use the CREATE TRIGGER statement to create a trigger on a table in the database. A trigger is a database object that automatically sets off a specified set of SQL statements when a specified event occurs.

Syntax


+
DB
E/C
SQLE
CREATE TRIGGER
Trigger Name p. 1-258
Trigger Object Modes p. 1-283
INSERT
ON
Table Name p. 1-1044
Action Clause p. 1-261
Insert REFERENCING Clause p. 1-264
Action Clause Referencing p. 1-267
DELETE
ON
Table Name p. 1-1044
Action Clause p. 1-261
Delete REFERENCING Clause p. 1-265
Action Clause Referencing p. 1-267
UPDATE Clause p. 1-259
ON
Table Name p. 1-1044
Action Clause p. 1-261
Update REFERENCING Clause p. 1-266
Action Clause Referencing p. 1-267

Usage

You must be either the owner of the table or have DBA status to create a trigger on a table.

You can use roles with triggers. Role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements can be triggered inside a trigger. 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.

You can define a trigger with a stand-alone CREATE TRIGGER statement.

DB

You can define a trigger as part of a schema by placing the CREATE TRIGGER statement inside a CREATE SCHEMA statement. ♦

You can create a trigger only on a table in the current database. You cannot create a trigger on a temporary table, a view, or a system catalog table.

You cannot create a trigger inside a stored procedure if the procedure is called inside a data manipulation statement. For example, you cannot create a trigger inside the stored procedure **sp_items** in the following INSERT statement:

```
INSERT INTO items EXECUTE PROCEDURE sp_items
```

See “Data Manipulation Statements” on page 1-15 for a list of data manipulation statements.

ESQL

If you are embedding the CREATE TRIGGER statement in an ESQL/C program, you cannot use a host variable in the trigger specification. ♦

You cannot use a stored procedure variable in a CREATE TRIGGER statement.

Trigger Event

The trigger event specifies the type of statement that activates a trigger. The trigger event can be an INSERT, DELETE, or UPDATE statement. Each trigger can have only one trigger event. The occurrence of the trigger event is the *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. For each table, you can define multiple triggers that are activated by UPDATE statements. See "UPDATE Clause" on page 1-259 for more information about multiple triggers on the same table.

You cannot define a DELETE trigger event on a table with 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 triggered actions. See "Action Clause" on page 1-261 and "Triggered Action List" on page 1-268 for more information on the behavior of triggered actions.

If Universal Server is the database server, a triggering statement from an external database server can activate the trigger. As shown in the following example, 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 stores7@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 stores7@dbserver1:newtab
    SELECT item_num, order_num, quantity, stock_num,
manu_code,
    total_price FROM items;
```

Trigger Events with Cursors

If the triggering statement uses a cursor, the complete trigger is activated each time the statement executes. For example, if you declare a cursor for a triggering INSERT statement, each PUT statement executes the complete trigger. Similarly, if a triggering UPDATE or DELETE statement contains the clause WHERE CURRENT OF, each update or delete activates the complete trigger. This behavior is different from what occurs when a triggering statement does not use a cursor and updates multiple rows. In this case, the set of triggered actions executes only once. For more information on the execution of triggered actions, see "Action Clause" on page 1-261.

Privileges on the Trigger Event

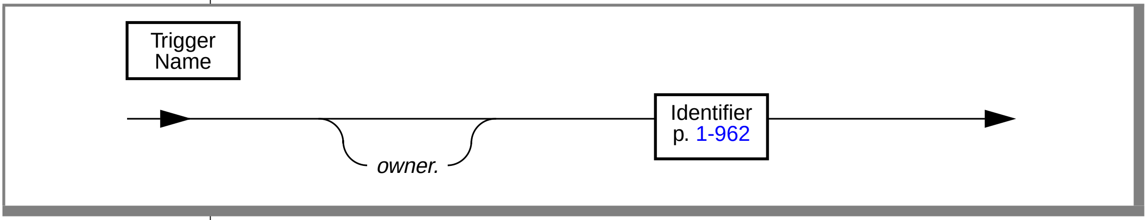
You must have the appropriate Insert, Delete, or Update privilege on the triggering table to execute the INSERT, DELETE, or UPDATE statement that is the trigger event. The triggering statement might still fail, however, if you do not have the privileges necessary to execute one of the SQL statements in the action clause. When the triggered 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 you need to execute a trigger, see "Privileges to Execute Triggered Actions" on page 1-277.

Impact of Triggers

The INSERT, DELETE, and UPDATE 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 triggering data manipulation statement depends on the complexity of the triggered action and whether it initiates other triggers. Obviously, the elapsed time for the triggering data manipulation statement increases as the number of cascading triggers increases. For more information on triggers that initiate other triggers, see "Cascading Triggers" on page 1-278.

Trigger Name



Element	Purpose	Restrictions	Syntax
owner	The user name of the owner of the trigger	The specified name must be a valid user name.	Identifier, p. 1-962

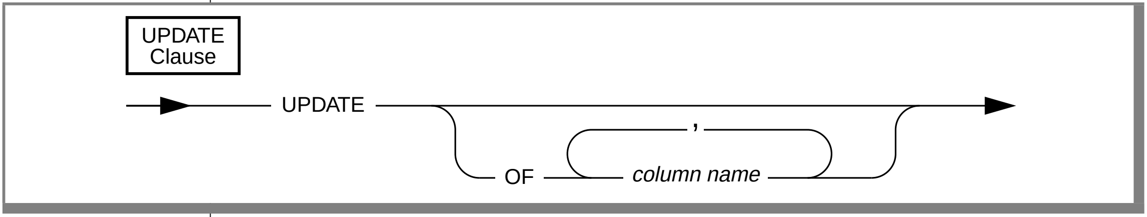
When you create a trigger, the name of the trigger must be unique within a database.

ANSI

When you create a trigger, the *owner.name* combination (the combination of the owner name and trigger name) must be unique within a database. ♦

For information about the relationship between the trigger owner's privileges and the privileges of other users, see "Privileges to Execute Triggered Actions" on page 1-277.

UPDATE Clause



Element	Purpose	Restrictions	Syntax
column name	The name of a column or columns that activate the trigger. The default is all the columns in the table on which you create the trigger.	The specified columns must belong to the table on which you create the trigger. If you define more than one update trigger on a table, the column lists of the triggering statements must be mutually exclusive.	Identifier, p. 1-962

If the trigger event is an UPDATE statement, the trigger executes when any column in the triggering column list is updated.

If the triggering UPDATE statement updates more than one of the triggering columns in a trigger, the trigger executes only once.

Defining Multiple Update Triggers

If you define more than one update trigger event on a table, the column lists of the triggers must be mutually exclusive. The following example shows that **trig3** is illegal on the **items** table because its column list includes **stock_num**, which is a triggering column in **trig1**. Multiple update triggers on a table cannot include the same columns.

```
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 Activates Multiple Triggers

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:

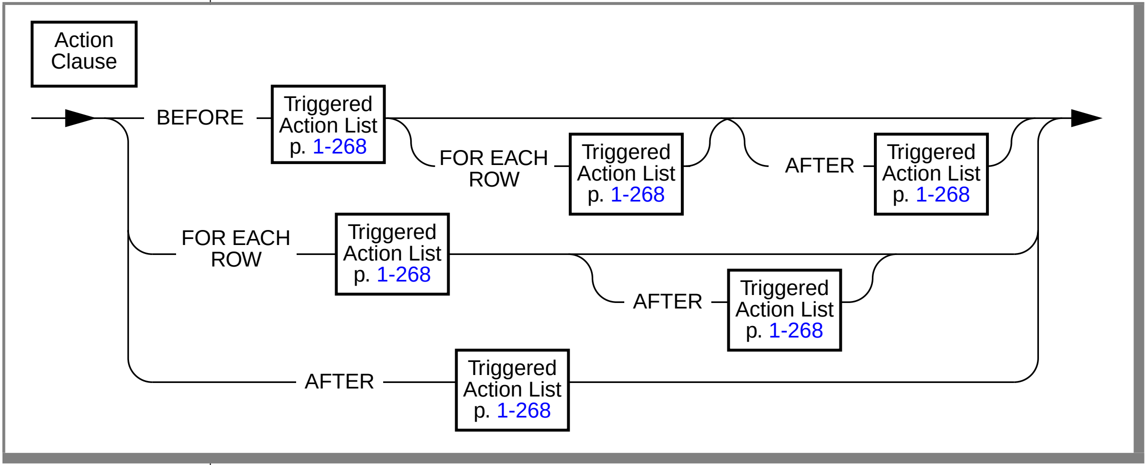
```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
    AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
    AFTER (UPDATE tabb SET z = z + 1);
```

The triggering statement is shown in the following example:

```
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**).

Action Clause


Action Clause
BEFORE
Triggered Action List p. 1-268
FOR EACH ROW
Triggered Action List p. 1-268
AFTER
Triggered Action List p. 1-268
FOR EACH ROW
Triggered Action List p. 1-268
AFTER
Triggered Action List p. 1-268
AFTER
Triggered Action List p. 1-268

The action clause defines the characteristics of triggered actions and specifies the time when these actions occur. You must define at least one triggered action, using the keywords BEFORE, FOR EACH ROW, or AFTER to indicate when the action occurs relative to the triggering statement. You can specify triggered actions for all three options on a single trigger, but you must order them in the following sequence: BEFORE, FOR EACH ROW, and AFTER. You cannot follow a FOR EACH ROW triggered action list with a BEFORE triggered action list. If the first triggered action list is FOR EACH ROW, an AFTER action list is the only option that can follow it. See "Action Clause Referencing" on page 1-267 for more information on the action clause when a REFERENCING clause is present.

BEFORE Actions

The BEFORE triggered action or actions execute once before the triggering statement executes. If the triggering statement does not process any rows, the BEFORE triggered actions still execute because the database server does not yet know whether any row is affected.

FOR EACH ROW Actions

The FOR EACH ROW triggered action or actions execute once for each row that the triggering statement affects. The triggered SQL statement executes after the triggering statement processes each row.

If the triggering statement does not insert, delete, or update any rows, the FOR EACH ROW triggered actions do not execute.

AFTER Actions

An AFTER triggered action or actions execute once after the action of the triggering statement is complete. If the triggering statement does not process any rows, the AFTER triggered actions still execute.

Actions of Multiple Triggers

When an UPDATE statement activates multiple triggers, the triggered actions merge. Assume that **taba** has columns **a**, **b**, **c**, and **d**, as the following 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 have triggered actions that are executed BEFORE, FOR EACH ROW, and AFTER, the triggered actions are executed in the following sequence:

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 the triggers as a single trigger, and the triggered action is the merged-action list. All the rules governing a triggered action apply to the merged list as one list, and no distinction is made between the two original triggers.

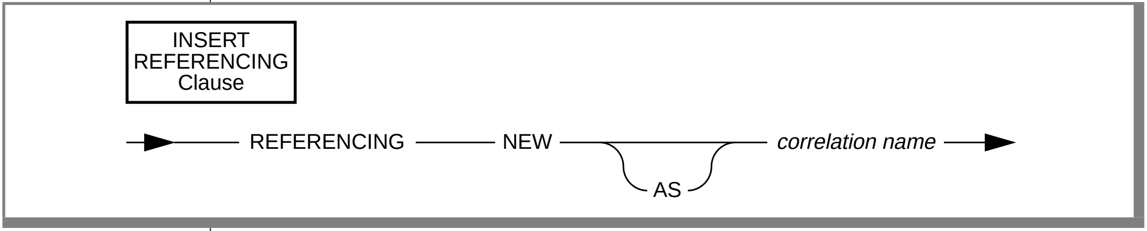
Guaranteeing Row-Order Independence

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 1-278.
- In the FOR EACH ROW section, avoid updating a table with values derived from the current row of the triggering table. If the triggered 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 triggered statement in the same FOR EACH ROW section, including any cascading triggered actions. If you modify a table in this section and refer to it later, the changes to the table might not be complete when you refer to it. Consequently, 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 triggered action can select. Furthermore, the result of most triggered actions is independent of row order. Consequently, you are responsible for ensuring that the results of the triggered actions are independent of row order.

INSERT REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to a new column value so that you can refer to it within the triggered action. The new column value in the triggering table is the value of the column after execution of the triggering statement.	The correlation name must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-962

Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing" on page 1-267.

To use the correlation name, precede the column name with the correlation name, followed by a period. 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 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 the use of correlation names, see "Using Correlation Names in Triggered Actions" on page 1-271.

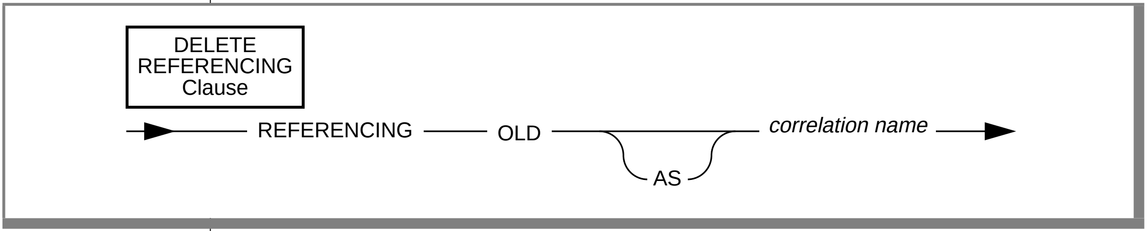
You can use the INSERT REFERENCING clause only if you define a FOR EACH ROW triggered action.

The following example illustrates the 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 advantage of the INSERT REFERENCING clause is that it allows you to refer to the data values that the trigger event in your triggered action produces.

DELETE REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to an old column value so that you can refer to it within the triggered action. The old column value in the triggering table is the value of the column before execution of the triggering statement.	The correlation name must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-962

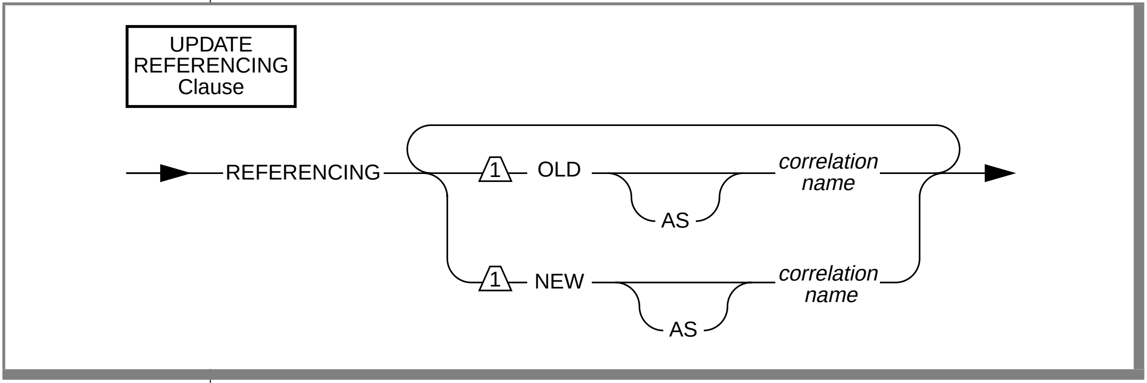
Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing" on page 1-267.

Use the correlation name to refer to an old column value by preceding the column name with the correlation name and a period (.). For example, if the old correlation name is **pre**, refer to the old value for the column **fname** as **pre.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. See "Using Correlation Names in Triggered Actions" on page 1-271 for the rules governing the use of correlation names.

You can use the DELETE REFERENCING clause only if you define a FOR EACH ROW triggered action.

UPDATE REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to an old or new column value so that you can refer to it within the triggered action. The old column value in the triggering table is the value of the column before execution of the triggering statement. The new column value in the triggering table is the value of the column after the statement executes.	You can specify a correlation name for an old column value only (OLD option), for a new column value only (NEW option), or for both the old and new column values. Each correlation name you specify must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-962

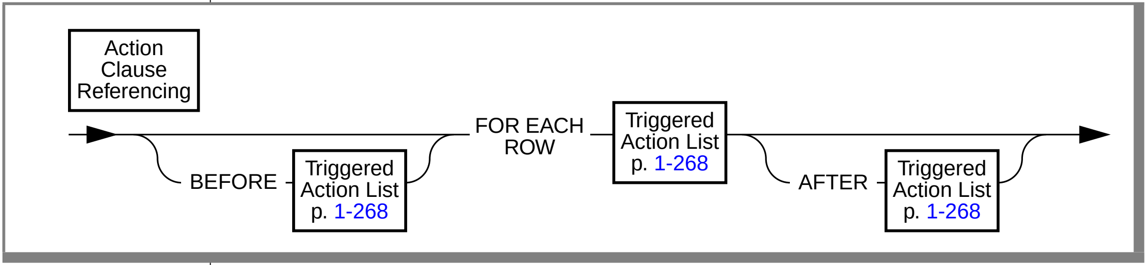
After you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing".

Use the correlation name to refer to an old or new column value by preceding the column name with the correlation name and a period (.). For example, if the new correlation name is **post**, you refer to the new value for the 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. See "Using Correlation Names in Triggered Actions" on page 1-271 for the rules that govern the use of correlation names.

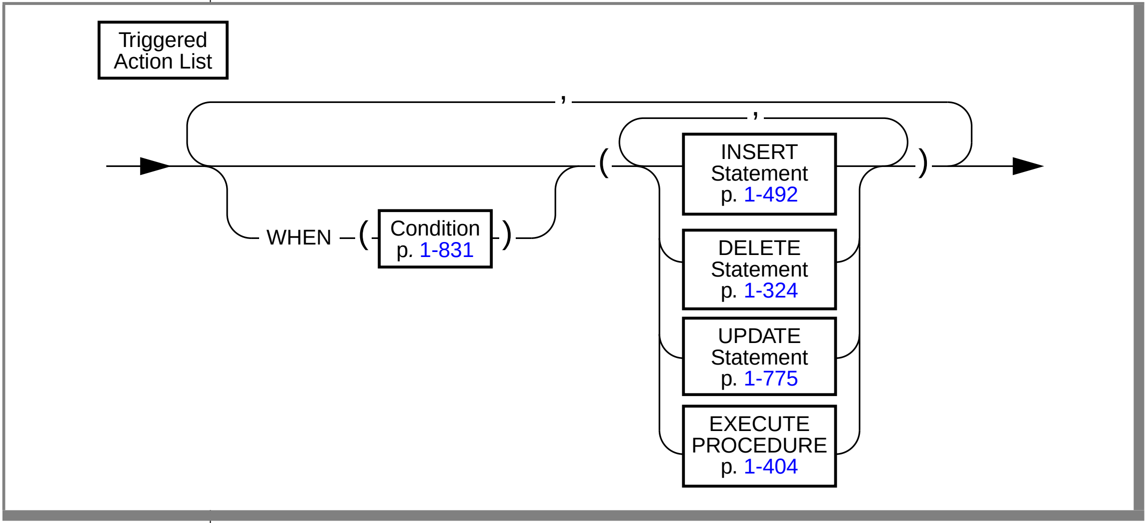
You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW triggered action.

Action Clause Referencing


Action Clause Referencing
BEFORE
Triggered Action List p. 1-268
FOR EACH ROW
Triggered Action List p. 1-268
AFTER
Triggered Action List p. 1-268

If the CREATE TRIGGER statement contains an INSERT REFERENCING clause, a DELETE REFERENCING clause, or an UPDATE 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. See "Action Clause" on page 1-261 for information on the BEFORE, FOR EACH ROW, and AFTER triggered-action lists.

Triggered Action List


Triggered Action List
,
,
WHEN
(
Condition p. 1-831
)
(
INSERT Statement p. 1-492
DELETE Statement p. 1-324
UPDATE Statement p. 1-775
EXECUTE PROCEDURE p. 1-404
)

The triggered action consists of an optional WHEN condition and the action statements. Objects that are referenced in the triggered action, that is, tables, columns, and stored procedures, must exist when the CREATE TRIGGER statement is executed. This rule applies only to objects that are referenced directly in the trigger definition.

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. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the date expression. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the date expression, so the triggered action might produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

WHEN Condition

The WHEN condition lets you make the triggered action dependent on the outcome of a test. When you include a WHEN condition in a triggered action, if the triggered action evaluates to *true*, the actions in the triggered action list execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered action list are not executed. If the triggered action is in a FOR EACH ROW section, its search 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))
```

A routine that executes inside the WHEN condition carries the same restrictions as a routine that is called in a data manipulation statement. See the *Extending INFORMIX-Universal Server: User-Defined Routines* manual for more information about a routine that is called within a data manipulation statement.

Action Statements

The triggered-action statements can be INSERT, DELETE, UPDATE, or EXECUTE PROCEDURE statements. If a triggered-action list contains multiple statements, these statements execute in the order in which they appear in the list.

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 in a multiple-row INSERT statement

Using Keywords

If you use the INSERT, DELETE, UPDATE, or EXECUTE keywords 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 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**. 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)
```

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

The following rules apply when you use correlation names in triggered actions:

- You can use the correlation names for the old and new column values only in statements in the FOR EACH ROW triggered-action list. You can use the old and new correlation names to qualify any column in the triggering table in either the WHEN condition or the triggered SQL statements.
- 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 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 routine that is a triggered action.

When to Use Correlation Names

In an SQL statement in a FOR EACH ROW triggered action, 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 nonqualified 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.

When a column is not qualified by a correlation name, and the statement is valid independent of the triggered action, the column name refers to the current value in the database. In the triggered action for trigger **t1** in the following example, **mgr** in the WHERE clause of the correlated subquery is an unqualified column from 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

The following table summarizes the value retrieved when you use the column name qualified by the old correlation name and the column name qualified 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 table.

Term	Meaning
original value	is the value before the triggering statement.
current value	is the value after the triggering statement.
(N)	cannot be changed by triggered action.
(U)	can be updated by triggered statements; value may 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.

Action on the Triggering Table

You cannot reference the triggering table in any triggered SQL statement, with the following exceptions:

- The trigger event is UPDATE and the triggered SQL statement is also UPDATE, and the columns in both statements, including any nontriggering columns in the triggering UPDATE, are mutually exclusive.

  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 triggered actions in the following example. The first UPDATE statement is a valid triggered 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;-- ILLEGAL
  ```

- The triggered SQL statement is a SELECT statement. The SELECT statement can be a triggered statement in the following instances:
    - The SELECT statement appears in a subquery in the WHEN clause or a triggered-action statement.
    - The triggered action is a stored procedure, and the SELECT statement appears inside the stored procedure.

This rule, which states that a triggered SQL statement cannot reference the triggering table, with the two noted exceptions, applies recursively to all cascading triggers, which are considered part of the initial trigger. This situation 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 the following UPDATE statement is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1)
```

Then in the cascading triggers shown in the following example, **trig2** fails at runtime because it references column **b**, which is updated by the triggering UPDATE statement. See "Cascading Triggers" on page 1-278 for more information about cascading triggers.

```
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);
```

Rules for Procedures

The following rules apply to a procedure that is used as a triggered action:

- The routine cannot be a cursory procedure (that is, a procedure that returns more than one row) in a place where only one row is expected.
- When an EXECUTE PROCEDURE statement is the triggered action, you can specify the INTO clause only for an UPDATE trigger when the triggered action occurs in the FOR EACH ROW section. In this case, the INTO clause can contain only column names from the triggering table. The following 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)
  ```

  When the INTO clause appears in the EXECUTE PROCEDURE statement, the database server updates the columns named there with the values returned from the routine. The database server performs the update immediately upon returning from the routine. See "EXECUTE PROCEDURE" on page 1-404 for more information about the statement.
- You cannot use the old or new correlation name inside the routine. If you need to use the corresponding values in the procedure, you must pass them as parameters. The routine should be independent of triggers, and the old or new correlation name do not have any meaning outside the trigger.
- You cannot use the following statements inside the routine: ALTER FRAGMENT, ALTER INDEX, ALTER OPTICAL, ALTER TABLE, BEGIN WORK, COMMIT WORK, CREATE TRIGGER, DELETE, DROP INDEX, DROP OPTICAL, DROP SYNONYM, DROP TABLE, DROP TRIGGER, DROP VIEW, INSERT, RENAME COLUMN, RENAME TABLE, ROLLBACK WORK, SET CONSTRAINTS, and UPDATE.

When you use a procedure as a triggered action, the objects that it references are not checked until the procedure is executed.

Privileges to Execute Triggered Actions

If you are not the trigger owner, but the trigger owner's privileges include the WITH GRANT OPTION privilege, you inherit the owner's privileges as well as the WITH GRANT OPTION privilege for each triggered SQL statement. You have these privileges in addition to your privileges.

If the triggered action is an SPL or external routine, you must have the Execute privilege on the routine or the owner of the trigger must have the Execute privilege and the WITH GRANT OPTION privilege.

While executing the routine, you do not carry the privileges of the trigger owner; instead you receive the privileges granted with the routine, as follows:

1. Privileges for a DBA routine

   When the routine is registered with the CREATE DBA keywords and you are granted the Execute privilege on the routine, the database server automatically grants you temporary DBA privileges while the routine executes. These DBA privileges are available only when you are executing the routine.

2. Privileges for a routine without DBA restrictions

   If the routine owner has the WITH GRANT OPTION right for the necessary privileges on the underlying objects, you inherit these privilege when you are granted the Execute privilege. In this case, all the nonqualified objects that the Routine references are qualified by the name of the Routine owner.

   If the Routine owner does not have the WITH GRANT OPTION right, you have your original privileges on the underlying objects when the Routine executes.

For more information on privileges on routines, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.

Creating a Triggered Action That Anyone Can Use

To create a trigger that is executable by anyone who has the privileges to execute the triggering statement, you can ask the DBA to create a DBA-privileged procedure and grant you the Execute privilege with the WITH GRANT OPTION right. You then use the DBA-privileged procedure as the triggered action. Anyone can execute the triggered action because the DBA-privileged procedure carries the WITH GRANT OPTION right. When you activate the procedure, the database server applies privilege-checking rules for a DBA. For more information about privileges on stored procedures, see Chapter 14 of the *Informix Guide to SQL: Tutorial*.

Cascading Triggers

The database server allows triggers to cascade, meaning that the triggered actions of one trigger can activate another trigger. The maximum number of triggers in a cascading sequence 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, as the following example shows:

```
Exceeded limit on maximum number of cascaded triggers.
```

The following example illustrates a series of cascading triggers that enforce referential integrity on the **manufact**, **stock**, and **items** tables in the **stores7** database. When a manufacturer is deleted from the **manufact** table, the first trigger, **del_manu**, deletes all the items from that manufacturer from the **stock** table. Each delete in the **stock** table activates a second trigger, **del_items**, that deletes all the **items** from that manufacturer from the **items** table. Finally, each delete in the **items** table triggers the stored procedure **log_order**, which creates 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 would prohibit the triggers in this example from executing. When you use INFORMIX-Universal Server with logging, however, the triggers execute successfully because constraint checking is deferred until all the triggered actions are complete, including the actions of cascading triggers. See "Constraint Checking" for more information about how constraints are handled when triggers execute.

The database server prevents loops of cascading triggers by not allowing you to modify the triggering table in any cascading triggered action, except an UPDATE statement, which does not modify any column that the triggering UPDATE statement updated.

Constraint Checking

When you use logging, INFORMIX-Universal Server defers constraint checking on the triggering statement until after the statements in the triggered-action list execute. Universal Server effectively executes a SET statement (SET CONSTRAINTS ALL DEFERRED) before it executes the triggering statement. After the triggered action is completed, it effectively executes another SET statement (SET CONSTRAINTS *constr_name* IMMEDIATE) to check the constraints that were deferred. This action allows you to write triggers so that the triggered action can resolve any constraint violations that the triggering statement creates. For more information, see the SET statement on page 1-644.

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 triggered 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 triggered 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 triggered action to see whether the parent row exists, and if so, bypass the insert.

For a database without logging, Universal Server does *not* defer constraint checking on the triggering statement. In this case, it immediately returns an error if the triggering statement violates a constraint.

Universal Server does not allow the SET statement in a triggered action. Universal Server checks this restriction when you activate a trigger because the statement could occur inside a stored procedure.

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 triggered 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 triggered action. Then, inside the triggered 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 trigger has a BEFORE action, it applies to all columns because you cannot yet detect whether a column has changed.
- If the triggering UPDATE statement sets a column to the current value, you cannot detect the update, so the triggered action is skipped. You might want to execute the triggered action even though the value of the column has not changed.

Client/Server Environment

In an Universal Server database, the statements inside the triggered action can affect tables in external databases. 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 stores7@dbserver2:items
    SET quantity = post.qty WHERE stock_num = post.stock
    AND manu_code = post.mc)
```

If a statement from an external database server initiates the trigger, however, and the triggered action affects tables in an external database, the triggered actions fail. For example, the following combination of triggered action and triggering statement results in an error when the triggering statement executes:

```
-- Triggered action from dbserver1 to dbserver3:

CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores7@dbserver3:items
    SET quantity = post.qty WHERE stock_num = post.stock
    AND manu_code = post.mc);

-- Triggering statement from dbserver2:

UPDATE stores7@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. However, when the database does not have logging, you cannot roll back when the triggering statement fails. In this case, you are responsible for maintaining data integrity in the database.

In INFORMIX-Universal Server, if the trigger fails and the database has transactions, all triggered actions and the triggering statement are rolled back because the triggered actions are an extension of the triggering statement. The rest of the transaction, however, is not rolled back.

The row action of the triggering statement occurs before the triggered actions in the FOR EACH ROW section. If the triggered action fails for a database without logging, the application must restore the row that was changed by the triggering statement to its previous value.

When you use a stored procedure as a triggered action, if you terminate the procedure in an exception-handling section, any actions that modify data inside that section are rolled back along with the triggering statement. In the following partial example, when the exception handler traps an error, it inserts a row into the table **logtab**:

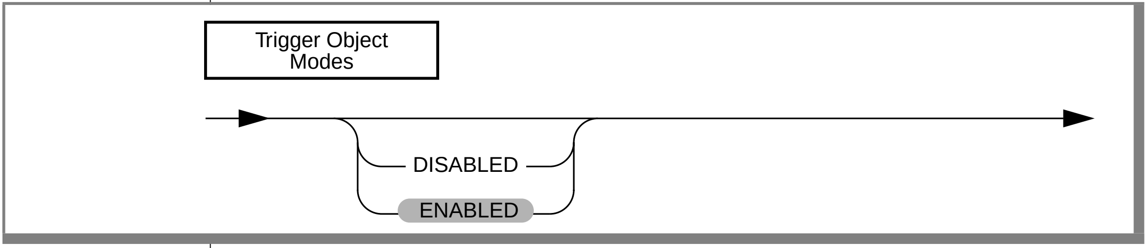
```
ON EXCEPTION IN (-201)
    INSERT INTO logtab values (errno, errstr);
    RAISE EXCEPTION -201
END EXCEPTION
```

When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this insert because it is part of the triggered actions. If the procedure is executed outside a triggered action, the insert is not rolled back.

The stored procedure that implements a triggered action cannot contain any BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. If the database has logging, you must either begin an explicit transaction before the triggering statement, or the statement itself must be an implicit transaction. In any case, another transaction-related statement cannot appear inside the stored procedure.

You can use triggers to enforce referential actions that the database server does not currently support. For any database without logging, you are responsible for maintaining data integrity when the triggering statement fails.

Trigger Object Modes



The Trigger Object Modes option allows you to create a trigger in either the enabled or disabled object mode.

You can create triggers in the following object modes.

Object Mode	Effect
disabled	When a trigger is created in disabled mode, the database server does not execute the triggered action when the trigger event (an insert, delete, or update operation) takes place. In effect, the database server ignores the trigger even though its catalog information is maintained.
enabled	When a trigger is created in enabled mode, the database server executes the triggered action when the trigger event (an insert, delete, or update operation) takes place.

Specifying Object Modes for Triggers

You must observe the following rules when you specify the object mode for a trigger in the CREATE TRIGGER statement:

- If you do not specify the disabled or enabled object modes explicitly, the default object mode is enabled.
- In contrast to unique indexes and constraints of all types, you cannot set triggers to the filtering object mode because a trigger does not impose any type of data-integrity requirement on the tables in the database.
- You can use the SET statement to switch the mode of a disabled trigger to the enabled mode. Once the trigger has been re-enabled, the database server executes the triggered action whenever the trigger event takes place. However, the re-enabled trigger does not perform retroactively. The database server does not attempt to execute the trigger for rows that were inserted, deleted, or updated after the trigger was disabled and before it was enabled; therefore, be cautious about disabling a trigger. If disabling a trigger will eventually destroy the semantic integrity of the database, do not disable the trigger in the first place.
- You cannot create a trigger on a violations table or a diagnostics table.

References

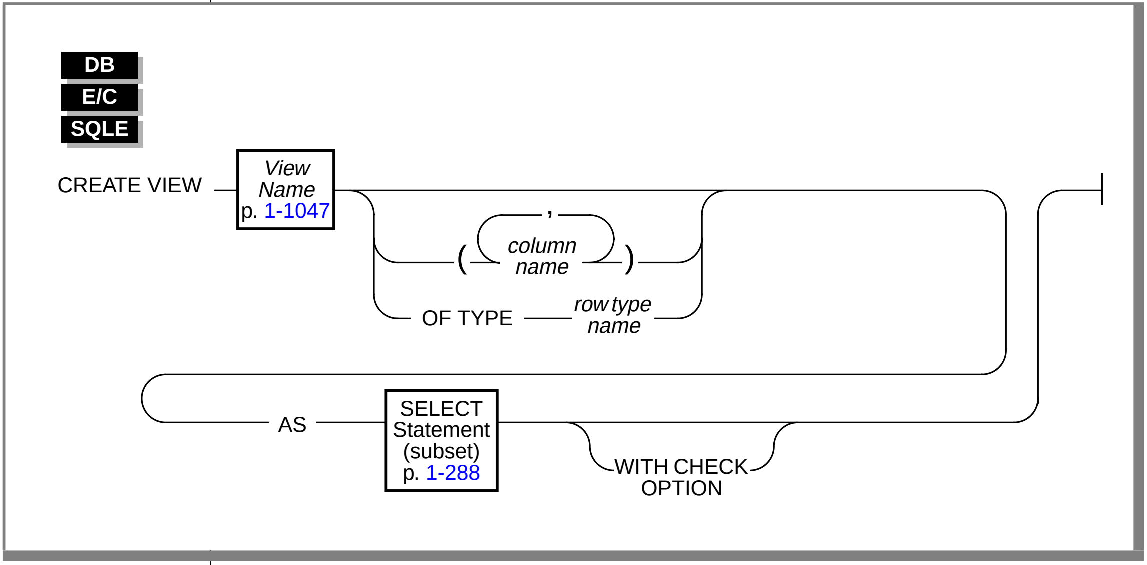
See the DROP TRIGGER, CREATE PROCEDURE, and EXECUTE PROCEDURE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about stored procedures.

CREATE VIEW

Use the CREATE VIEW statement to create a new view that is based upon existing tables and views in the database.

Syntax



Element	Purpose	Restrictions	Syntax
row type name	The name of a named row type that you use to specify the type of a typed view	You must have USAGE privileges on the named row type or be its owner or the DBA. The named row type must exist before you can assign it to a view.	Data Type, p. 1-855
column name	The name of a column in the view being created	See "Naming View Columns" on page 1-288.	Identifier, p. 1-962

Usage

You can create typed or untyped views. If you omit the OF TYPE clause, the rows in the view are considered to be 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.

You can use a view in any SQL statement where you can use a table, except the following.

ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
CREATE INDEX
CREATE TABLE
CREATE TRIGGER
DROP INDEX
DROP TABLE
DROP TRIGGER
LOCK TABLE
RECOVER TABLE
RENAME TABLE
UNLOCK TABLE

The view behaves like a table that is called *view name*. It consists of the set of rows and columns that the SELECT statement returns each time the SELECT statement is executed by using the view. The view reflects changes to the underlying tables with one exception. If a SELECT * clause defines the view, the view has only the columns in the underlying tables at the time the view is created. New columns that are subsequently added to the underlying tables with the ALTER TABLE statement do not appear in the view.

The view name must be unique; that is, a view name cannot have the same name as another database object, such as a table, synonym, or temporary table.

The view inherits the data types of the columns from the tables from which they come. Data types of virtual columns are determined from the nature of the expression.

To create a view, you must have the Select privilege on all columns from which the view is derived.

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.

You cannot create a view on a temporary table.

DB

If you create a view outside the CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Subset of a SELECT Allowed in CREATE VIEW

The SELECT statement has the form that is described on page 1-593, but in CREATE VIEW, it cannot have an ORDER BY clause, INTO TEMP clause, or UNION operator. Do not use display labels in the select list; display labels are interpreted as column names.

Naming View Columns

The number of columns that you specify in the *column name* parameter 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'
```

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. You must also provide a column name in cases where the selected columns have duplicate column names when the table prefixes are stripped. For example, when both **orders.order_num** and **items.order_num** appear in the SELECT statement, you must provide two separate column names to label them in the CREATE VIEW statement, as the following example shows:

```
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
```

If you must 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.

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 listed in Chapter 11 of the *Informix Guide to SQL: Tutorial*. See that manual for further information.

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
```

What do the WITH CHECK OPTION keywords really check and prevent? It is possible to 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). It is also possible to 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.

However, even if the view was created with the WITH CHECK OPTION keywords, 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 it did 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 derived value for a column, which was created using an arithmetical expression

In an updatable view, you can update the values in the underlying table by inserting values into the view.

Important: *You cannot update or insert rows in a remote table through views with check options.*

Examples

The following statement creates a view that is based on the **person** table. When you create a view without an OF TYPE clause, the view is referred to as an *untyped view.*

```
CREATE VIEW v1 AS SELECT *
FROM person
```

The following statement creates a typed view that is based on the table **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.

```
CREATE VIEW v2 OF TYPE person_t AS SELECT *
FROM person
```

For more information about how to create and use typed views, see Chapter 11 of the *Informix Guide to SQL: Tutorial*.

References

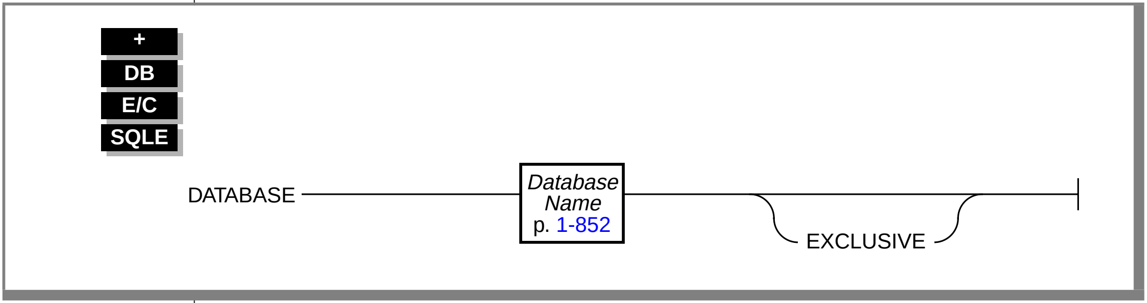
See the CREATE TABLE, DROP VIEW, GRANT, SELECT, and SET SESSION AUTHORIZATION statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of views and security in Chapter 11. Also, see the discussion of named row types in Chapter 10.

DATABASE

Use the DATABASE statement to select an accessible database as the current database.

Syntax



Usage

You can use the DATABASE statement to select any database on your database server. To select a database on another Universal Server database server, specify the name of the database server with the database name.

If you specify the name of the current database server or another database server with the database name, the database server name cannot be uppercase.

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 held by the database server, which invalidates any cursors you have declared up to that point. If the user identity was changed through a SET SESSION AUTHORIZATION statement, the original user name is restored.

The current user (or PUBLIC) must have the Connect privilege on the database specified in the DATABASE statement. The current user cannot have the same user name as an existing role in the database.

ESQL

You cannot include the DATABASE statement in a multistatement PREPARE operation.

You can determine the type of database a user selects by checking the warning flag after a DATABASE statement in the **sqlca** structure.

If the database has transactions, the second element of the **sqlwarn** structure (**sqlca.sqlwarn.sqlwarn1**) contains a W after the DATABASE statement executes. ♦

ESQL

ANSI

If the database is ANSI compliant, the third element of the **sqlwarn** structure (**sqlca.sqlwarn.sqlwarn2**) contains a W after the DATABASE statement executes. ♦

ESQL

If the database is an INFORMIX-Universal Server database, the fourth element of the **sqlwarn** structure (**sqlca.sqlwarn.sqlwarn3**) contains a W after the DATABASE statement executes.

If the database is running in secondary mode, the seventh element of the **sqlwarn** structure (**sqlca.sqlwarn.sqlwarn6**) contains a W after the DATABASE statement executes. ♦

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 execute the CLOSE DATABASE statement and then reopen the database without the EXCLUSIVE keyword.

The following statement opens the **stores7** database on the **training** database server in exclusive mode:

```
DATABASE stores7@training EXCLUSIVE
```

If another user has already opened the database, exclusive access is denied, an error is returned, and no database is opened.

References

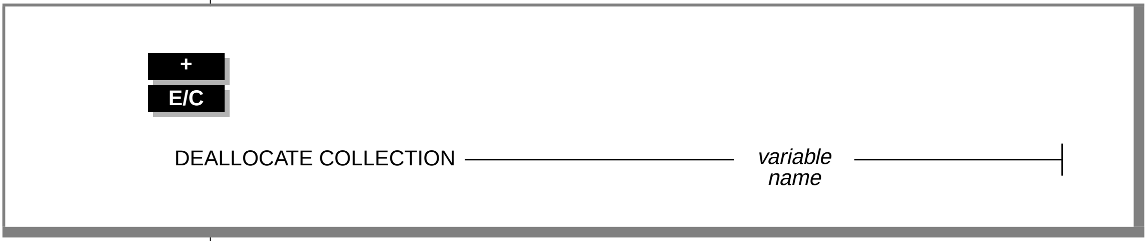
See the CLOSE DATABASE and CONNECT statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of database design in Chapter 8 and implementing the data model in Chapter 9.

DEALLOCATE COLLECTION

Use the DEALLOCATE DESCRIPTOR statement to release memory for an INFORMIX-ESQL/C **collection** variable that was previously allocated with the ALLOCATE COLLECTION statement.

Syntax



Element	Purpose	Restrictions	Syntax
variable name	Variable name that identifies a typed or untyped **collection** variable for which to deallocate memory	Variable must contain the name of an ESQL/C **collection** variable that has already been allocated.	Name must conform to language-specific rules for variable names.

Usage

The DEALLOCATE COLLECTION statement frees all the memory that is associated with the ESQL/C **collection** variable that *variable name* identifies. You must explicitly release memory resources for a **collection** variable with DEALLOCATE COLLECTION. Otherwise, deallocation does not occur automatically until the end of the program.

The following 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;
```

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 ESQL/C* ***row*** *variables, 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.

References

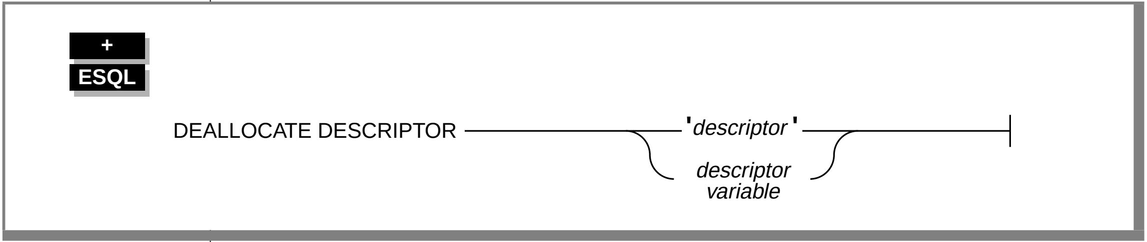
See the ALLOCATE COLLECTION and DEALLOCATE ROW statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of collection data types in Chapter 10. In the *INFORMIX-ESQL/C Programmer's Manual*, see the discussion of complex data types.

DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free a system-descriptor area that was previously allocated with the ALLOCATE DESCRIPTOR statement.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. The surrounding quotes must be single.	Quoted String, p. 1-1010
descriptor variable	Host variable name that identifies a system-descriptor area	System-descriptor area must already be allocated.	Name must conform to language-specific rules for variable names.

Usage

The DEALLOCATE DESCRIPTOR statement frees all the memory that is associated with the system-descriptor area that *descriptor* or *descriptor variable* identifies. It also frees all the item descriptors (including memory for data values in the value descriptors).

The following examples show the DEALLOCATE DESCRIPTOR statement for INFORMIX-ESQL/C. The first line shows an embedded-variable name, and the second line shows a quoted string that identifies the allocated system-descriptor area.

```
EXEC SQL deallocate descriptor :descname;

EXEC SQL deallocate descriptor 'desc1';
```

You can reuse a descriptor or descriptor variable after it is deallocated. 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.

References

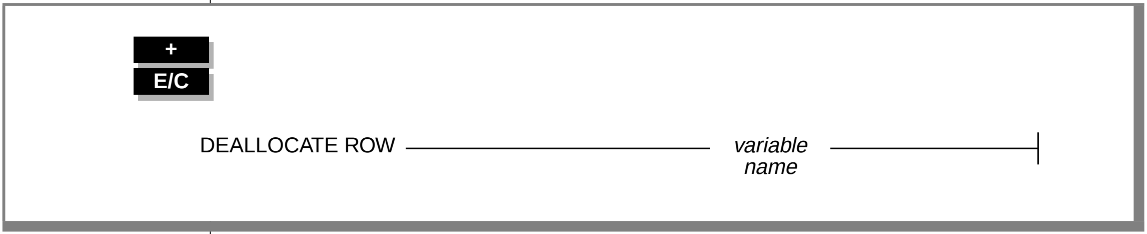
See the ALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of dynamic SQL in Chapter 5. In the *INFORMIX-ESQL/C Programmer's Manual*, see the discussion of dynamic SQL.

DEALLOCATE ROW

Use the DEALLOCATE ROW statement to release memory for an INFORMIX-ESQL/C **row** variable that was previously allocated with the ALLOCATE ROW statement.

Syntax



Element	Purpose	Restrictions	Syntax
variable name	Variable name that identifies a typed or untyped **row** variable for which to deallocate memory	Variable must contain the name of an ESQL/C **row** variable that has already been allocated.	Name must conform to language-specific rules for variable names.

Usage

The DEALLOCATE ROW statement frees all the memory that is associated with the ESQL/C **row** variable that *variable name* identifies. You must explicitly release memory resources for a **row** variable with DEALLOCATE ROW. Otherwise, deallocation does not occur automatically until the end of the program.

The following example shows how to deallocate resources for the **row** variable, **a_row**, with 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;
```

The DEALLOCATE COLLECTION statement releases resources for both typed and untyped **row** variables.

Tip: *The DEALLOCATE ROW statement deallocates memory for an ESQL/C* ***row*** *variable only. To deallocate memory for ESQL/C* ***collection*** *variables, use the DEALLOCATE COLLECTION statement.*

If you deallocate a nonexistent **row** variable or a variable that is not an ESQL/C **row** variable, an error results. Once you deallocate a **row** variable, you can use the ALLOCATE ROW to reallocate resources, and you can then reuse a **row** variable.

References

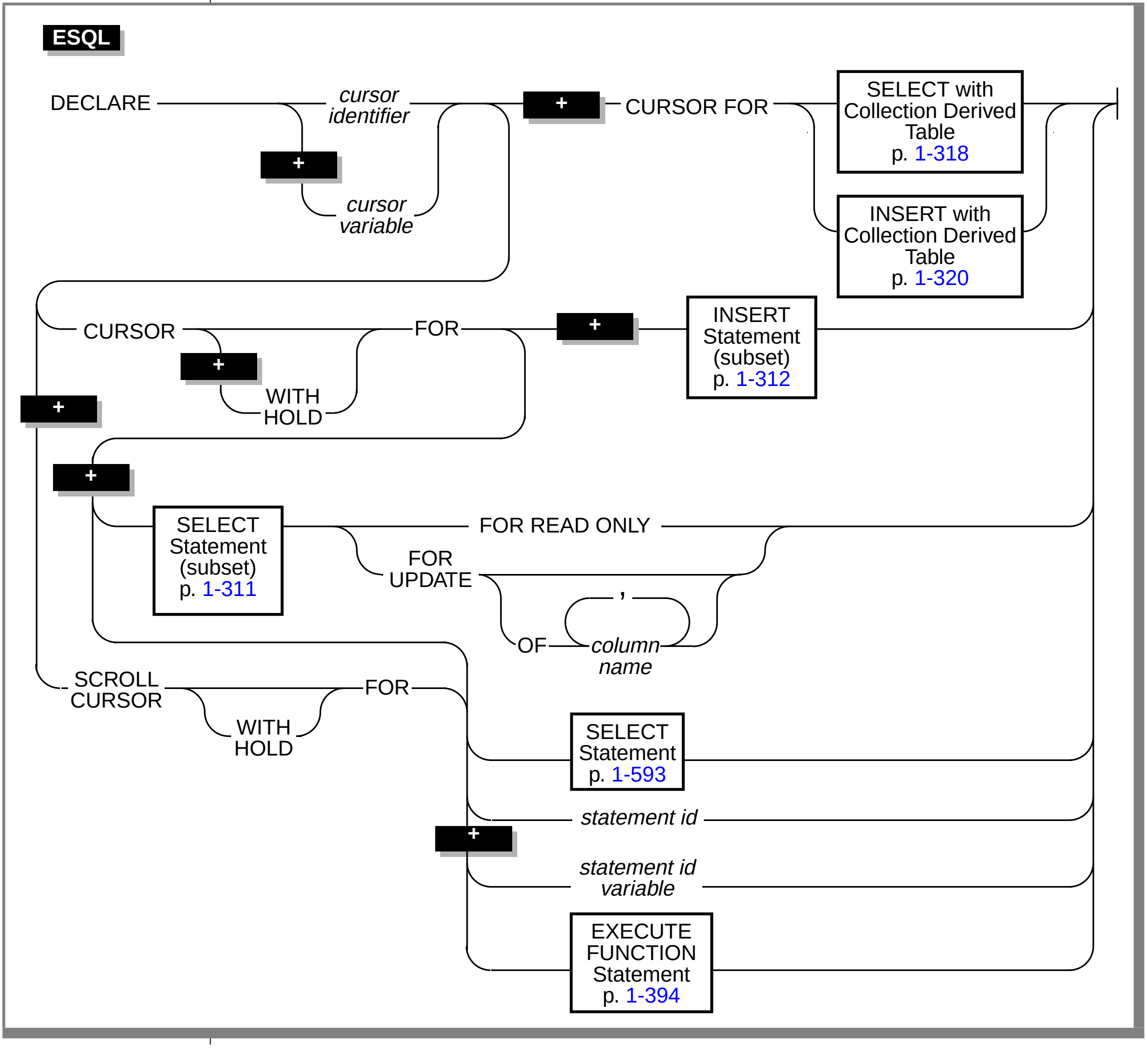
See the ALLOCATE ROW and DEALLOCATE COLLECTION statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of row types in Chapter 10. In the *INFORMIX-ESQL/C Programmer's Manual*, see the discussion of complex data types.

DECLARE

Use the DECLARE statement to define a cursor, which associates rows with a SELECT, INSERT, or EXECUTE FUNCTION statement.

Syntax


ESQL
DECLARE
cursor identifier
+
cursor variable
+
CURSOR FOR
SELECT with Collection Derived Table p. 1-318
INSERT with Collection Derived Table p. 1-320
CURSOR
+
WITH HOLD
FOR
+
INSERT Statement (subset) p. 1-312
+
+
SELECT Statement (subset) p. 1-311
FOR READ ONLY
FOR UPDATE
OF
,
column name
SCROLL CURSOR
WITH HOLD
FOR
SELECT Statement p. 1-593
statement id
+
statement id variable
EXECUTE FUNCTION Statement p. 1-394

Element	Purpose	Restrictions	Syntax
column name	A column that you can update through the cursor	The specified column must exist, but it does not have to be in the select list of the SELECT clause.	Identifier, p. 1-962
cursor id	The name that the DECLARE statement assigns to the cursor and that refers to the cursor in other statements	You cannot specify a cursor name that a previous DECLARE statement in the same program has specified.	Identifier, p. 1-962
cursor variable	An embedded variable name that holds the value of *cursor id*	Variable must be a character data type.	The name must conform to language-specific rules for variable names.
statement id	A statement identifier that is a data structure representing the text of a prepared SQL statement	The *statement id* must have already been specified in a PREPARE statement in the same program.	Identifier, p. 1-962, and PREPARE, p. 1-538
statement id variable	An embedded variable name that holds the value of *statement id*	Variable must be a character data type.	The name must conform to language-specific rules for variable names.

Usage

A *cursor* is an identifier that you associate with a group of rows. The DECLARE statement associates the cursor with one of the following database objects:

- With an SQL statement, such as SELECT, EXECUTE FUNCTION, or INSERT

  Each of these SQL statements creates a different type of cursor. For more information, see “Overview of Cursor Types” on page 1-303.

- With the statement identifier (*statement id* or *statement id variable*) of a prepared statement.

  You can prepare a SELECT, EXECUTE FUNCTION, or INSERT statement and associate the prepared statement with a cursor. For more information, see “Associating a Cursor With a Prepared Statement” on page 1-316.

- With a collection variable in an INFORMIX-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 1-317.

The DECLARE statement assigns an identifier to the cursor, specifies its uses, and directs the preprocessor to allocate storage to hold the cursor. The DECLARE statement must precede any other statement that refers to the cursor during the execution of the program.

The amount of available memory in the system limits the number of open cursors and prepared statements that you can have at one time in one process. Use FREE *statement id* or FREE *statement id variable* to release the resources that a prepared statement holds; use FREE *cursor id* or FREE *cursor variable* to release resources that a cursor holds.

A program can consist of one or more source-code files. By default, the scope of a cursor is global to a program, so a cursor declared in one file can be referenced from another file. In a multiple-file program, if you want to limit the scope of cursors to the files in which they are declared, you must preprocess all the files with the **-local** command-line option. See your SQL API product manual for more information, restrictions, and performance issues when you preprocess with the **-local** option.

A host variable used in place of the cursor name or statement identifier must be a character data type. The following ESQL/C code defines a **char** host variable called **cursname**:

```
EXEC SQL BEGIN DECLARE SECTION;
    char cursname[20];
EXEC SQL END DECLARE SECTION;
```

Other ESQL/C character data types are also valid to hold cursor names and statement identifiers.

To declare multiple cursors, use a single statement identifier. For instance, the following INFORMIX-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 identifier names and statements that use derived tables. Some error checking is performed at runtime. The following list indicates the typical checks:

- Illegal use of cursors (that is, normal cursors used as scroll cursors)
- Use of undeclared cursors
- Bad cursor 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 an identifier. For example, the code in the first example below results in a compile error. The code in the second example does not result in a compile error because it uses a host variable to hold the cursor name.

Results in error

```
EXEC SQL declare x cursor for
    select * from customer;
. . .
EXEC SQL declare x cursor for
    select * from orders;
```

Runs successfully

```
EXEC SQL declare x cursor for
    select * from customer;
. . .
stcopy("x", s);
EXEC SQL declare :s cursor for
    select * from customer;
```

Overview of Cursor Types

With the DECLARE statement, you can declare the following types of cursors:

- A *select cursor* is a cursor that is associated with a SELECT statement.
- A *function cursor* is a cursor that is associated with an EXECUTE FUNCTION statement, which executes routines that return values.
- An *insert cursor* is a cursor that is associated with an INSERT statement.

Any of these cursor types can have cursor characteristics: sequential, scroll, and hold. These characteristics determine the structure of the cursor. For more information, see "Cursor Characteristics" on page 1-313. In addition, a select or function cursor can have a cursor mode: read-only or update. For more information, see "Cursor Modes" on page 1-306.

The following table summarizes types of cursors that are available.

	Cursor Mode		**Cursor Characteristic**		
Cursor Type	**Read-Only**	**Update**	**Sequential**	**Scroll**	**Hold**
Select and Function	✓		✓		
	✓		✓		✓
		✓	✓		
		✓	✓		✓
	✓			✓	
	✓			✓	✓
Insert			✓		
			✓		✓

Tip: *A cursor can also be associated with a statement identifier, enabling you to use a cursor with INSERT, SELECT, or EXECUTE FUNCTION 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 the OPEN statement on page 1-525). For more information, see "Associating a Cursor With a Prepared Statement" on page 1-316.*

The following sections describe each of these cursor types.

Select 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:

- When 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.

- When you associate an EXECUTE FUNCTION statement with a cursor, the cursor is called a *function cursor.*

  The function cursor represents the columns or values that a user-defined function (and external function or an SPL function) returns. Function cursors behave the same as select cursors, which are enabled as update cursors.

Important: *In previous releases of Informix products, the EXECUTE PROCEDURE statement was used to execute stored procedures that returned values. For backward compatibility, you can still use EXECUTE PROCEDURE to execute stored procedures that return a value. However, Informix recommends that you execute new SPL routines that return values, called SPL functions, with the EXECUTE FUNCTION statement. For more information on how to use EXECUTE PROCEDURE with function names, see page 1-404.*

When you associate a SELECT or EXECUTE FUNCTION statement with a cursor, the statement can include an INTO clause. However, if you prepare the SELECT or EXECUTE FUNCTION 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 enables you to scan returned rows of data and to move data row by row into a set of receiving variables, as the following steps describe:

1. Use a DECLARE statement to define a cursor and associate the SELECT statement or the EXECUTE FUNCTION statement with the cursor.
2. Open the cursor with the OPEN statement. The database server processes the query until it locates or constructs the first row of the active set.
3. Retrieve successive rows of data from the cursor with the FETCH statement.
4. Close the cursor with the CLOSE statement when the active set is no longer needed.
5. Free the cursor with the FREE statement. The FREE statement releases the resources that are allocated for a select or function cursor.

Cursor Modes

You can declare a select or function cursor with one of two cursor modes:

- Read-only mode, with the FOR READ ONLY keywords
- Update mode, with the FOR UPDATE keywords

You cannot specify both the FOR UPDATE option and the FOR READ ONLY option in the same DECLARE statement because these options are mutually exclusive.

Read-Only Cursor

In a database that is *not* ANSI compliant, data in a select cursor or function cursor is read only. That is, you cannot directly update the data that is within a select or function cursor. Such a cursor is called a *read-only cursor*. To update data in a read-only cursor, you must copy the data out of the read-only cursor, perform the modifications on the copy, and then explicitly update the row with an UPDATE statement and a WHERE clause to identify the row you are updating.

ANSI

In an ANSI-compliant database, you can directly update the data that is within a select cursor because a select cursor and a function cursor are, by default, update cursors. (For more information, see "Update Cursor".) If you want a select or function cursor to be for read only, you must declare a read-only cursor with the FOR READ ONLY option of the DECLARE statement. The FOR READ ONLY keywords state explicitly that a select or function cursor cannot be used to modify data. The database server can use less stringent locking for a read-only cursor than for an update cursor.

The following example declares a read-only cursor:

```
EXEC SQL declare z_curs cursor for
    select * from customer_ansi
    for read only;
```

The SELECT statement for the cursor must conform to all of the restrictions for read-only cursors listed in "Subset of the SELECT Statement Associated with Cursors" on page 1-311. ♦

In a database that is not ANSI compliant, a select cursor and a select cursor with the FOR READ ONLY option are the same. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation. The following example declares a read-only cursor in a non-ANSI 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, you can 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;
```

Update Cursor

In a database that is *not* ANSI compliant, you cannot directly update the data that is within a select cursor or function cursor because these cursors are, by default, read-only cursors. (For more information, see "Read-Only Cursor" on page 1-306.) To update data in a select or function cursor, you must declare an *update cursor* with the FOR UPDATE option of the DECLARE statement.

The following example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
    select * from customer_notansi
    for update;
```

ANSI

In an ANSI-compliant database, a select cursor and a select cursor with the FOR UPDATE option are the same. You can use a select cursor to update or delete data as long as the cursor was not declared with the FOR READ ONLY option and it follows the restrictions on update cursors that are described in "Subset of the SELECT Statement Associated with Cursors" on page 1-311.

The following example declares an update cursor in an ANSI-compliant database:

```
EXEC SQL declare x_curs cursor for
    select * from customer_ansi;
```

If you want to make it clear in the program documentation that this cursor is an update cursor, you can specify the FOR UPDATE option as the following example shows:

```
EXEC SQL declare x_curs cursor for
    select * from customer_ansi
    for update;
```

The SELECT statement for the cursor must conform to all of the restrictions for update cursors listed in "Subset of the SELECT Statement Associated with Cursors" on page 1-311. ♦

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.

Locking with an update cursor

Use the FOR UPDATE keywords to 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 or write lock. Before the program modifies the row, the row lock is promoted to an exclusive lock.

Although it is possible to declare an update cursor with the WITH HOLD keywords, 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. (For more information on hold cursors, see page 1-315.)

If an operation involves fetching and updating a very 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. However, you must plan such an application very carefully because COMMIT WORK releases all locks, even those that are placed through a hold cursor.

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...WHERE CURRENT OF statements. The columns need not be in the select list of the SELECT clause.

The following 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, a select cursor in a database that is ANSI compliant is an update cursor. Therefore, the FOR UPDATE keywords are optional. However, 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;
```

♦

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 speed, 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 must keep a list of each row that is updated to ensure that no row is updated twice. When you use the OF keyword to specify the columns that can be updated, the database server determines whether to keep the list of updated rows. If the database server determines that the list is unnecessary, then eliminating the work of keeping the list results in a performance benefit. If you do not use the OF keyword, the database server keeps the list of updated rows, although it might be unnecessary.

This column restriction applies only to UPDATE...WHERE CURRENT OF statements. The OF *column* clause has no effect on subsequent DELETE statements that use a WHERE CURRENT OF clause. (A DELETE statement removes the contents of all columns.)

The following example contains INFORMIX-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;
```

Subset of the SELECT Statement Associated with Cursors

Not all SELECT statements can be associated with an update cursor or a read-only cursor. If the DECLARE statement includes the FOR UPDATE clause or the FOR READ ONLY clause, 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 UPDATE clause, 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.

If the DECLARE statement includes the FOR READ ONLY clause, the SELECT statement must conform to the following restrictions:

- The SELECT statement cannot have a FOR READ ONLY clause.
- The SELECT statement cannot have a FOR UPDATE clause.

For a complete description of SELECT syntax and usage, see the SELECT statement on page 1-593.

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 the following steps describe:

1. Use a DECLARE statement 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. Put 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.

An insert cursor increases processing efficiency (compared with embedding the INSERT statement directly). This process reduces communication between the program and the database server and also increases the speed of the insertions.

Cursor Characteristics

Structurally, you can declare a cursor with the following cursor characteristics:

- As a *sequential* cursor, which is the default characteristic
- As a *scroll* cursor, with the SCROLL keyword
- As a *hold* cursor, with the WITH HOLD keywords

A select or function cursor can be either a sequential or scroll cursor. An insert cursor can only be a sequential cursor. Select, function, and insert cursors can optionally be hold cursors. The following sections explain these structural characteristics.

Sequential Cursor

If you use only the CURSOR keyword in a DECLARE statement, 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 INFORMIX-ESQL/C 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 1-312.) The following example contains 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);
```

Scroll Cursor

When you specify the SCROLL keyword in a DECLARE statement, you create a *scroll cursor*, which can fetch rows of the active set in any sequence. The following example creates a scroll cursor for a SELECT:

```
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.

To implement a scroll cursor, the database server creates a temporary table to hold the active set. With the active set retained as a table, you can fetch the first, last, or any intermediate rows as well as fetch rows repeatedly without having to close and reopen the cursor. See the FETCH statement on page 1-408 for a discussion of these abilities.

The database server retains the active set for a scroll cursor in a temporary table until the cursor is closed. On a multiuser system, the rows in the tables from which the active-set rows were derived might change after a copy is made in the temporary table. (For information about temporary tables, see the *INFORMIX-Universal Server Administrator's Guide*.) 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 the SET ISOLATION statement on page 1-719 and the LOCK TABLE statement on page 1-522.)

Hold Cursor

If you use the WITH HOLD keywords in a DECLARE statement, you 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.

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;
```

When you associate a hold cursor with an insert cursor, 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 trigger an automatic write to the database, you can execute a COMMIT WORK statement to flush the row buffer. If you use a hold cursor, the COMMIT WORK statement 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.

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.

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 statement or EXECUTE FUNCTION statement that returns values could produce more than one row of data, the prepared statement must be associated with a cursor. (See the PREPARE statement on page 1-538 for more information about preparing SQL statements.)

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. You declare a cursor for the statement text by associating a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT or EXECUTE FUNCTION 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 INFORMIX-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 use a prepared SELECT statement to modify data, add a FOR UPDATE clause to the statement text that you wish to prepare, as the following INFORMIX-ESQL/C example shows:

```
EXEC SQL prepare sel_1 from 'select * from customer for update';
EXEC SQL declare sel_curs cursor for sel_1;
```

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.

Tip: *To access only one element of a collection variable, you do not need to declare a cursor. For information on how to select a single element, see "Selecting From a Collection Variable" on page 1-610***.** *For information on how to insert a single element, see "Inserting Into a Collection Variable" on page 1-506.*

You can declare the following types of cursors for a collection variable:

- A select cursor for a **collection** variable

  Include the Collection Derived Table clause with the SELECT statement that you associate with the cursor.
- An insert cursor for a **collection** variable

  Include the Collection Derived Table clause with the INSERT statement that you associate with the cursor.

The Collection Derived Table clause identifies the **collection** variable for which to declare the cursor. For more information on the Collection Derived Table clause, see page 1-827.

A Select Cursor for a Collection Variable

To declare a select cursor for a **collection** variable, include the Collection Derived Table clause with the SELECT statement that you associate with the cursor. A select cursor allows you to select one or more elements from the **collection** variable. The DECLARE for this select cursor has the following restrictions:

- The select cursor is an update cursor.

  The DECLARE statement cannot include the FOR READ ONLY clause that specifies the read-only cursor mode.
- The select cursor must be a sequential cursor.

  The DECLARE statement cannot specify the SCROLL or WITH HOLD cursor characteristics.

The SELECT statement that you associate with the cursor also has some restrictions:

- The SELECT statement cannot include the following clauses and options: WHERE, GROUP BY, ORDER BY, HAVING, INTO TEMP, and WITH REOPTIMIZATION.
- The select list of the SELECT cannot contain expressions.
- The select list must be an asterisk (*) if the collection contains elements of opaque, distinct, built-in, or other collection data types.
- Column names in the select list must be simple column names.

  These columns cannot use the following syntax:

  ```
  database@server:table.column
  ```

When you declare a select cursor for a **collection** variable, the Collection Derived Table clause of the SELECT statement *must* contain the name of the collection variable. You cannot specify an input parameter (the question-mark (?) symbol) for the collection variable. For example, the following 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);
```

To select the element(s) from the **collection** variable, use the FETCH statement with the INTO clause. For more information, see “Fetching From a Collection Cursor” on page 1-419.

If you want to modify the 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. For more information, see the DELETE and UPDATE statements in this manual.

A collection cursor that includes a SELECT statement with the Collection Derived Table clause allows you to access the elements in a **collection** variable. To select elements, follow these steps:

1. Create a client **collection** variable in your ESQL/C program.
2. Declare the collection cursor for the SELECT statement with the DECLARE statement and open this cursor with the OPEN statement.
3. Fetch the element(s) from the collection cursor with the FETCH statement and the INTO clause.
4. 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, you can 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).
5. Close the collection cursor with the CLOSE statement.

For a code example that uses a collection cursor for a SELECT statement, see "Fetching From a Collection Cursor" on page 1-419. For more information on how to use ESQL/C **collection** variables, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*.

An Insert Cursor For a Collection Variable

To declare an insert cursor for a **collection** variable, include the Collection Derived Table clause with the INSERT statement that you associate with the cursor. An insert cursor allows you to insert one or more elements in the collection. The insert cursor must be a sequential cursor; the DECLARE statement cannot specify the WITH HOLD cursor characteristic.

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. For example, the following DECLARE statement 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 element(s) into the collection variable, use the PUT statement with the FROM clause. For more information, see "Inserting into a Collection Cursor" on page 1-560.

A collection cursor that includes an INSERT statement with the Collection Derived Table clause allows you to insert many elements into a **collection** variable. To insert elements, follow these steps:

1. Create a client **collection** variable in your ESQL/C program.
2. Declare the collection cursor for the INSERT statement with the DECLARE statement and open the cursor with the OPEN statement.
3. Put the element(s) into the collection cursor with the PUT statement and the FROM clause.
4. Once the collection variable contains all the elements, you then 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).
5. Close the collection cursor with the CLOSE statement.

For a code example that uses a collection cursor for an INSERT statement, see "Inserting into a Collection Cursor" on page 1-560. For more information on how to use ESQL/C **collection** variables, see the discussion on complex data types *INFORMIX-ESQL/C Programmer's Manual*.

Using Cursors within 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 ANSI-compliant databases, transactions are always in effect. ♦

The database server enforces the following guidelines for insert and update cursors. These guidelines 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 hold or update cursors outside a transaction.

The following example produces an error when the database server tries to execute the UPDATE statement:

Results in error

```
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 fetch q_curs into :cust_rec; /* fetch before begin */
EXEC SQL begin work;
EXEC SQL update customer set lname = 'Smith'
    where current of q_curs;
/* error here */
EXEC SQL commit work;
```

The following example does not produce an error when the database server tries to execute the UPDATE statement:

Runs successfully

```
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 hold.

References

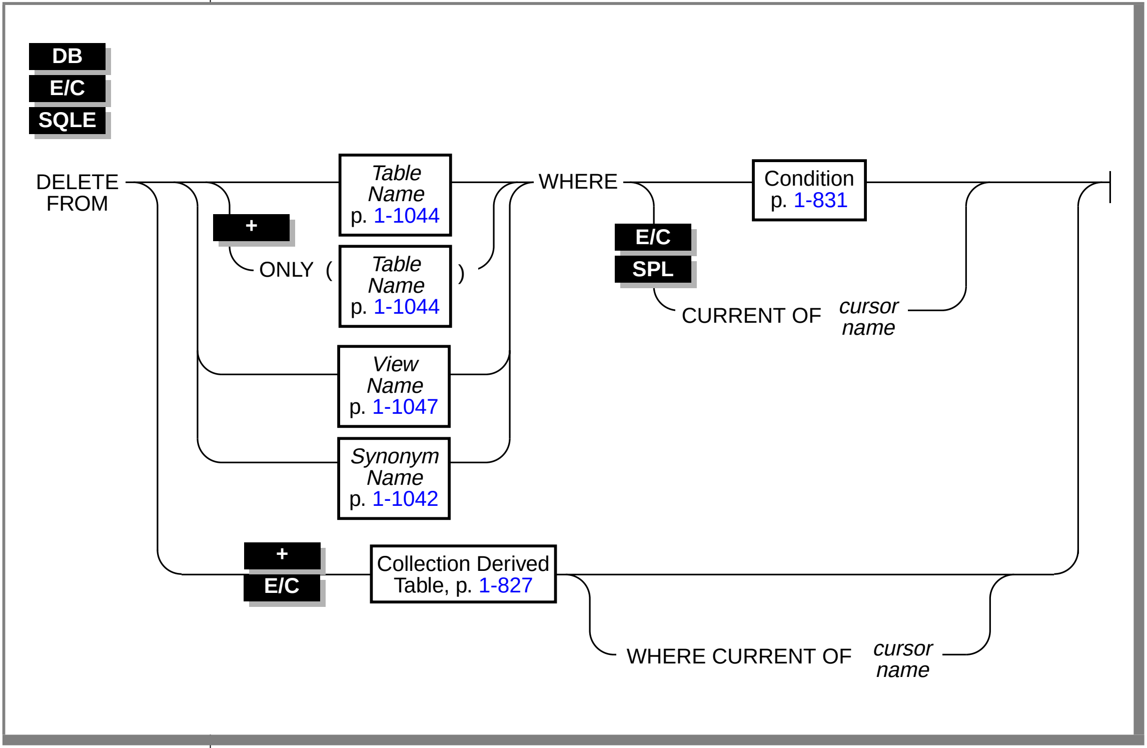
See the CLOSE, DELETE, EXECUTE FUNCTION, FETCH, FREE, INSERT, OPEN, PREPARE, PUT, SELECT, and UPDATE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification in Chapter 5 and Chapter 6, respectively.

DELETE

Use the DELETE statement to delete one or more rows from a table, or one or more elements in an SPL or INFORMIX-ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
cursor name	The name of the cursor whose current row or current collection element will be deleted	The cursor must have been previously declared in an SPL FOREACH statement or a DECLARE statement with a FOR UPDATE clause.	Identifier, p. 1-962

Usage

Use the DELETE statement to remove either of the following types of objects:

- A row in a table: a single row, a group of rows, all rows in a table, or rows from multiple tables in a table hierarchy

E/C

SPL

- An element in a collection variable ♦

For information on how to delete an element from a collection variable, see “Deleting from a Collection Variable” on page 1-330. The other sections of this DELETE statement describe how to remove a row in a table.

If you use the DELETE statement without a WHERE clause, all the rows in the table are deleted.

If you use the DELETE statement to remove rows of a supertable, rows from both the supertable and its subtables can be deleted. To delete rows from the supertable only, you must use the ONLY keyword prior to the table name, as the following example shows:

```
DELETE FROM ONLY(super_tab)
WHERE name = "johnson"
```

Warning: *If you use the DELETE statement on a supertable without the ONLY keyword and without a WHERE clause, all rows of the supertable and its subtables are deleted.*

If you use the DELETE statement outside a transaction in a database that uses transactions, each DELETE statement that you execute is treated as a single transaction.

Each row affected by a DELETE statement within a transaction is locked for the duration of the transaction; therefore, a single DELETE statement that affects a large number of rows locks the rows until the entire operation is complete. If the number of rows affected is very large, 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 entire table before you execute the statement.

If you specify a view name, the view must be updatable. See “Updating Through Views” on page 1-290 for an explanation of an updatable view.

DB

If you omit the WHERE clause while you are working within 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 the DELETE statement within a command file. ♦

ANSI

Statements are always within an implicit transaction in an ANSI-compliant database; therefore, you cannot have a DELETE statement outside a transaction. ♦

Deleting Rows That Contain Opaque Data Types

Some opaque data types require special processing when they are deleted. A 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 very large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **destroy()**. When you use the DELETE statement to delete a row that contains one of these opaque types, the database server automatically invokes the **destroy()** function for the type. The **destroy()** support function can decide how remove the data, regardless of where it is stored. For more information on the **destroy()** support function, see the *Extending INFORMIX-Universal Server: Data Types* manual.

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
```

Using Cascading Deletes

Use the ON DELETE CASCADE option of the REFERENCES clause on either the CREATE TABLE or ALTER TABLE statement to specify that you want deletes to cascade from one table to another. For example, 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 referred through the foreign keys.

If a cascading delete is performed without a WHERE clause, all rows in the parent table (and subsequently, the affected child tables) are deleted.

WHERE Clause

Use the WHERE clause to specify one or more rows that you want to delete. The WHERE conditions are the same as the conditions in the SELECT statement. For example, the following 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. ♦

E/C

Deleting and the WHERE Clause

If you delete from a table in an ANSI-compliant database with a DELETE that contains a WHERE clause and no rows are found, that 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 ... is a part of a multistatement prepare and the database server returns no rows. ♦

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 a DELETE statement. The **SQLSTATE** code is '00000' and the **SQLCODE** code is zero (0). However, if the DELETE ... WHERE ... is a part of a multistatement prepare, and no rows are returned, the database server does issue a warning. It sets **SQLSTATE** to '02000' and **SQLCODE** value to 100.

For additional information about the **SQLSTATE** code, see the GET DIAGNOSTICS statement in this manual. For information about the **SQLCODE** code, see the description of the **sqlca** structure in the *Informix Guide to SQL: Tutorial*.

WHERE CURRENT OF Clause

ESQL

SPL

You can use the WHERE CURRENT OF clause to delete either of the following objects:

- The current row of the active set of a cursor
- The current element of a collection cursor (INFORMIX-ESQL/C only)

You access both of these objects with an update cursor. An update cursor is a sequential cursor that is associated with a SELECT statement but can modify and delete the contents of the cursor. For more information on the update cursor, see page 1-307. ♦

ESQL

To use the WHERE CURRENT OF clause, you must have previously used the DECLARE statement with the FOR UPDATE clause to define the *cursor name* for the update cursor. (See the DECLARE statement on page 1-300.) ♦

SPL

Before you can use the WHERE CURRENT OF clause, you must declare a cursor with the FOREACH statement. (See the FOREACH statement on page 2-27.)♦

ANSI

All select cursors are potentially update cursors in ANSI-compliant databases. You can use the WHERE CURRENT OF clause with any select cursor. ♦

Deleting the Current Row

ESQL

SPL

When you specify a table or view name in the FROM clause of the SELECT, the DECLARE statement defines a cursor that populates an active set with the rows of the specified tables or views. The DELETE....WHERE CURRENT OF statement deletes the current row of the active set of a cursor. When you use the WHERE CURRENT OF 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. ♦

Deleting a Collection Element

ESQL

SPL

You declare a collection cursor when you associate a cursor with SELECT statement that includes a Collection Derived Table clause. You use one of the following statements to declare a collection cursor:

- In an ESQL/C program, use the DECLARE statement.

  For more information, see "Associating a Cursor With a Collection Variable" on page 1-317 in the DECLARE statement.
- In an SPL routine, use the FOREACH statement.

  For more information, see the FOREACH statement on page 2-27.

A collection cursor is an update cursor by default. However, you can optionally specify the FOR UPDATE clause with the SELECT statement. With an update cursor, you can use the DELETE...WHERE CURRENT OF statement to delete the current element of a collection cursor. For more information, see "Deleting from a Collection Variable" on page 1-330.

Important: *You can only declare a select cursor on a collection variable. Neither INFORMIX-ESQL/C nor SPL supports cursors on row variables. For more information, see "Updating a Row Variable" on page 1-798.*♦

E/C

SPL

Deleting from a Collection Variable

The DELETE statement with the Collection Derived Table clause allows you to delete elements from a collection variable. The Collection Derived Table clause identifies the collection variable in which to delete the elements. For more information, see "Collection Derived Table" on page 1-827.

E/C

In an INFORMIX-ESQL/C program, declare a host variable of type **collection** for a collection variable. This **collection** variable can be typed or untyped. ♦

SPL

In an SPL routine, declare a variable of type COLLECTION, LIST, MULTISET, or SET for a collection variable. This collection variable can be typed or untyped. ♦

To delete elements, follow these steps:

1. Create a collection variable in your SPL routine or ESQL/C program.
2. Optionally, select a collection column into the collection variable with the SELECT statement (without the Collection Derived Table clause).
3. Delete elements of the collection variable with the DELETE statement and the Collection Derived Table Clause.
4. After the collection variable contains the correct elements, use the INSERT or UPDATE statement on a table name to save the collection variable in the collection column (SET, MULTISET, or LIST).

The DELETE statement and the Collection Derived Table clause allow you to perform the following operations on a **collection** variable:

- Delete a *particular* element in the collection.

  You must declare an update cursor for the collection variable and use DELETE with the WHERE CURRENT OF clause. For more information on how to use an update cursor with ESQL/C, see the DECLARE statement on page 1-300. For more information on how to use an update cursor with SPL, see "FOREACH" on page 2-27.

  The application or SPL routine must position the update cursor on the element to be deleted and then use DELETE...WHERE CURRENT OF to delete this value. For more information on the WHERE CURRENT OF clause of DELETE, see page 1-328.

- Delete *all* elements in the collection.

  Use the DELETE statement (without the WHERE CURRENT OF clause). No cursor is required to delete all elements of a collection.

  E/C

  For example, the following DELETE statement removes all elements in the **a_list** ESQL/C collection variable:

  ```
  EXEC SQL delete from table(:a_list);
  ```

  ♦

  SPL

  You could also use the following statements in an SPL routine:

  ```
  DEFINE a COLLECTION;
  DELETE FROM TABLE (a);
  ```

  ♦

A DELETE of an element or elements in a **collection** variable cannot include a WHERE clause.

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 collection column with one of the following SQL statements:

- To update the collection column in the table with the collection variable, use an UPDATE statement on a table or view name and specify the collection variable in the SET clause.

  For more information, see "Updating Collection Columns" on page 1-786 in the UPDATE statement.

- To insert a collection in a column, use the INSERT statement on a table or view name and specify the collection variable in the VALUES clause.

  For more information, see "Inserting Values into Collection Columns" on page 1-501 in the INSERT statement.

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 of the database. Without this UPDATE statement, the collection column never has element 4 deleted.

For information on how to use **collection** host variables in an ESQL/C program, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

SPL

You can also delete the element with the value 4 from the set {1,8,4,5,2} with an SPL routine, as the following example shows.

```
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 defines two variables, **a** and **b**, each to hold a SET of SMALLINT values. The first SELECT statement selects a SET column from one row of **table1** into **b**. Then, the routine declares a cursor that selects one element at a time from **b** into **a**. When the cursor is positioned on the element with the value 4, the DELETE statement deletes that element from **b**. Last, the UPDATE statement updates the row of **table1** with the new collection that is stored in **b**.

For information on how to use collection variables in an SPL routine, see Chapter 14 of the *Informix Guide to SQL: Tutorial*. ♦

E/C

SPL

Deleting a Row Variable

The DELETE statement does not support a row variable in the Collection Derived Table clause. A row variable must have a value for each field. For more information, see "Updating a Row Variable" on page 1-798. ♦

References

See the DECLARE, INSERT, OPEN, and SELECT statements in Chapter 1 of this manual. See the FOREACH statement in Chapter 2 of this manual.

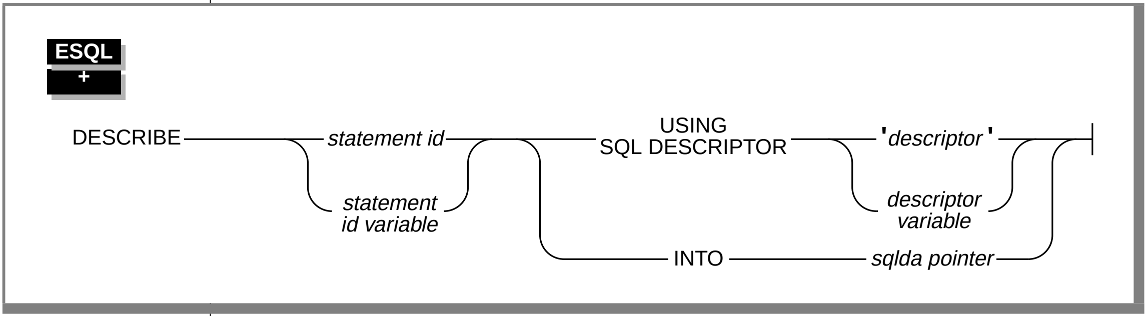
In the *Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification in Chapter 5 and Chapter 6, respectively, and the discussion of stored routines in Chapter 14. In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the DELETE statement.

For information on how to access row and collections with ESQL/C host variables, see the chapter on complex data types in the *INFORMIX-ESQL/C Programmer's Manual*.

DESCRIBE

Use the DESCRIBE statement to obtain information about a prepared statement before you execute it. The information can be stored in a system-descriptor area or in an **sqlda** structure.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	A quoted string that identifies a system-descriptor area to which values are assigned	The system-descriptor area must have been previously allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-1010
descriptor variable	A host variable that holds the value of *descriptor*	The same restrictions apply to *descriptor variable* as apply to *descriptor.*	Variable name must conform to language-specific rules for variable names.
sqlda pointer	A pointer to an **sqlda** structure	You cannot begin an **sqlda** pointer with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	See the discussion of **sqlda** structure in the *INFORMIX-ESQL/C Programmer's Manual*.

(1 of 2)

Element	Purpose	Restrictions	Syntax
statement id	The statement identifier for a prepared SQL statement	The statement identifier must be defined in a previous PREPARE statement.	PREPARE, p. 1-538
statement id variable	A host variable that contains the value of *statement id*	The statement identifier must be defined in a previous PREPARE statement. The variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The DESCRIBE statement allows you to determine, at runtime, the following information about a prepared statement:

- The DESCRIBE statement returns the prepared statement type.
- The DESCRIBE statement can determine whether an UPDATE or DELETE statement contains a WHERE clause.
- For a SELECT, EXECUTE FUNCTION, or INSERT 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.

With this information, you can write code to allocate memory to hold retrieved values and display or process them after they are fetched.

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** (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. See the discussion on exception handling in the *INFORMIX-ESQL/C Programmer's Manual* for more information about possible **SQLCODE** values after a DESCRIBE statement.

You can test the number against the constant names that are defined. In INFORMIX-ESQL/C, the constant names are defined in the **sqlstype.h** header file. A printed list of the possible values and their constant names appears in the *INFORMIX-ESQL/C Programmer's Manual*.

The DESCRIBE statement uses the **SQLCODE** field differently than 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 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 following **sqlca** variable to W.

Product	Field Name
ESQL/C	sqlca.sqlwarn.sqlwarn4

Without a WHERE clause, the update or delete action is applied to the entire table. Check this variable to avoid unintended global changes to your table.

Describing SELECT, EXECUTE FUNCTION, or INSERT

If the prepared statement text includes a SELECT statement without an INTO TEMP clause, an EXECUTE FUNCTION statement, or an INSERT statement, the DESCRIBE statement also returns a description of each column or expression that is included in the SELECT, EXECUTE FUNCTION, or INSERT list. You can store these descriptions in one of the following dynamic-management structures:

- A system-descriptor area

  For more information, see "USING SQL DESCRIPTOR Clause".
- An **sqlda** structure

  For more information, see "INTO sqlda pointer Clause" on page 1-340.

These dynamic-management structures provide 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

USING SQL DESCRIPTOR Clause

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 in a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values.

X/O

You can also use an **sqlda** structure to dynamically supply parameters. However, a system-descriptor area conforms to the X/Open standards. ♦

The USING SQL DESCRIPTOR clause lets you store the description of a SELECT, INSERT, or EXECUTE FUNCTION list in a system-descriptor area that an ALLOCATE DESCRIPTOR statement creates. You can obtain information about the resulting columns of a prepared statement through a system-descriptor area.

The following example shows the use of a system-descriptor area in a DESCRIBE statement. In the first example system-descriptor area is a quoted string; in the second example, it is a host 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;
```

The DESCRIBE...USING SQL DESCRIPTOR statement performs the following tasks on a system-descriptor area:

- It sets the COUNT field in the system-descriptor area to the number of values in the SELECT, EXECUTE FUNCTION, or INSERT list. If COUNT is greater than the number of item descriptors (*occurrences*) in the system-descriptor area, the system returns an error.
- It sets the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields in the item descriptor.

  If the column has an opaque data type, DESCRIBE...USING SQL DESCRIPTOR sets the EXTYPEID, EXTYPENAME, EXTYPELENGTH, EXTYPEOWNERLENGTH, and EXTYPEOWNERNAME fields of the item descriptor.
- It allocates memory for the DATA field in 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 can modify the system-descriptor-area information with the SET DESCRIPTOR statement. You must modify the system-descriptor area 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 the data is fetched.

You can use the system-descriptor area in statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT.

For further information, refer to the discussion of the system-descriptor area in the *INFORMIX-ESQL/C Programmer's Manual*.

INTO sqlda pointer Clause

If the prepared statement contains parameters for which the number of parameters or their data types is to be supplied at runtime, you can describe these input values in an **sqlda** structure. An **sqlda** structure describes the data type and memory location of one or more values.

The INTO *sqlda pointer* clause lets you 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 for a SELECT, INSERT, or EXECUTE FUNCTION list.

The DESCRIBE statement sets the **sqlda.sqld** field to the number of values in the SELECT, INSERT, or EXECUTE FUNCTION list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the SELECT, INSERT, or EXECUTE FUNCTION 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...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. However, the application program must designate the storage area of the **sqlda.sqlvar.sqldata** fields.

See the *INFORMIX-ESQL/C Programmer's Manual* for further information on the **sqlda** structure.

Describing a Collection Variable

The DESCRIBE statement can provide this 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 following ESQL/C code fragment shows how to dynamically select 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';
```

References

See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for further information about using dynamic management statements.

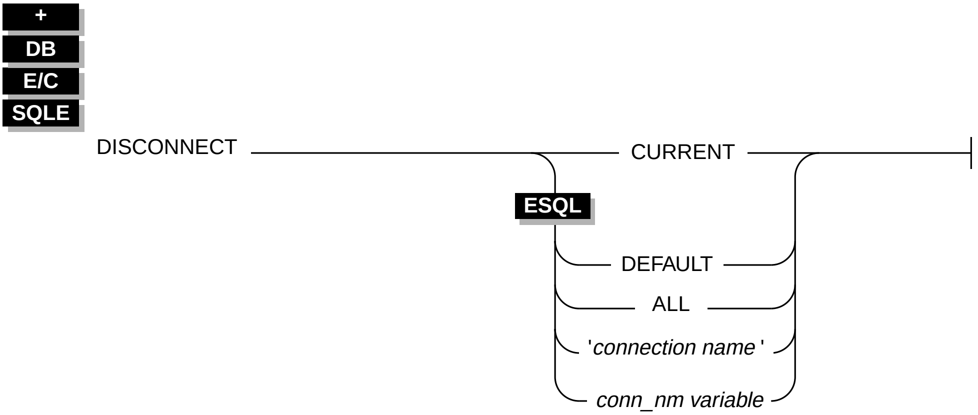
In the *Informix Guide to SQL: Tutorial*, see the discussion of the DESCRIBE statement in Chapter 5.

For further information about how to use a system-descriptor area or an **sqlda** pointer with a FETCH or an INSERT statement, refer to the *INFORMIX-ESQL/C Programmer's Manual*.

DISCONNECT

The DISCONNECT statement terminates a connection between an application and a database server.

Syntax



Element	Purpose	Restrictions	Syntax
connection name	Quoted string that identifies a connection to be terminated	Specified connection name must match a *connection name* assigned by the CONNECT statement.	Quoted String, p. 1-1010
conn_nm variable	Host variable that holds the value of *connection name*	Variable must be a fixed-length character data type. Specified connection name must match a *connection name* assigned by the CONNECT statement.	Variable name must conform to language-specific rules for variable names.

Usage

The DISCONNECT statement lets you terminate 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, that connection to the database server is terminated by the DISCONNECT statement.

You cannot use the PREPARE statement for the DISCONNECT statement.

ESQL

If you disconnect a specific connection using *connection name* or *conn_nm variable*, DISCONNECT generates an error if the specified connection is not a current or dormant connection.

A DISCONNECT statement that does not terminate the current connection does not change the context of the current environment (the connection context). ♦

DEFAULT Option

Use the DEFAULT option to identify the default connection for a DISCONNECT 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 made using the DATABASE, CREATE DATABASE, or START DATABASE statements)

You can use DISCONNECT to disconnect the default connection. See "DEFAULT Option" on page 1-100 and "Implicit Connection with DATABASE Statements" on page 1-101 for more information.

If the DATABASE statement does not specify a database server, as shown in the following example, the default connection is made to the default database server:

```
EXEC SQL database 'stores7';
.
.
.
EXEC SQL disconnect default;
```

If the DATABASE statement specifies a database server, as shown in the following example, the default connection is made to that database server:

```
EXEC SQL database 'stores7@mydbsrvr';
.
.
.
EXEC SQL disconnect default;
```

In either case, the DEFAULT option of DISCONNECT disconnects this default connection. See "DEFAULT Option" on page 1-100 and "Implicit Connection with DATABASE Statements" on page 1-101 for more information about the default database server and implicit connections.

CURRENT Keyword

Use the CURRENT keyword with the DISCONNECT statement as a shorthand form of identifying the current connection. The CURRENT keyword replaces the current connection name. For example, the DISCONNECT statement in the following excerpt terminates the current connection to the database server **mydbsrvr**:

```
CONNECT TO 'stores7@mydbsrvr'
.
.
.
DISCONNECT CURRENT
```

When a Transaction is Active

When a transaction is active, the DISCONNECT statement generates an error. The transaction remains active, and the application must explicitly commit it or roll it back. If an application terminates without issuing a DISCONNECT statement (because of a system crash or program error, for example), active transactions are rolled back.

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.

However, once a connection becomes dormant, any other thread can disconnect this connection unless an ongoing transaction is associated with the dormant connection (the connection 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.

See the SET CONNECTION statement on page 1-682 for an explanation of connections in a thread-safe ESQL/C application.

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
```

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.

References

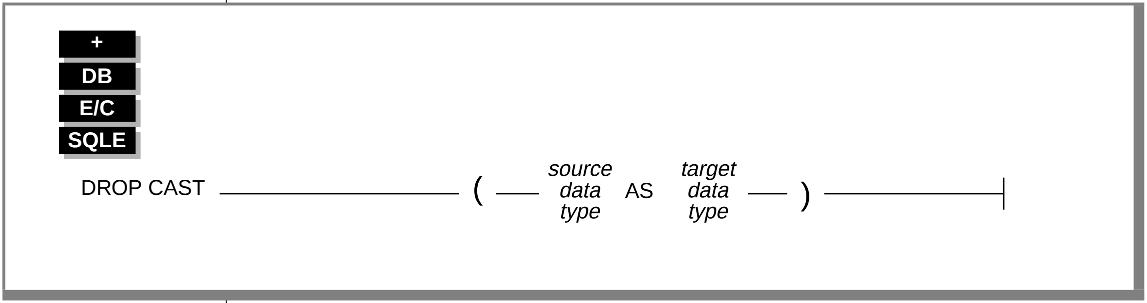
See the CONNECT, SET CONNECTION, and DATABASE statements in this manual.

For information on multithreaded applications, see the *INFORMIX-ESQL/C Programmer's Manual*.

DROP CAST

Use the DROP CAST statement to remove a previously defined cast from the database.

Syntax



Element	Purpose	Restrictions	Syntax
source data type	The data type on which the cast operates	The type must exist at the time the cast is dropped.	Data Type, p. 1-855
target data type	The data type that results when the cast is invoked	The type must exist at the time the cast is dropped.	Data Type, p. 1-855

Usage

You must be the owner of the cast or have the DBA privilege to use the DROP CAST statement.

What Happens When You Drop a Cast

When you drop a cast, the cast definition is removed from the database. Once you drop a cast, it cannot be invoked either explicitly or implicitly. Dropping a cast has no effect on the function associated with the cast. Use the DROP FUNCTION statement to remove a function from the database.

Warning: *Do not drop the system-defined casts, which are owned by user* ***informix****. The database server uses system-defined casts for automatic conversions between built-in data types.*

A cast that is defined on a particular data type can also be used on any distinct types created from that type. When you drop the cast, you can no longer invoke it for the distinct types. 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.

References

See the CREATE CAST statement in this manual for information about creating a cast.

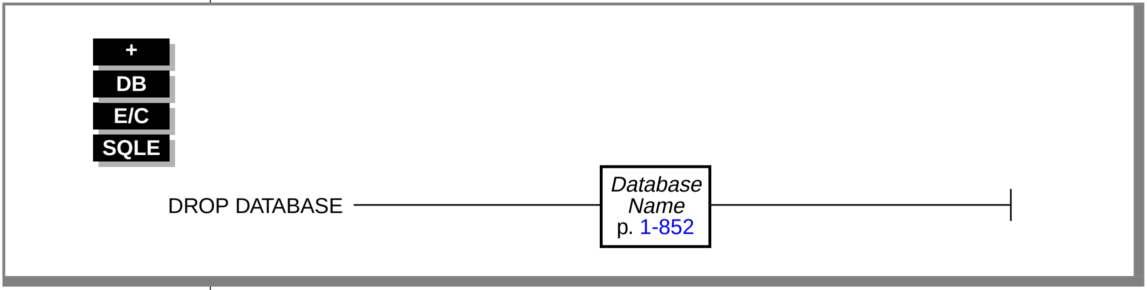
See the DROP FUNCTION statement in this manual for information about how to remove a function that is used to implement a cast.

See the Data Types segment in this manual and Chapter 3, “Environment Variables” in the *Informix Guide to SQL: Reference* for information about data types.

DROP DATABASE

Use the DROP DATABASE statement to delete an entire database, including all system catalog tables, indexes, and data.

Syntax



Usage

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 database users must first execute the CLOSE DATABASE statement.

The DROP DATABASE statement cannot appear in a multistatement PREPARE statement.

The following statement drops the **stores7** database:

```
DROP DATABASE stores7
```

When you drop a database with transactions, the transaction-log file that is associated with the database is removed.

The DROP DATABASE statement does not remove the database directory if it includes any files other than those created for database tables and their indexes.

You can specify the full pathname of the database in quotes, as the following example shows:

```
DROP DATABASE '/u/training/stores7'
```

You cannot use a ROLLBACK WORK statement to undo a DROP DATABASE statement. If you roll back a transaction that contains a DROP DATABASE statement, the database is not re-created, and you do not receive an error message.

DB

Use this statement with caution. DB-Access does not prompt you to verify that you want to delete the entire database. ♦

ESQL

You can use a simple database name in a program or host variable, or you can use the full database server and database name. See "Database Name" on page 1-852 for more information. ♦

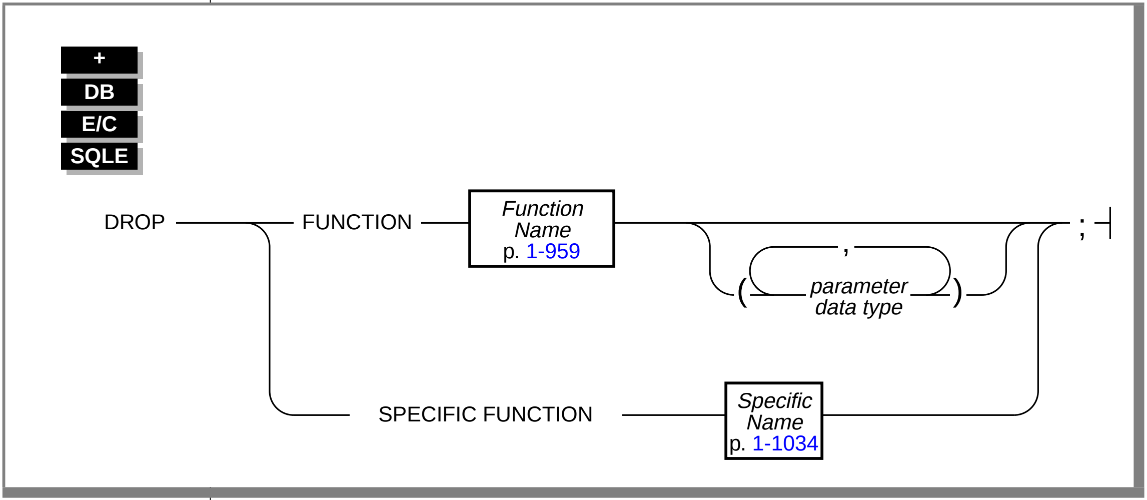
References

See the CREATE DATABASE and CLOSE DATABASE statements in this manual.

DROP FUNCTION

Use the DROP FUNCTION statement to remove an external function or anSPL function from the database.

Syntax



Element	Purpose	Restrictions	Syntax
parameter data type	The data type of the parameter	The data type must be the data type (or list of data types) specified in the CREATE FUNCTION statement when the function was created.	Identifier, p. 1-962

Usage

A function is a user-defined routine that returns one or more values. In INFORMIX-Universal Server, you can write functions in Stored Procedure Language (SPL) or in an external language, such as C.

Because you can overload routines in INFORMIX-Universal Server, you can define more than one function with the same name but with different parameter lists. Therefore, a function name alone might not identify a function. In that case, you must specify one of the following in the DROP FUNCTION statement:

- The SPECIFIC keyword and a specific name
- The parameter data types after the function name

The keyword FUNCTION, the function name, and the number, type, and order of parameters (as they appear from left to right in the DROP FUNCTION statement) make up the function signature. The function signature unambiguously identifies the function. For a given function, at least one item in the signature must be unique among all the functions stored in a name space or database.

Dropping a function removes the text and executable versions of the function.

You cannot use DROP FUNCTION to drop any type of procedure.

You can also use DROP ROUTINE to drop a function. For more information on DROP ROUTINE, see page 1-365.

Function Name

The function name can be the name of any user-defined function stored on the local database server. You can use a fully qualified function name to drop a function stored on a remote server, if either of the following conditions is true:

- The fully qualified function name uniquely identifies the function and you do not need to specify a parameter list to drop the function.
- All of the parameters the function accepts are of built-in data types.

You cannot drop a remote function if any of its parameters are opaque, distinct, collection, or row types.

The syntax of the function name is described in the Function Name segment on page 1-959.

Specific Name

A specific name uniquely identifies the function within the database. If you use the DROP SPECIFIC FUNCTION statement, you must use the function's specific name as it is defined in the CREATE FUNCTION statement.

With DROP SPECIFIC FUNCTION, you must use the specific name of a function. You cannot use the specific name of a procedure.

The syntax of the specific name is described in the Specific Name segment on page 1-1034.

Required Permissions

You must be the owner of the function or have the DBA privilege to use the DROP FUNCTION statement.

Examples

If you use parameter data types to identify a function, they follow the function name, as in the following example:

```
DROP FUNCTION compare(int, int)
```

If you use the specific name for the function, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC FUNCTION compare_point
```

SPL

SPL Functions

Because you cannot change the text of an SPL function, you must drop it using DROP FUNCTION or DROP ROUTINE and then re-create it using CREATE FUNCTION. 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 function within the same SPL function. ♦

References

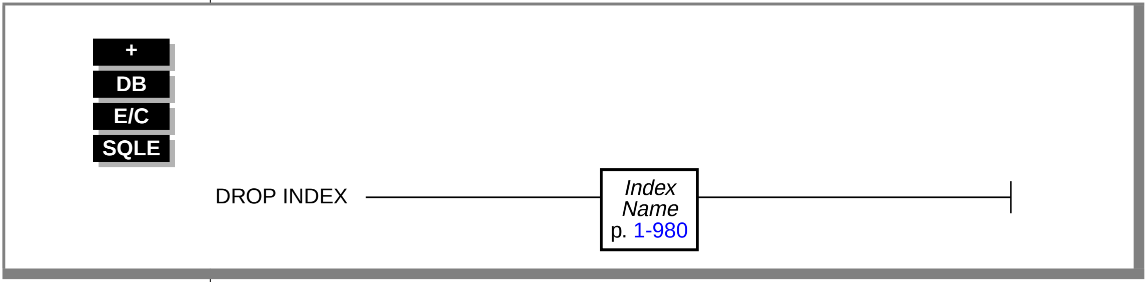
In this manual, see the CREATE FUNCTION, CREATE FUNCTION FROM, DROP FUNCTION, DROP ROUTINE, and EXECUTE FUNCTION statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of user-defined routines in Chapter 14 and the discussion of SPL routines in Chapter 14.

DROP INDEX

Use the DROP INDEX statement to remove a previously defined index from the database.

Syntax



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 **stores7** database must be the current database.

```
DROP INDEX stores7:joed.o_num_ix
```

You cannot use the DROP INDEX statement on a column or columns to drop a unique constraint that is created with a CREATE TABLE statement; you must use the ALTER TABLE statement to remove indexes that are created as constraints with a CREATE TABLE or ALTER TABLE statement.

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>
```

In this example, *tabid* and *constraint_id* are from the **systables** and **sysconstraints** system catalog tables, respectively. The **idxname** (index name) column in the **sysconstraints** table is then updated to reflect this change. For example, the renamed index name might be something like the following (quotes used to show the spaces):

```
“121_13”
```

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.

References

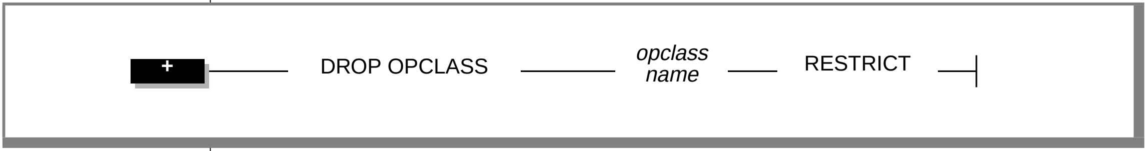
See the ALTER TABLE, CREATE INDEX, and CREATE TABLE statements in this manual.

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of indexes.

DROP OPCLASS

Use the DROP OPCLASS statement to remove an existing operator class from the database.

Syntax



Element	Purpose	Restrictions	Syntax
opclass name	Name of the operator class being dropped	The operator class must have been created with the CREATE OPCLASS statement. You must remove all dependent objects (such as indexes) defined on this operator class, before you can drop the operator class.	Identifier, p. 1-962

Usage

You must be the owner of the operator class or have DBA privilege to use the DROP OPCLASS statement.

The RESTRICT keyword is required with the DROP OPCLASS statement. RESTRICT causes DROP OPCLASS to fail if the database contains indexes or secondary access methods that use the *opclass name* operator class. The DROP OPCLASS statement cannot drop these indexes or the access methods.

The following DROP OPCLASS statement drops an operator class called **abs_btree_ops**:

```
DROP OPCLASS abs_btree_ops RESTRICT
```

References

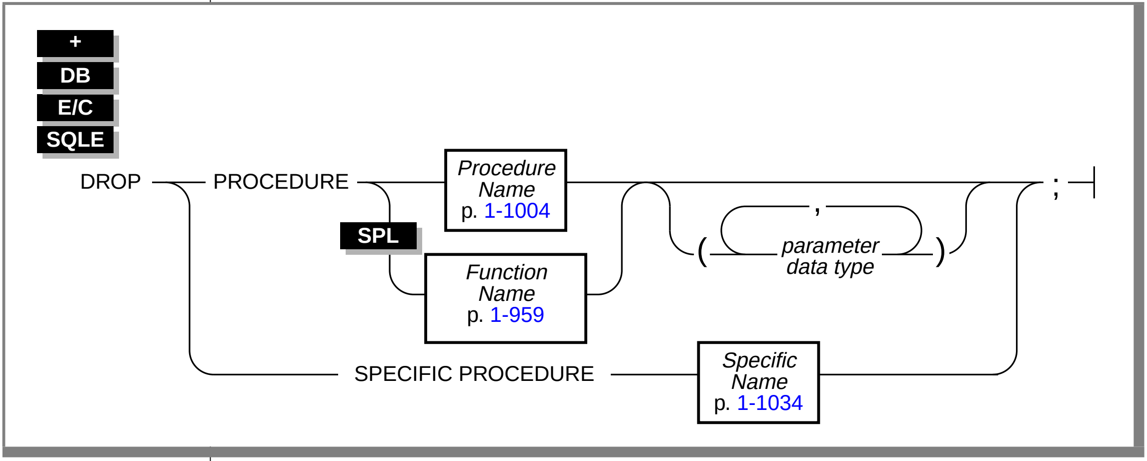
See CREATE OPCLASS in this manual.

For information on how to create or extend an operator class, see the *Extending INFORMIX-Universal Server: Data Types* manual.

DROP PROCEDURE

Use the DROP PROCEDURE statement to remove an external procedure or an SPL procedure from the database.

Syntax



Element	Purpose	Restrictions	Syntax
parameter data type	The data type of the parameter	The data type must be the data type (or list of data types) specified in the CREATE PROCEDURE statement when the procedure was created.	Identifier, p. 1-962

Usage

A procedure is a user-defined routine that does not return a value. In INFORMIX-Universal Server, you can write procedures in Stored Procedure Language (SPL) or in an external language, such as C.

Because you can overload routines in INFORMIX-Universal Server, you can define more than one procedure with the same name but with different parameter lists. Therefore, a procedure name alone might not identify a procedure. In that case, you must specify one of the following in the DROP PROCEDURE statement:

- The SPECIFIC keyword and a specific name
- The parameter data types after the procedure name

The keyword PROCEDURE, the procedure name, and the number, type, and order of parameters (as they appear from left to right in the DROP PROCEDURE statement) make up the signature for the procedure. The procedure signature unambiguously identifies the procedure. For a given procedure, at least one item in the signature must be unique among all the procedures stored in a name space or database.

Dropping a procedure removes the text and executable versions of the procedure.

You can also use DROP ROUTINE to drop a procedure. For more information on DROP ROUTINE, see page 1-365.

Procedure Name

The procedure name can be the name of any user-defined procedure stored on the local database server. You can use a fully qualified procedure name to drop a procedure stored on a remote server, if either of the following conditions is true:

- The fully qualified procedure name uniquely identifies the procedure and you do not need to specify a parameter list to drop the procedure.
- All of the parameters the procedure accepts are built-in data types.

You cannot drop a remote procedure if any of its parameters are opaque, distinct, collection, or row types.

The syntax of a procedure name, including a fully qualified procedure name, is described in the Procedure Name segment on page 1-1004.

Specific Name

A specific name uniquely identifies the procedure within the database. If you use the DROP SPECIFIC PROCEDURE statement, you must use the specific name for the procedure as it is defined in the CREATE PROCEDURE statement.

SPL

When you use DROP SPECIFIC PROCEDURE with SPL routines, you can use the name of an SPL procedure or SPL function. This feature provides backward compatibility with earlier Informix products and is described in "SPL Backward Compatibility Option" on page 1-362. ♦

EXT

When you use DROP SPECIFIC PROCEDURE with external routines, you must use the specific name of a procedure. You cannot use the specific name of a function. ♦

The syntax of the specific name is described in the Specific Name segment on page 1-1034.

Required Permissions

You must be the owner of the procedure or have the DBA privilege to use the DROP PROCEDURE statement.

Examples

If you use parameter data types to identify a procedure, they follow the procedure name, as in the following example:

```
DROP PROCEDURE compare(int, int)
```

If you use the specific name for the procedure, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC PROCEDURE compare_point
```

SPL

SPL Procedures

Because you cannot change the text of an SPL function, you must drop it using DROP PROCEDURE or DROP ROUTINE and then recreate it using CREATE PROCEDURE. If you want to recreate it after it is dropped, make sure that you have a copy of the SPL procedure text somewhere outside the database.

You cannot drop an SPL procedure within the same SPL procedure. ♦

SPL

SPL Backward Compatibility Option

For backward compatibility with earlier Informix products, you can use DROP PROCEDURE to drop an SPL function (that is, an SPL routine that returns a value). However, Informix recommends that you use DROP PROCEDURE only with procedures. You can also use DROP FUNCTION or DROP ROUTINE to drop an SPL function. ♦

References

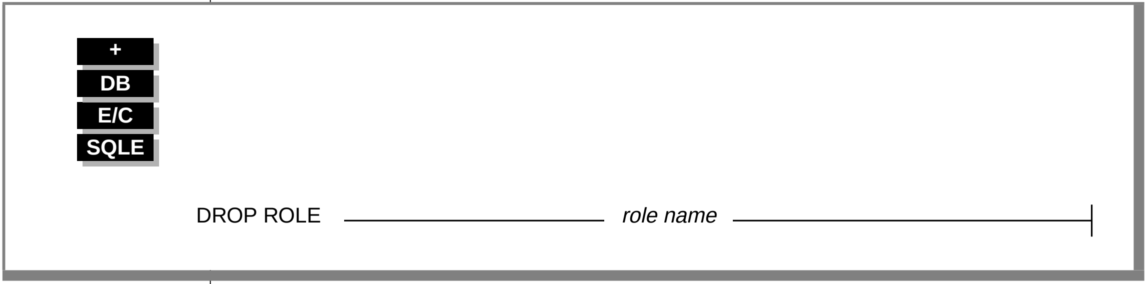
In this manual, see the CREATE PROCEDURE, CREATE PROCEDURE FROM, DROP PROCEDURE, DROP ROUTINE, and EXECUTE PROCEDURE statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of user-defined routines in Chapter 14 and the discussion of SPL routines in Chapter 14.

DROP ROLE

Use the DROP ROLE statement to remove a previously created role from the database.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name of the role being dropped	The role name must have been created with the CREATE ROLE statement	Identifier, p. 1-962

Usage

The DROP ROLE statement is used to remove an existing role. 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 fragment-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.

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.

The following statement drops the role **engineer**:

```
DROP ROLE engineer
```

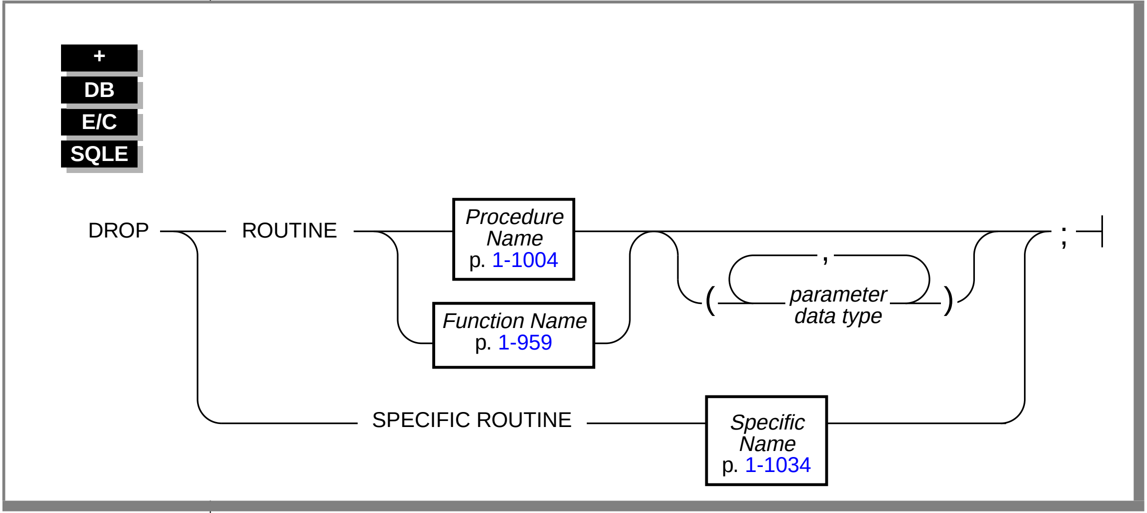
References

See the CREATE ROLE, GRANT, REVOKE, and SET ROLE statements in this manual.

DROP ROUTINE

Use the DROP ROUTINE statement to remove any type of user-defined routine from the database.

Syntax



Element	Purpose	Restrictions	Syntax
parameter data type	The data type of the parameter	The data type must be the data type (or list of data types) specified in the CREATE FUNCTION or CREATE PROCEDURE statement when the routine was created.	Identifier, p. 1-962

Usage

You can use DROP ROUTINE with any type of routine—an external function, an external procedure, an SPL function, or an SPL procedure. The DROP ROUTINE statement is useful when you do not know whether a routine is a function or a procedure.

Because you can overload routines in INFORMIX-Universal Server, you can define more than one routine with the same name but with different parameter lists. Therefore, a routine name alone might not uniquely identify a routine. In that case, you must specify one of the following in the DROP ROUTINE statement:

- The SPECIFIC keyword and a specific name
- The parameter data types after the routine name

The keyword PROCEDURE or FUNCTION, the routine name, and the number, type, and order of parameters (as they appear from left to right in the DROP ROUTINE statement) make up the routine signature. The routine signature unambiguously identifies the routine. For a given routine, at least one item in the signature must be unique among all the routines stored in a name space or database.

Dropping a routine removes the text and executable versions of the routine.

You can also use DROP FUNCTION to drop a function and DROP PROCEDURE to drop a procedure. The DROP FUNCTION statement is described on page 1-351, and the DROP PROCEDURE statement is described on page 1-359.

Procedure Name or Function Name

A procedure name identifies a routine registered with the CREATE PROCEDURE statement and a function name identifies a function registered with the CREATE FUNCTION statement. Without a database qualifier, the routine must reside on the local database server.

You can use a fully qualified procedure name to drop a routine stored on a remote server, if either of the following conditions is true:

- The fully qualified name uniquely identifies the routine, and you do not need to specify a parameter list to drop the procedure.
- The parameter list contains only built-in data types.
- No ambiguity is caused by both a procedure and a function having the same name.

You cannot drop a remote procedure if any of its parameters are opaque, distinct, collection, or row types.

For the syntax of a fully qualified name, see "Procedure Name" on page 1-1004 or "Function Name" on page 1-959.

Specific Name

A specific name uniquely identifies a routine within the database. If you use the DROP SPECIFIC ROUTINE statement, you must use the identifier assigned with the SPECIFIC clause of the CREATE PROCEDURE or CREATE FUNCTION statement.

The syntax of Specific Name is described in the Specific Name segment on page 1-1034.

Required Permissions

You must be the owner of the routine or have the DBA privilege to use the DROP ROUTINE statement.

Examples

If you use parameter data types to identify a routine, they follow the routine name, as in the following example:

```
DROP ROUTINE compare(int, int)
```

If you use the specific name for the routine, you must use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC ROUTINE compare_point
```

SPL

SPL Routines

Because you cannot change the text of an SPL routine, you must drop it with DROP PROCEDURE, DROP FUNCTION, or DROP ROUTINE and then re-create it with CREATE PROCEDURE or CREATE FUNCTION. If you want to recreate it after it is dropped, make sure that you have a copy of the SPL routine text somewhere outside the database.

You cannot drop an SPL routine from within the same SPL routine. ♦

References

In this manual, see the CREATE FUNCTION, CREATE PROCEDURE, DROP FUNCTION, DROP PROCEDURE, EXECUTE FUNCTION, and EXECUTE PROCEDURE statements.

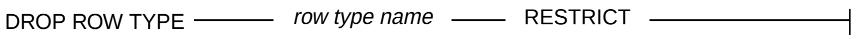
In the *Informix Guide to SQL: Tutorial*, see the discussion of user-defined routines in Chapter 14 and the discussion of SPL routines in Chapter 14.

DROP ROW TYPE

Use the DROP ROW TYPE statement to remove an existing named row type from the database.

Syntax

+
DB
E/C
SQLE



Element	Purpose	Restrictions	Syntax
row type name	The name of the named row type to be dropped	The type must have been created with the CREATE ROW TYPE statement. The named row type must already exist. The named row type cannot be dropped if it is currently used in any columns, tables or inheritance hierarchies.	Data Type, p. 1-855 Identifier, p. 1-962 The named row type can be of the form *owner.type*.

Usage

You must be the owner of the row type or have the DBA privilege to use the DROP ROW TYPE statement.

You cannot drop a named row type if the row type 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 row type.
- The row type is a supertype in an inheritance hierarchy.
- A view is defined on the row type.

To drop a named row type column from a table, use ALTER TABLE.

The DROP ROW TYPE statement does *not* 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:

- 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 types have child types.

Example

The following statement drops the row type named **employee_t**:

```
DROP ROW TYPE employee_t RESTRICT
```

References

See the CREATE ROW TYPE statement in this manual to learn how to create row types.

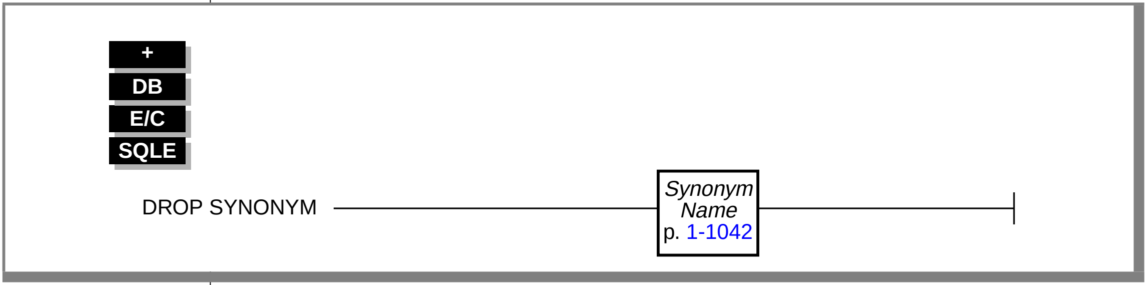
See the *Informix Guide to SQL: Reference* for a description of the system catalog tables.

See Chapter 10 of the *Informix Guide to SQL: Tutorial* for a discussion of named row types.

DROP SYNONYM

Use the DROP SYNONYM statement to remove a previously defined synonym from the database.

Syntax



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 is dropped, any synonyms that are in the same database as the table and that refer to the table are also dropped.

If a synonym refers to an external table, and the table is dropped, the synonym remains in place until you explicitly drop it using DROP SYNONYM. You can create another table or synonym in place of the dropped table and give the new object the name of the dropped table. The old synonym then refers to the new object. See the CREATE SYNONYM statement for a complete discussion of synonym chaining.

References

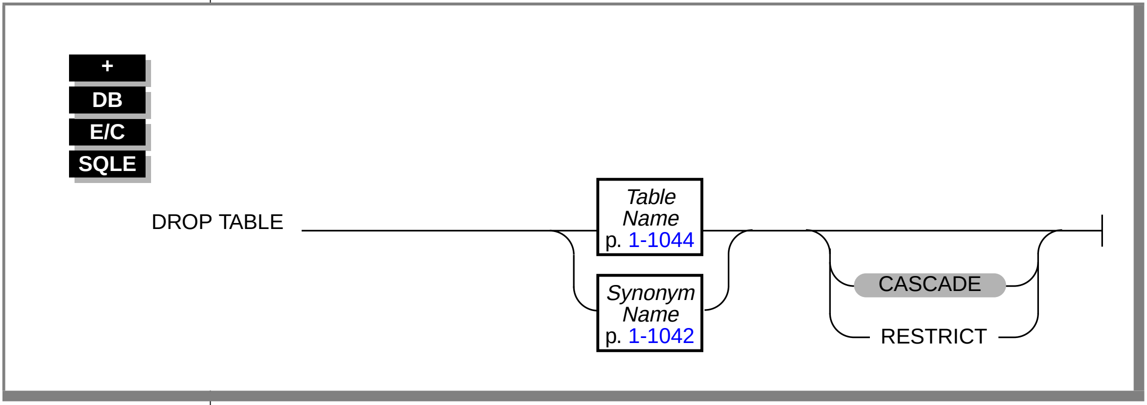
See the CREATE SYNONYM statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 11.

DROP TABLE

Use the DROP TABLE statement to remove a previously defined table, along with its associated indexes and data from the database.

Syntax



Usage

You must be the owner of the table or have the DBA privilege to use the DROP TABLE statement.

DB

If you issue a DROP TABLE statement, you are not prompted to verify that you want to delete an entire table. ♦

Effects of 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 for it, and any authorizations you have granted on the table. You also drop all views based on the table and any violations and diagnostics tables associated with the table. You do not remove any synonyms for the table that have been created in an external database.

Specifying CASCADE Mode

The CASCADE mode means that a DROP TABLE statement removes the table and all 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. 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

With the RESTRICT keyword, you can control the success or failure of the drop operation for supertables, for tables that have referential constraints and views defined on the table, and for tables that have violations and diagnostics tables associated with the table. 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 name.*
- Existing views are defined on *table name.*
- Any violations and diagnostics tables are associated with *table name.*
- The *table name* is the supertable in an inheritance hierarchy.

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 data type contains spatial or multi-representational data, it might provide a choice of how to store the data: inside the internal structure or, for very 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 the *Extending INFORMIX-Universal Server: Data Types* manual.

Tables That Cannot Be Dropped

You cannot drop the following types of tables:

- You cannot drop any system catalog tables.
- You cannot drop a table that is not in the current database.
- You cannot drop a violations 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.

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 stores7@accntg:joed.state
```

References

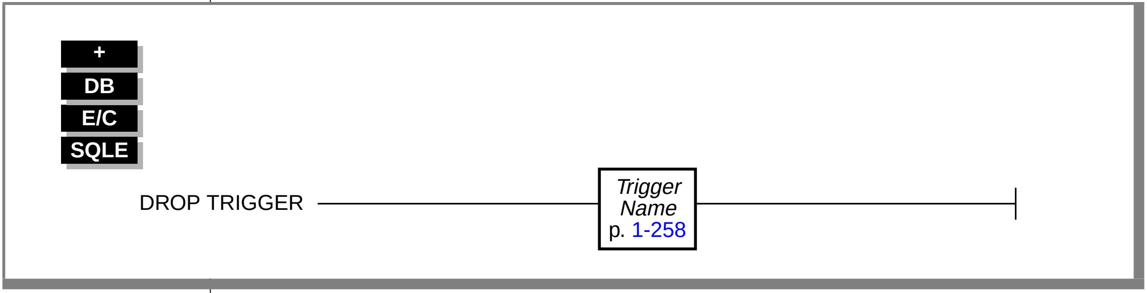
See the CREATE TABLE and DROP DATABASE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of data integrity and creating a table in Chapter 4 and Chapter 9, respectively.

DROP TRIGGER

Use the DROP TRIGGER statement to remove a previously defined trigger definition from the database.

Syntax



Usage

You must be the owner of the trigger or have the DBA privilege to use the DROP TRIGGER statement.

Dropping a trigger removes the text of the trigger definition and the executable trigger from the database.

The following statement drops the **items_pct** trigger:

```
DROP TRIGGER items_pct
```

You cannot drop a trigger inside a stored procedure if the procedure is called within a data manipulation statement. For example, in the following INSERT statement, a DROP TRIGGER statement is illegal inside the stored procedure **proc1**:

```
INSERT INTO orders EXECUTE PROCEDURE proc1(vala, valb)
```

References

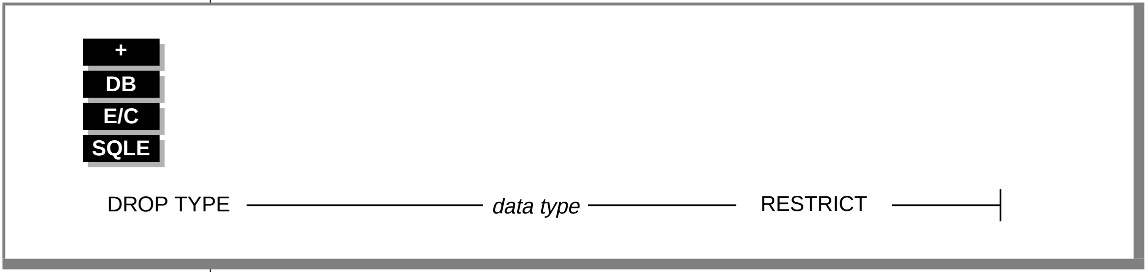
See the CREATE PROCEDURE statement in this manual for more information about a stored procedure that is called within a data manipulation statement.

For more information about triggers, see the CREATE TRIGGER statement in this manual.

DROP TYPE

Use the DROP TYPE statement to remove an existing distinct or opaque data type from the database.

Syntax



Element	Purpose	Restrictions	Syntax
data type	The distinct or opaque data type to be removed from the database	The type must have been created with the CREATE DISTINCT TYPE or CREATE OPAQUE TYPE statement. Do not remove built-in types.	Data Type, p. 1-855 The distinct type or opaque type can be of the form *owner.type*.

Usage

To drop a distinct or opaque type with the DROP TYPE statement, you must be the owner of the data type or have the DBA privilege.

When use the DROP TYPE 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 functions whose definitions reference the type.

The following statement drops the **new_type** type:

```
DROP TYPE new_type RESTRICT
```

References

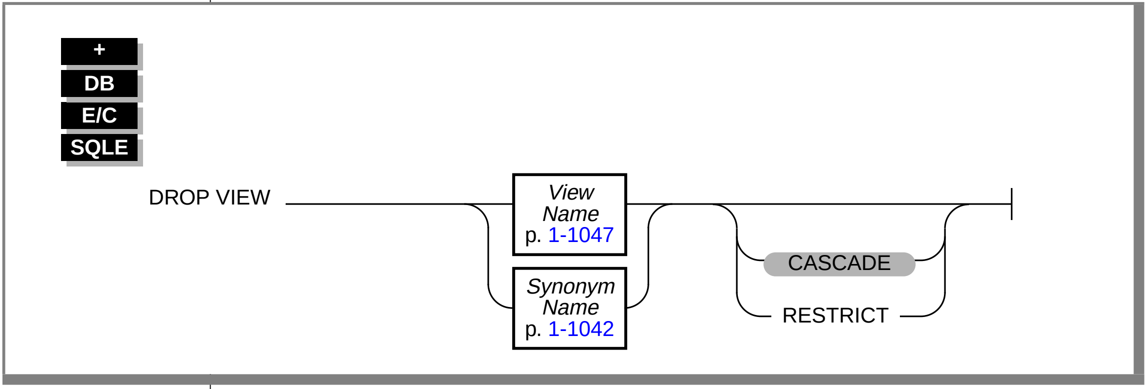
See the CREATE DISTINCT TYPE and CREATE OPAQUE TYPE statements in this manual for information. See the CREATE ROW TYPE and DROP ROW TYPE statements in this manual for information about how to define and remove row types from the database. See the CREATE TABLE statement in this manual for more information about creating tables that reference a data type.

See the Data Types segment in this manual for more information about data types.

DROP VIEW

Use the DROP VIEW statement to remove a previously defined view from the database.

Syntax



Usage

You must own the view or have the DBA privilege to use the DROP VIEW statement.

When you drop *view name*, you also drop all views that have been defined in terms of that view. You can also specify this default condition with the CASCADE keyword.

When you use the RESTRICT keyword in the DROP VIEW statement, the drop operation fails if any existing views are defined on *view name*, which 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
```

References

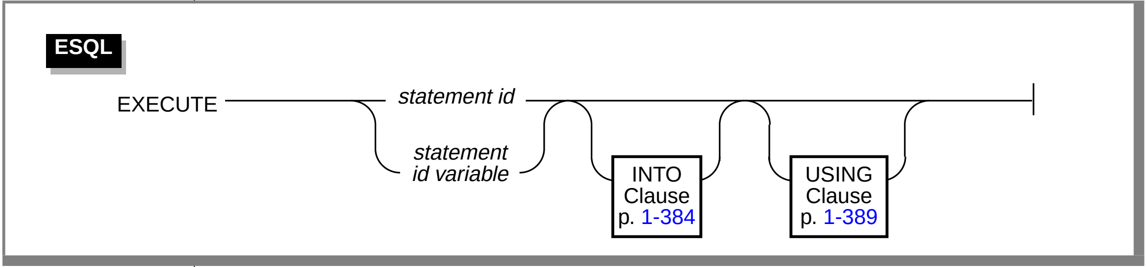
See the CREATE VIEW and DROP TABLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of views in Chapter 11.

EXECUTE

Use the EXECUTE statement to run a previously prepared statement or set of statements.

Syntax



Element	Purpose	Restrictions	Syntax
statement id	Identifier for an SQL statement	You must have defined the statement identifier in a previous PREPARE statement. 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.	PREPARE, p. 1-538
statement id variable	Host variable that identifies an SQL statement	You must have defined the host variable in a previous PREPARE statement. The host variable must be a character data type.	PREPARE, p. 1-538

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. The following example shows an EXECUTE statement within an INFORMIX-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.

If the statement contained question mark (?) placeholders, you use the USING clause to provide specific values for them before execution.For more information, see the "USING Clause" on page 1-389.

You can execute any prepared statement except the following:

- A prepared SELECT statement that returns more than one row

  When you use a prepared SELECT statement to return multiple rows of data, you can use the DECLARE, OPEN, and FETCH cursor statements to retrieve the data rows. In addition, you can use EXECUTE on a prepared SELECT INTO TEMP statement to achieve the same result.

- A prepared EXECUTE FUNCTION statement for an SPL function that returns more than one row

  When you prepare an EXECUTE FUNCTION statement for a SPL function that returns multiple rows, you need to use the DECLARE, OPEN and FETCH cursor statements just as you would with a SELECT statement that returns multiple rows.

For more information on how to execute a SELECT or an EXECUTE FUNCTION, see "PREPARE" on page 1-538.

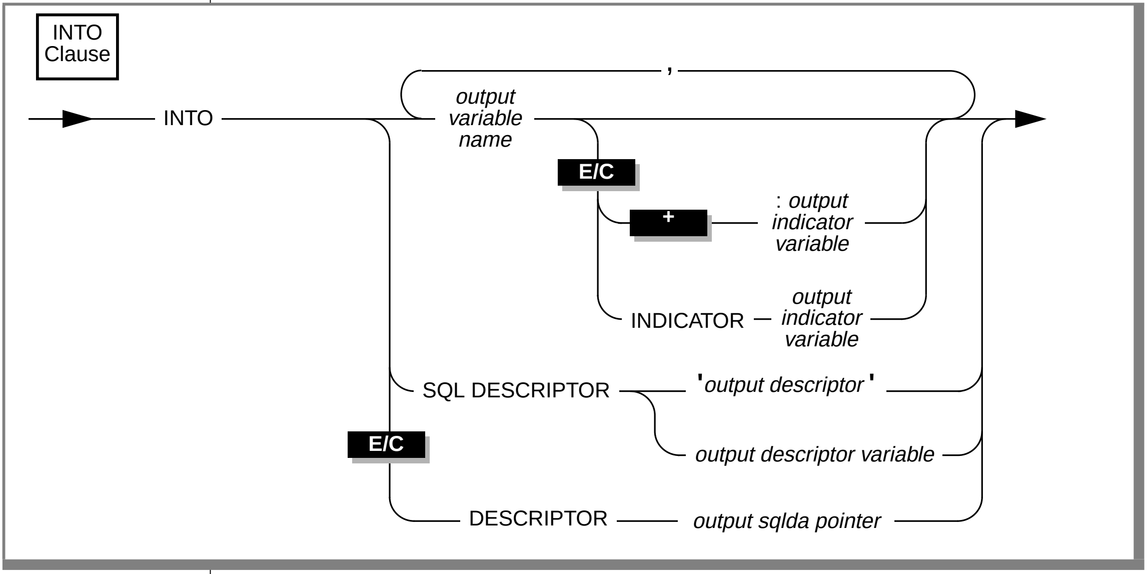
If you create or drop a trigger after you prepared a triggering INSERT, DELETE, or UPDATE statement, the prepared statement returns an error when you execute it.

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of a statement identifier is global to the program, so 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 a statement identifier to the file in which it is executed, you can preprocess all the files with the **-local** command-line option. See your SQL API product manual for more information, restrictions, and performance issues when you preprocess files with the **-local** option.

INTO Clause



Element	Purpose	Restrictions	Syntax
output descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-1010
output descriptor variable	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-1010
output indicator variable	Host variable that receives a return code if null data is placed in the corresponding *output variable*	Variable cannot be DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
output sqlda pointer	Points to an **sqlda** structure that defines the data type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement.	You cannot begin an output sqlda pointer with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-335
output variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

The INTO clause allows you to save the return values of the following 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 statement for an SPL function that returns only one group of values

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 the PREPARE, DECLARE, OPEN, and FETCH sequence of statements to retrieve values.

Important: *If you execute a prepared SELECT statement that returns more than one row of data or a prepared EXECUTE FUNCTION 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, make sure after you select the data that you check the indicator variable that is associated with the column to determine if the value is null.

The following list describes how to use the INTO clause with the EXECUTE statement:

1. Declare the output variables that the EXECUTE statement uses in its INTO clause.
2. Use the PREPARE statement to prepare your SELECT or EXECUTE FUNCTION statement.
3. Use the EXECUTE statement, with the INTO clause, to execute your SELECT or EXECUTE FUNCTION statement.

You can specify any of the following items to store return values from a SELECT or EXECUTE FUNCTION statement before you execute it:

- A host variable name (if the number and data type of the return values are known at compile time)
- A system-descriptor area that identifies a dynamically generated descriptor for the value

E/C

- An **sqlda** structure that is a pointer to a dynamically generated descriptor for the value. ♦

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 statement returns as host variables in your program. You 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 returns. The data type of each variable must be compatible with the corresponding return value of the prepared statement.

The following example shows how to use the INTO clause of an EXECUTE statement to execute a singleton SELECT and store the column values in host variables:

```
EXEC SQL prepare sel1 from
    'select fname, lname from customer \
    where customer_num =123';
EXEC SQL execute sel1 into :fname, :lname;
```

The following example shows how to use the INTO clause to execute a SELECT statement that returns 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);
    }
```

For more information on how to use input parameters, see "USING Clause" on page 1-389.

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

You can also use an **sqlda** structure (page 1-388) to supply parameters dynamically. However, 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 show how to use system-descriptor area to execute prepared statements in 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 further information, refer to the discussion of the system-descriptor area in the *INFORMIX-ESQL/C Programmer's Manual*.

Saving Values in an sqlda Structure

E/C

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.

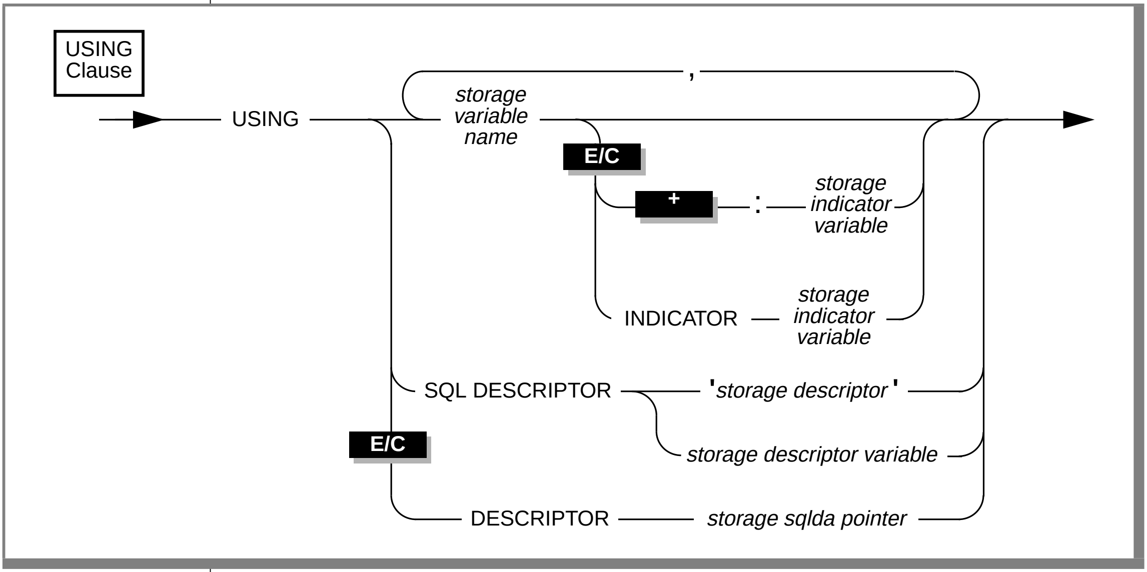
The following example shows how to use an **sqlda** structure to execute a prepared statement in INFORMIX-ESQL/C:

```
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 statement returns.

For more information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

USING Clause



Element	Purpose	Restrictions	Syntax
storage descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. Make sure surrounding quotes are single.	Quoted String, p. 1-1010
storage descriptor variable	Host variable name that identifies a system-descriptor area	System-descriptor area must already be allocated.	Variable name must conform to language-specific rules for variable names.
storage indicator variable	Host variable that receives a return code if null data is placed in the corresponding *data variable*. It receives truncation information if truncation occurs.	Variable cannot be DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
storage sqlda pointer	Points to an **sqlda** structure that defines the data type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement.	You cannot begin *storage sqlda pointer* with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-335
storage variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

The USING clause specifies 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.

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 area that identifies a dynamically-generated descriptor for the value
- An **sqlda** structure that is a pointer to a dynamically-generated descriptor for the value ♦

E/C

Supplying Parameters Through Host or Program Variables

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 prepared statement question-mark (?) parameters in a one-to-one correspondence, from left to right.

You must supply one storage variable name 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 INFORMIX-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 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.

X/O

You can also use an **sqlda** structure (page 1-392) to dynamically supply parameters. However, a system-descriptor area conforms to the X/Open standards. ♦

To specify a system-descriptor area as the location of parameters, use 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 following example show how to use system-descriptor area to execute prepared statements in INFORMIX-ESQL/C:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

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 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 further information, refer to the discussion of the system-descriptor area in the *INFORMIX-ESQL/C Programmer's Manual*.

Supplying INFORMIX-ESQL/C Parameters Through an sqlda Structure

E/C

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. To specify an **sqlda** structure as the location of parameters, use the USING DESCRIPTOR clause of the EXECUTE statement. Each time the EXECUTE statement is run, the values that the **sqlda** structure describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The following example shows how to use an **sqlda** structure to execute a prepared statement in INFORMIX-ESQL/C:

```
struct sqlda *pointer2;
...
EXEC SQL execute prep_stmt using descriptor pointer2;
```

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.

For more information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

Error Conditions with EXECUTE

Following an EXECUTE statement, the **sqlca** record (see the *INFORMIX-ESQL/C Programmer's Manual*) can reflect two results:

- The **sqlca** record can reflect an exception within the EXECUTE statement.
- The **sqlca** structure can also reflect the success or failure of the prepared statement that EXECUTE runs. For example, when an UPDATE ... WHERE ... statement within a prepared object processes zero rows, the database server sets **sqlca.sqlcode** to 100.

In a database that is not ANSI compliant, if any statement fails to access any rows, the database server returns an **SQLCODE** value of zero(0).

ANSI

In an ANSI-compliant database, if you prepare and execute any of the following statements, and no rows are returned, the database server returns an **SQLCODE** value of `SQLNOTFOUND (100)`:

- INSERT INTO *table-name* SELECT ... WHERE ...
- SELECT INTO TEMP ... WHERE ...
- DELETE ... WHERE
- UPDATE ... WHERE ... ♦

In a multistatement prepare, if any statement in the preceding list fails to access rows, in either ANSI databases or databases that are not ANSI compliant, the database server returns `SQLNOTFOUND (100)`.

Tip: *When you encounter an **SQLCODE** error, a corresponding **SQLSTATE** error value might exist. See the GET DIAGNOSTICS statement for information about the **SQLSTATE** status variable.*

References

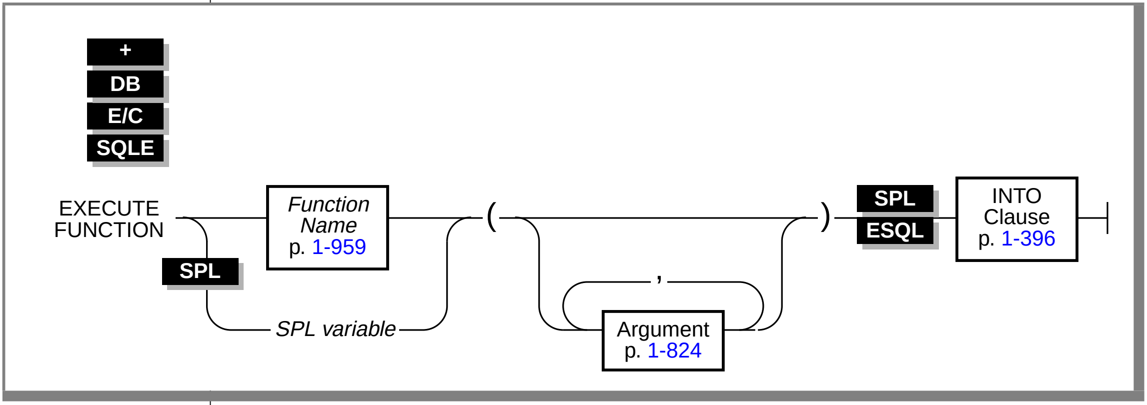
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE IMMEDIATE, GET DESCRIPTOR, GET DIAGNOSTICS, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the EXECUTE statement in Chapter 5.

EXECUTE FUNCTION

Use the EXECUTE FUNCTION statement to execute an SPL function or external function.

Syntax



Element	Purpose	Restrictions	Syntax
SPL variable	A variable created with the DEFINE statement that contains the name of an SPL routine to be executed	The SPL variable must be CHAR, VARCHAR, NCHAR, or NVARCHAR data type. The name assigned to *SPL variable* must be non-null and the name of an existing SPL function.	Identifier, p. 1-962

Usage

The EXECUTE FUNCTION statement invokes the named user-defined function, specifies its arguments, and determines where the results are returned. A function is a user-defined routine that returns one or more values. An external function, written in a language other than SPL, returns exactly one value. An SPL function can return one or more values.

You can use EXECUTE FUNCTION to execute an SPL function or an external function. You cannot use EXECUTE FUNCTION to execute any type of user-defined procedure. Instead, use the EXECUTE PROCEDURE statement to execute procedures.

How EXECUTE FUNCTION Works

For a 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 an EXECUTE FUNCTION statement specifies fewer arguments than the called function expects, the unspecified arguments are said to be *missing*. Missing arguments are initialized to their corresponding parameter default values, if you specified default values. The syntax of specifying default values for parameters in described in "Routine Parameter List" on page 1-1028.

The EXECUTE FUNCTION statement returns an error under the following conditions:

- It specifies more arguments than the called 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 function 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.

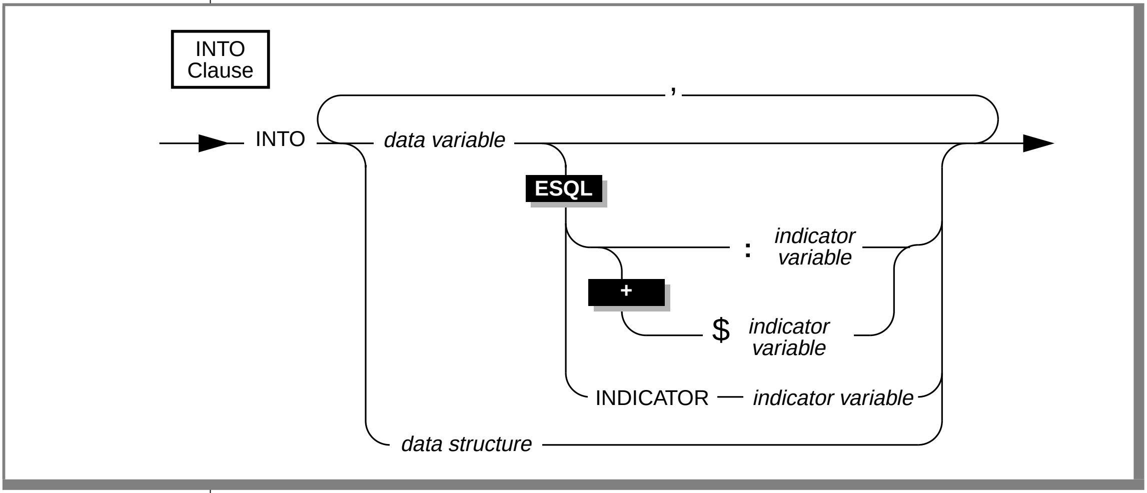
Function Name

With EXECUTE FUNCTION, you can use either of the following types of names to execute a remote function:

- If you use a *fully qualified function name*, the database server determines which function to use based only on the routine type (which is FUNCTION) and the function name.
- If you use a *function signature*, the database server uses the function name and its full parameter list during routine resolution to determine which function to use.

For more detailed information, see the Function Name segment on page 1-959.

INTO Clause



Element	Purpose	Restrictions	Syntax
data variable	A variable that receives the value returned by a function	If you issue this statement within an ESQL/C program, the *data variable* must be a host variable. If you issue this statement within an SPL routine, the *data variable* must be an SPL variable. If you issue this statement within a CREATE TRIGGER statement, the *data variable* must be column names within the triggering table or another table.	The name of a receiving variable must conform to language-specific rules for variable names. For the syntax of SPL variables, see Identifier, p. 1-962. For the syntax of column names, see Identifier, p. 1-962.
data structure	A structure that has been declared as a host variable	The individual elements of the structure must be matched appropriately to the data type of values being selected.	The name of the data structure must conform to language-specific rules for data structures.
indicator variable	A program variable that receives a return code if null data is placed in the corresponding *data variable*	This parameter is optional, but you should use an indicator variable if the possibility exists that the value of the corresponding *data variable* is null.	The name of the indicator variable must conform to language-specific rules for indicator variables.

You must specify an INTO clause with EXECUTE FUNCTION to name the variables that receive the values that the 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 INFORMIX-ESQL/C:

```
EXEC SQL execute function cust_num(fname, lname, company_name)
    into :c_num;
```

INTO Clause with Indicator Variables

ESQL

You should use an indicator variable if the possibility exists that data returned from the user-defined function statement is null. See the *INFORMIX-ESQL/C Programmer's Manual* for more information about indicator variables. ♦

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.

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 *DataBlade API Programmer's Manual*. ♦

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 Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

E/C

In an INFORMIX-ESQL/C program, use the DECLARE statement to declare the function cursor and the FETCH statement to fetch the rows individually from the function cursor. You can put the INTO clause in the FETCH statement rather than in the EXECUTE FUNCTION statement, but you cannot put it in both. The following 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 holds the fetched values. For more information, see the FOREACH statement on page 2-27. ♦

Preparing an EXECUTE FUNCTION...INTO Statement

ESQL

You cannot prepare an EXECUTE FUNCTION statement that has an INTO clause. You can prepare the EXECUTE FUNCTION without the INTO clause, declare a function cursor for the prepared statement, open the cursor, and then use the FETCH statement with an INTO clause to fetch the return values into the program variable(s). Alternatively, you can 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. Then open the cursor, and fetch the return 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, you can use an SPL variable to hold the routine name, instead of listing the explicit name of an SPL routine.

For more information about how to execute SPL functions dynamically, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

References

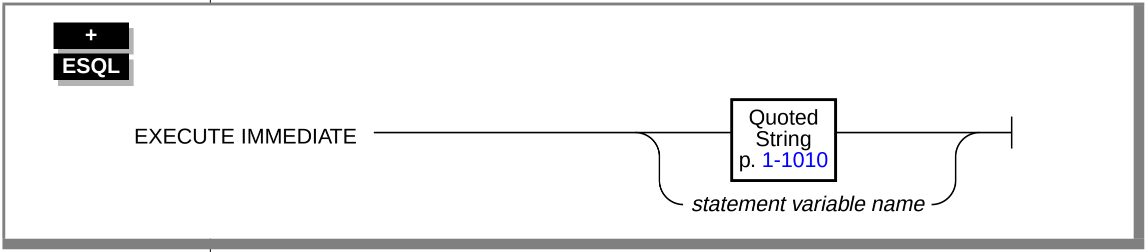
See the CREATE FUNCTION, CREATE FUNCTION FROM, DROP FUNCTION, DROP ROUTINE, and EXECUTE PROCEDURE statements in Chapter 1 of this manual. Also see the CALL and FOREACH statements in Chapter 2 of this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of writing SPL routines in Chapter 14.

EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to perform the functions of the PREPARE, EXECUTE, and FREE statements.

Syntax



Element	Purpose	Restrictions	Syntax
statement variable name	Host variable whose value is a character string that consists of one or more SQL statements	The host variable must have been defined within the program. The variable must be character data type. For additional restrictions, see "EXECUTE IMMEDIATE and Restricted Statements" on page 1-402 and "Restrictions on Allowed Statements" on page 1-403.	Variable name must conform to language-specific rules for variable names.

Usage

The quoted string is a character string that includes one or more SQL statements. The string, or the contents of *statement variable name*, is parsed and executed if correct; then all data structures and memory resources are released immediately. In the usual method of dynamic execution, these functions are distributed among the PREPARE, EXECUTE, and FREE statements.

The EXECUTE IMMEDIATE statement makes it easy to execute dynamically a single simple SQL statement, which is constructed during program execution. For example, you could 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 following example shows the EXECUTE IMMEDIATE statement in INFORMIX-ESQL/C:

```
sprintf(cdb_text, "create database %s", usr_db_id);
EXEC SQL execute immediate :cdb_text;
```

EXECUTE IMMEDIATE and Restricted Statements

You *cannot* use the EXECUTE IMMEDIATE statement to execute the following SQL statements.

CLOSE
CONNECT
CREATE FUNCTION FROM
CREATE PROCEDURE FROM
DECLARE
DISCONNECT
EXECUTE
EXECUTE FUNCTION
EXECUTE PROCEDURE (if the SPL routine returns values)
FETCH
GET DIAGNOSTICS
GET DESCRIPTOR
OPEN
OUTPUT
PREPARE
SELECT
SET CONNECTION
SET DESCRIPTOR
WHENEVER

Use a PREPARE statement 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 *statement variable name*:

- 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 or terminator, such as the dollar sign ($), colon (:), or the words EXEC SQL.
- 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 EXECUTE or a cursor to process prepared accesses to collection variables.

References

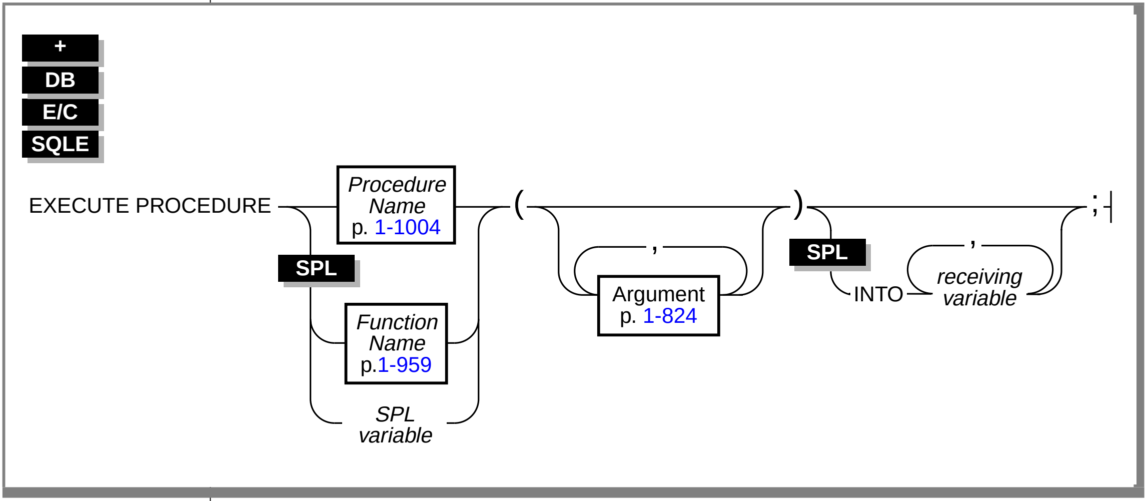
See the EXECUTE, FREE, and PREPARE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of quick execution in Chapter 5.

EXECUTE PROCEDURE

Use the EXECUTE PROCEDURE statement to execute an SPL procedure or an external procedure.

Syntax



Element	Purpose	Restrictions	Syntax
receiving variable	A variable that receives the value returned by an SPL function that you execute with EXECUTE PROCEDURE.	If you issue this statement within an ESQL/C program, the *receiving variable* must be a host variable. If you issue this statement within an SPL routine, the *receiving variable* must be an SPL variable. If you issue this statement within a CREATE TRIGGER statement, the *receiving variable* must be a column name within the triggering table or another table.	The name of a receiving variable must conform to language-specific rules for variable names. For the syntax of SPL variables, see Identifier, p. 1-962**.** For the syntax of column names, see Identifier, p. 1-962.
SPL variable	A variable created with the DEFINE statement that contains the name of an SPL routine to be executed.	The *SPL variable* must have the data type CHAR, VARCHAR, NCHAR, or NVARCHAR. The name you assign to *SPL variable* must be non-null and the name of an existing routine.	Identifier, p. 1-962

Usage

The EXECUTE PROCEDURE statement invokes the named user-defined procedure and specifies its arguments. A procedure is a user-defined routine that does *not* return a value. Use EXECUTE PROCEDURE to execute an SPL procedure or an external procedure.

SPL

For backward compatibility with earlier Informix products, INFORMIX-Universal Server continues to support the INTO clause of the EXECUTE PROCEDURE statement to save values that a stored procedure returns. However, Informix recommends that you use EXECUTE PROCEDURE only with procedures and EXECUTE FUNCTION with functions. For more information, see "INTO Clause" on page 1-407. ♦

How EXECUTE PROCEDURE Works

For a procedure to be executed with the EXECUTE PROCEDURE statement, the following conditions must exist:

- The qualified procedure name or the procedure signature (the procedure name with its parameter list) must be unique within the name space or database.
- The procedure must exist.

If an EXECUTE PROCEDURE statement has fewer arguments than the called procedure expects, the unspecified arguments are said to be *missing*. Missing arguments are initialized to their corresponding parameter default values, if you specified default values. The syntax of specifying default values for parameters in described in "Routine Parameter List" on page 1-1028.

The EXECUTE PROCEDURE statement 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 procedure signature is not unique.
- No procedure with the specified name or signature is found.

Procedure Name

With EXECUTE PROCEDURE, you can use either of the following types of names to execute a remote procedure:

- If you use a *fully qualified procedure name*, the database server determines which procedure to use based only on the routine type (which is PROCEDURE) and the procedure name.
- If you use a *procedure signature*, the database server uses the procedure name and its full parameter list during routine resolution to determine which procedure to use.

For more detailed information, see the Procedure Name segment on page 1-1004.

INTO Clause

SPL

For backward compatibility with earlier Informix products, you can use EXECUTE PROCEDURE to execute a stored procedure that returns a value. To save the return values of a stored procedure, specify an INTO clause of EXECUTE PROCEDURE to name the variables that receive the return values.

If the stored procedure (or 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 the stored procedure returns more than one row or a collection data type, you must access the rows or collection elements with a cursor.

For more information on stored procedures of earlier Informix products, see the CREATE PROCEDURE statement. ♦

Dynamic Routine-Name Specification of SPL Procedures

SPL

Dynamic routine-name specification simplifies the writing of an SPL procedure 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, you can use an SPL variable to hold the routine name, instead of listing the explicit name of an SPL routine.

If the SPL variable names a stored procedure that returns a value, include the INTO clause of EXECUTE PROCEDURE to specify a *receiving variable* (or variables) to hold the value (or values) that the stored procedure returns.

For more information on how to execute SPL procedures dynamically, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

References

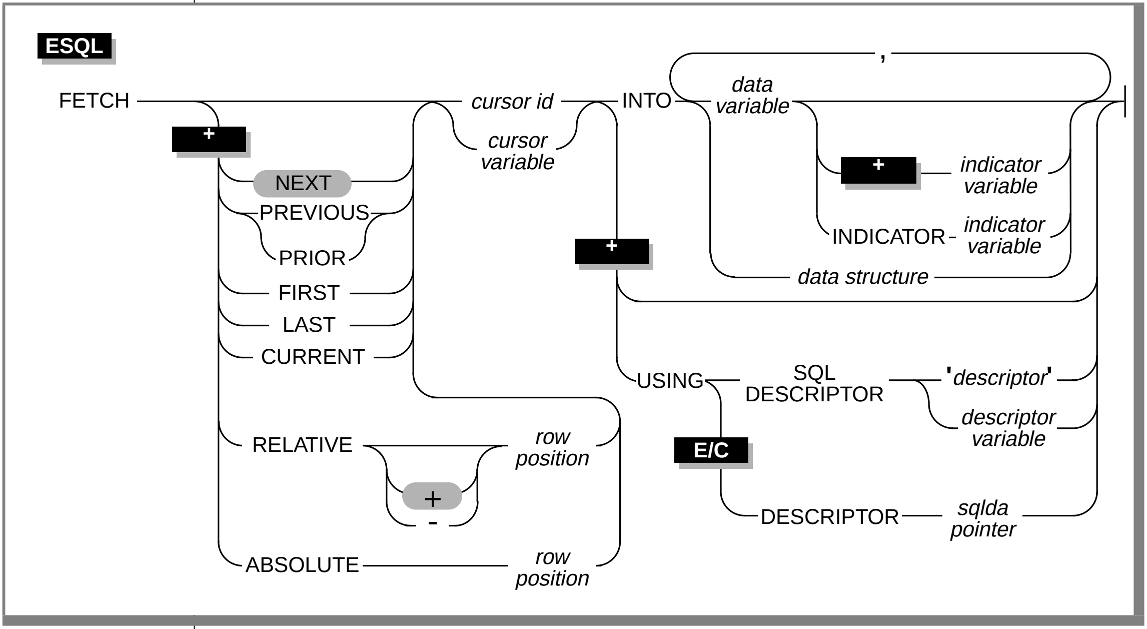
See the CREATE PROCEDURE, CREATE PROCEDURE FROM, DROP PROCEDURE, DROP ROUTINE, and EXECUTE FUNCTION statements in Chapter 1 of this manual. Also see the CALL statement in Chapter 2of this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of writing SPL routines in Chapter 14.

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.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for a select or function cursor from which rows are to be retrieved	A DECLARE statement must have previously created the cursor and the OPEN statement must have previously open it.	Identifier, p. 1-962
cursor variable	Host variable that holds the value of *cursor id*	The host variable must be a character data type. The cursor identified in *cursor variable* must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Variable name must conform to language-specific rules for variable names.
data structure	Structure that has been declared as a host variable	The individual members of the data structure must be matched appropriately to the type of values that are being fetched. If you use a program array, you must list both the array name and a specific element of the array in *data structure*.	Data-structure name must conform to language-specific rules for data-structure names.
data variable	Host variable that receives one value from the fetched row	The host variable must have a data type that is appropriate for the value that is fetched into it.	Variable name must conform to language-specific rules for variable names.
descriptor	Quoted string that identifies the system-descriptor area into which you fetch the contents of a row	The system-descriptor area must have been allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-1010
descriptor variable	Host variable name that holds the value of *descriptor*	The system-descriptor area that is identified in *descriptor variable* must have been allocated with the ALLOCATE DESCRIPTOR statement.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
indicator variable	Host variable that receives a return code if null data is placed in the corresponding *data variable*	This parameter is optional, but use an indicator variable if the possibility exists that the value of *data variable* is null. If you specify the indicator variable without the INDICATOR keyword, you cannot put a space between *data variable* and *indicator variable*. The rules for placing a prefix before *indicator variable* are language-specific. See your SQL API manual for further information on indicator variables. Variable cannot be a DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.
row position	Integer value or host variable that contains an integer value. The integer value gives the position of the desired row in the active set of rows. See "FETCH with a Scroll Cursor" on page 1-412 for a discussion of the RELATIVE and ABSOLUTE keywords and the meaning of *row position* with each keyword.	A value of 0 for *row position* is allowed with the RELATIVE keyword. A value of 0 fetches the current row. The value of *row position* must be 1 or higher with the ABSOLUTE keyword.	If you are using a host variable, variable name must conform to language-specific rules for variable names. If you are using a literal number, see Literal Number, p. 1-997.
sqlda pointer	Pointer to an **sqlda** structure that receives the values from the fetched row	You cannot begin an **sqlda** pointer with a dollar sign ($) or a colon (:).	See the discussion of **sqlda** structure in the *INFORMIX-ESQL/C Programmer's Manual*.

(2 of 2)

Usage

The FETCH statement is one of four statements that are used for queries that return more than one row from the database. The four statements, DECLARE, OPEN, FETCH, and CLOSE, are used in the following sequence:

1. Declare a select or function cursor to control the active set of rows.
2. Open the cursor to begin execution of the query.

3. Fetch from the cursor to retrieve the contents of each row.
4. Close the cursor to break the association between the cursor and the active set.

You can declare a select or function cursor with either of the following cursor characteristics: a sequential cursor or a scroll cursor. The way the database server creates and stores members of the active set and then fetches rows from the active set differs depending on whether the cursor is a sequential cursor or a scroll cursor. (For more information, see "Cursor Characteristics" on page 1-313 in the DECLARE statement)

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 select or function cursor can fetch only the next row in sequence from the active set. The sole cursor-direction 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 example in INFORMIX-ESQL/C illustrates a FETCH statement with a sequential cursor:

```
EXEC SQL fetch 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

A scroll select or function 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	Effect
NEXT	Retrieves the next row in the active set.
PREVIOUS	Retrieves the previous row in the active set.
PRIOR	Is synonymous with PREVIOUS; it retrieves the previous row in the active set.
FIRST	Retrieves the first row in the active set.
LAST	Retrieves the last row in the active set.
CURRENT	Retrieves the current row in the active set (the same row as returned by the preceding FETCH statement from the scroll cursor).
RELATIVE	Retrieves the *n*th row, relative to the current cursor position in the active set, where *row position* supplies *n*. A negative value indicates the *n*th row prior to the current cursor position. If *row position* is 0, the current row is fetched.
ABSOLUTE	Retrieves the *n*th row in the active set, where *row position* supplies *n*. Absolute row positions are numbered from *1*.

The following INFORMIX-ESQL/C examples illustrate a 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;
```

Row Numbers

The row numbers that are used with the ABSOLUTE keyword are valid only while the cursor is open. Do not confuse them 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 number for a FETCH statement is based on the position of the row in the 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 be sure which row the program asks for next. When a scroll cursor opens, the database server implements the active set as a temporary table although it might not fill 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 a user-defined function must be returned into a memory location. You can specify these destinations in one of the following ways:

- In the INTO clause of a SELECT statement
- In the INTO clause of a EXECUTE FUNCTION statement
- In the INTO clause of a FETCH statement
- In a system-descriptor area
- In an **sqlda** structure ♦

E/C

INTO Clause of SELECT

When you associate a SELECT statement with the cursor (a select cursor), the SELECT can contain an INTO clause to specify the program variables that are to receive the column values. In this case, the FETCH statement *cannot* contain an INTO clause. You can use this method only when the SELECT statement is written as part of the declaration of a cursor (see the DECLARE statement on page 1-300). The following example uses the INTO clause of the SELECT statement to specify program variables in INFORMIX-ESQL/C:

```
EXEC SQL declare ord_date cursor for
    select order_num, order_date, po_num
        into :o_num, :o_date, :o_po FROM orders;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

Use an indicator variable if the data that is returned from the SELECT statement might be null. See your SQL API manual for more information about indicator variables.

If you prepare a SELECT statement, the SELECT *cannot* include the INTO clause so you must use the INTO clause of the FETCH statement. For more information, see page 1-415.

INTO Clause of EXECUTE FUNCTION

When you associate an EXECUTE FUNCTION statement with the cursor (a function cursor), the EXECUTE FUNCTION can contain an INTO clause to specify the program variables that are to receive the return values. In this case, the FETCH statement *cannot* contain an INTO clause. You can use this method only when the EXECUTE FUNCTION statement is written as part of the declaration of a cursor (see the DECLARE statement on page 1-300).

Tip: *In earlier versions of Informix products, you could use the EXECUTE PROCEDURE statement to execute a stored procedure that returned values (an SPL function). For backward compatibility, the EXECUTE PROCEDURE statement still supports the INTO clause to receive return values. However, Informix recommends that new SPL functions use EXECUTE FUNCTION and the INTO clause. For more information, see "EXECUTE PROCEDURE" on page 1-404.*

The following example uses the INTO clause of the EXECUTE FUNCTION statement to specify program variables in INFORMIX-ESQL/C:

```
EXEC SQL declare ord_date cursor for
    execute function func1(20)
        into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

Use an indicator variable if the data that is returned from the EXECUTE FUNCTION statement might be null. See *INFORMIX-ESQL/C Programmer's Manual* for more information about indicator variables.

INTO Clause of FETCH

When the SELECT or EXECUTE FUNCTION statement omits the INTO clause, you must specify the destination of the data whenever a row is fetched. For example, to dynamically execute a SELECT or EXECUTE FUNCTION statement, the SELECT or EXECUTE FUNCTION 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.

In the following INFORMIX-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'stores7';
    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;
    }
```

You can fetch into a program-array element only by using an INTO clause in the FETCH statement. When you are declaring a cursor, do not refer to an array element within the SQL statement.

Using a System-Descriptor Area

If you do not know the number of return values or their data types that a SELECT or EXECUTE FUNCTION statement returns at runtime, you can store output values in a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more return values.

X/O

You can also use an **sqlda** structure to dynamically supply parameters (page 1-417). However, a system-descriptor area conforms to the X/Open standards. ♦

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 page 1-415.*

To specify a system-descriptor area as the location of output values, use the USING SQL DESCRIPTOR clause of the FETCH statement. This clause introduces 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.

The following example shows the 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';
    ...
```

The COUNT field in the system-descriptor area corresponds to the number of return values of the prepared statement. 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 on how to use a system-descriptor area, see the *INFORMIX-ESQL/C Programmer's Manual*.

Using an sqlda Structure

If you do not know the number of return values or their data types that a SELECT or EXECUTE FUNCTION statement returns at runtime, you can store output values in an **sqlda** structure. An **sqlda** structure lists the data type and memory location of one or more return values.

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 page 1-415.*

To specify an **sqlda** structure as the location of parameters, follow these steps:

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 the FETCH statement to name the **sqlda** structure as the location into which you fetch the return 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 return values from the prepared statement. For further information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*.

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 to keep it from changing 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 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 it, the lock might or might not be released when you fetch another row, depending on the isolation level you have set. The lock on an unchanged row is released as soon as another row is fetched, unless you are using Repeatable Read isolation (see the SET ISOLATION statement on page 1-719).

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.

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. Once you open the collection cursor with the OPEN statement, the cursor allows you to access the elements of the **collection** variable.

For more information, see the Collection Derived Table segment on page 1-827. For more information on how to declare a collection cursor for a SELECT statement, see "A Select Cursor for a Collection Variable" on page 1-318.

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_colors  SET(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;
```

Once you have fetched 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.

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 collection column with the INSERT or UPDATE 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 information. See the GET DIAGNOSTICS statement in this manual for more information.

Tip: *When you encounter an **SQLSTATE** (**sqlca.sqlcode**) error, a corresponding **SQLCODE** error might also exist. The **SQLCODE** variable contains the Informix-specific error code. For more information about **SQLCODE**, see the "Informix Guide to SQL: Tutorial" and the "INFORMIX-ESQL/C Programmer's Manual."*

References

See the ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, and SET DESCRIPTOR statements in this manual for further information about using the FETCH statement with dynamic management statements.

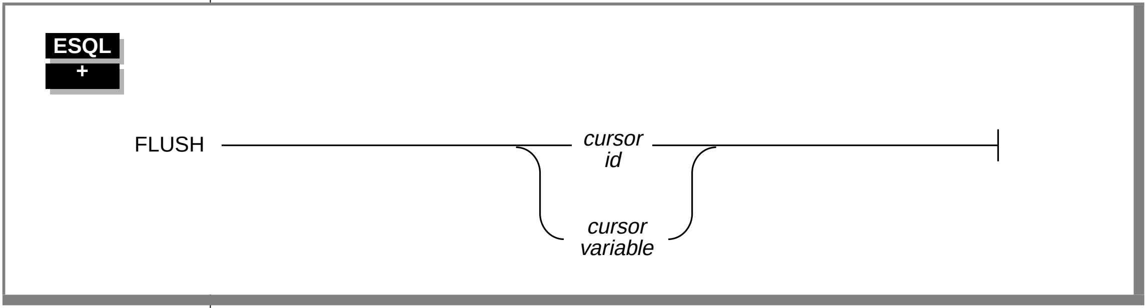
In the *Informix Guide to SQL: Tutorial*, see the discussion of the FETCH statement in Chapter 5.

For further information about error checking, the system-descriptor area and the **sqlda** structure, see *INFORMIX-ESQL/C Programmer's Manual*.

FLUSH

Use the FLUSH statement to force rows that a PUT statement buffered to be written to the database.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 1-962
cursor variable	Host variable that identifies a cursor	Host variable must be a character data type. A DECLARE statement must have previously created the cursor.	Variable name must conform to language-specific rules for variable names.

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.

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 **SQLCODE** variable (**sqlca.sqlcode**) and the **sqlerrd[2]** field of the **sqlca** structure.

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 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 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[2]** 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 might exist. Check the GET DIAGNOSTICS statement for information about getting the* ***SQLSTATE*** *value and using the GET DIAGNOSTICS statement to interpret the* ***SQLSTATE*** *value.*

Counting Total and Pending Rows

To count the number of rows actually inserted into the database as well as the number not yet inserted, perform the following steps:

1. Prepare two integer variables, such as **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**.

References

See the CLOSE, DECLARE, OPEN, and PUT statements in this manual.

For information about the **sqlca** structure, see the *INFORMIX-ESQL/C Programmer's Manual*.

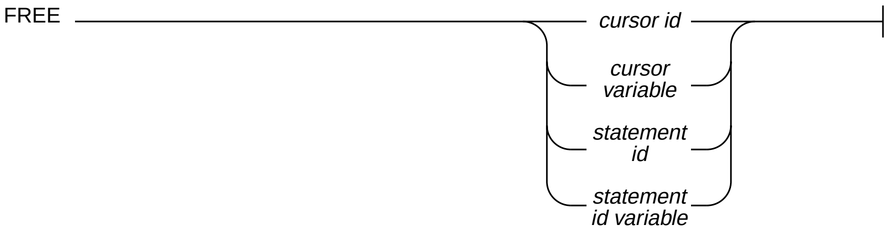
In the *Informix Guide to SQL: Tutorial*, see the discussion of FLUSH in Chapter 6.

FREE

The FREE statement releases resources that are allocated to a prepared statement or to a cursor.

Syntax

ESQL
+



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 1-962
cursor variable	Host variable that identifies a cursor	Variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Variable name must conform to language-specific rules for variable names

(1 of 2)

Element	Purpose	Restrictions	Syntax
statement id	Identifier for an SQL statement	The statement identifier must be defined in a previous PREPARE statement. After you release the database-server resources, you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.	PREPARE, p. 1-538
statement id variable	A host variable that identifies an SQL statement	This variable must be defined in a previous PREPARE statement. Variable must be a character data type.	PREPARE, p. 1-538

(2 of 2)

Usage

The FREE statement releases the resources that were allocated for a prepared statement or a declared cursor in the application-development tool and the database server. Resources are allocated when you prepare a statement or when you open a cursor (see the DECLARE and OPEN statements on pages 1-300 and 1-525, respectively.)

The amount of available memory in the system limits the total number of open cursors and prepared statements that are allowed at one time in one process. Use FREE *statement id* or FREE *statement id variable* to release the resources that a prepared statement holds; use FREE *cursor id* or FREE *cursor variable* to release resources that a cursor holds.

Freeing a Statement

If you prepared a statement (but did not declare a cursor for it), FREE *statement id* (or *statement id variable*) releases the resources in both the application development tool and the database server.

If you declared a cursor for a prepared statement, FREE *statement id* (or *statement id variable*) 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.

After you free a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following INFORMIX-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 INFORMIX-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;
```

Freeing a Cursor

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 *statement id variable*).

If a cursor is not declared for a prepared statement, freeing the cursor 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 second example in "Freeing a Statement" on page 1-427.

References

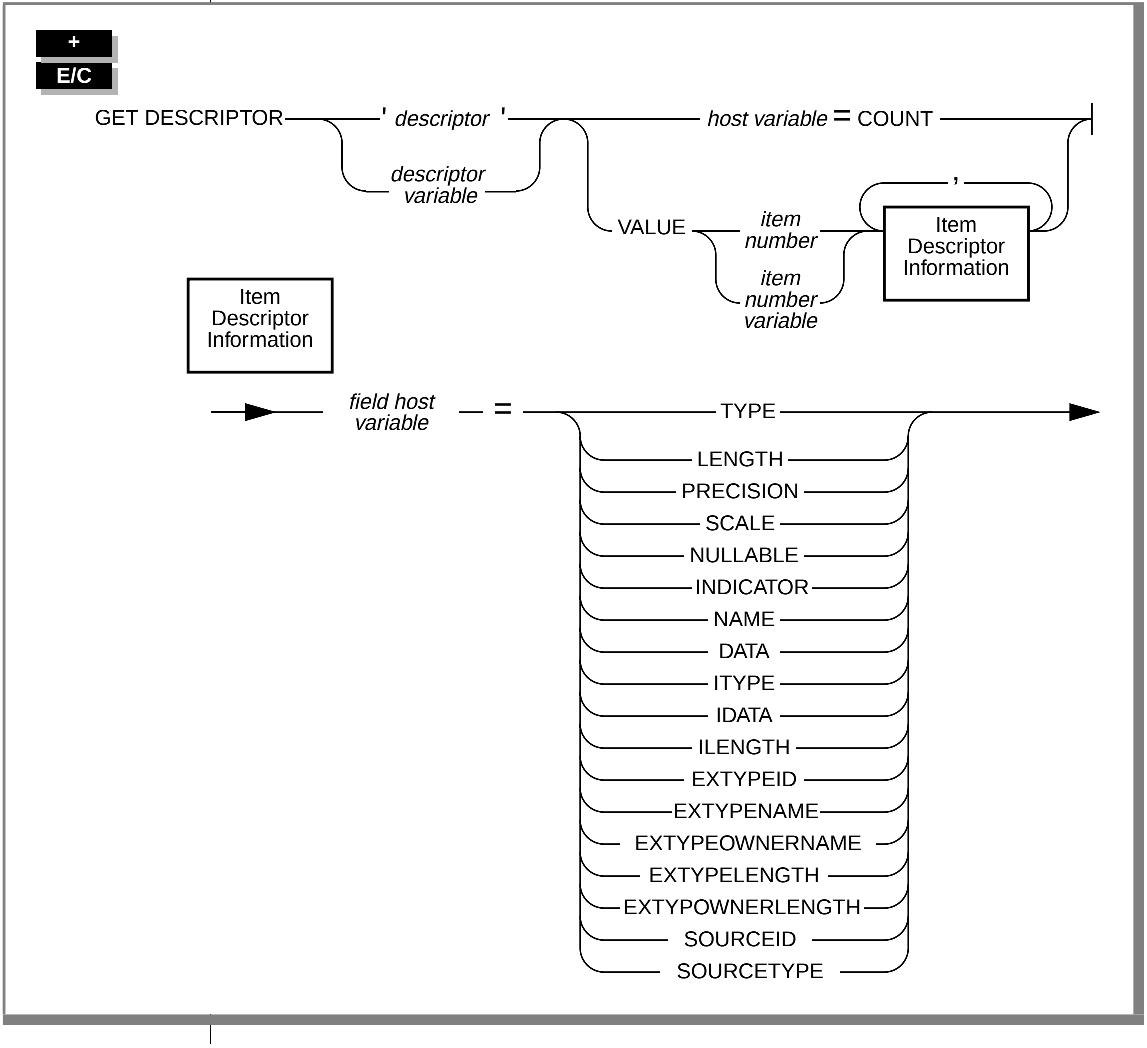
See the CLOSE, DECLARE, EXECUTE, EXECUTE IMMEDIATE, and PREPARE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the FREE statement in Chapter 5.

GET DESCRIPTOR

Use the GET DESCRIPTOR statement to obtain values from a system-descriptor area.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	A quoted string that identifies a system-descriptor area from which information is to be obtained	The system-descriptor area must have been allocated in an ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-1010
descriptor variable	An embedded variable name that holds the value of *descriptor*	The system-descriptor area identified in *descriptor variable* must have been allocated in an ALLOCATE DESCRIPTOR statement.	The name of the embedded variable must conform to language-specific rules for variable names.
field host variable	The name of a host variable that receives the contents of the specified field from the system-descriptor area	The *field host variable* must be an appropriate type to receive the value of the specified field from the system-descriptor area	The name of the *field host variable* must conform to language-specific rules for variable names.
host variable	The name of a *host variable* that indicates how many values are described in the system-descriptor area	The host variable must be an integer data type.	The name of the *host variable* must conform to language-specific rules for variable names.
item number	An unsigned integer that represents one of the occurrences (item descriptors) in the system-descriptor area	The value of *item number* must be greater than zero and less than the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-997
item number variable	The name of a host variable that holds the value of *item number*	The *item number variable* must be an integer data type.	The name of the *item number variable* must conform to language-specific rules for variable names.

Usage

The GET DESCRIPTOR statement can be used after you have described SELECT, EXECUTE FUNCTION, and INSERT statements with the DESCRIBE...USING SQL DESCRIPTOR statement. The GET DESCRIPTOR statement can obtain values from a system-descriptor area in the following instances:

- Determine how many values are described in a system-descriptor area by retrieving the value in the COUNT field.
- 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.

If an error occurs during the assignment to any identified host variable, the contents of the host variable are undefined. The host variables that are used in the GET DESCRIPTOR statement must be declared in the INFORMIX-ESQL/C program. See the *INFORMIX-ESQL/C Programmer's Manual* for information on the role and contents of each field in the system-descriptor area and on how to declare host variables.

Using the COUNT Keyword

Use the COUNT keyword to determine how many values are described in the system-descriptor area. The following INFORMIX-ESQL/C example shows how to use a GET DESCRIPTOR statement with a host variable to determine how many values 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 occurrences 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;
...
}
```

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. You can modify values for items after you use the DESCRIBE statement to fill the fields for a SELECT, EXECUTE FUNCTION, or INSERT statement, or you can obtain values for items for which you are providing a description (such as parameters in a WHERE clause).

The *item number* must be greater than zero and less than the number of occurrences that were specified when you allocated the system-descriptor area with the ALLOCATE DESCRIPTOR statement.

Using the VALUE Clause After a DESCRIBE

After you describe a SELECT, EXECUTE FUNCTION, or INSERT statement with the DESCRIBE...USING SQL DESCRIPTOR 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 statement, or the characteristics of each column in the INSERT statement are returned to the system-descriptor area. Each value in the system-descriptor area describes the characteristics of one returned column or expression. Each field and its possible contents are described in the *INFORMIX-ESQL/C Programmer's Manual*.

The following INFORMIX-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. The codes for the TYPE field are listed in the description of the SET DESCRIPTOR statement on page 1-699.

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 Informix products, executing a GET DESCRIPTOR statement can result in an error.

In X/Open mode, a warning message appears if ILENGTH, DATA, or ITYPE is used. It indicates that these fields are not standard X/Open fields for a system-descriptor area.

For more information about TYPE, ILENGTH, IDATA, and ITYPE, see the dynamic management chapter in the *INFORMIX-ESQL/C Programmer's Manual*. For more information about programming in X/Open mode, see the preprocessing and compilation syntax in the appropriate Informix SQL API programmer's manual. ♦

If the TYPE field for 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 INFORMIX-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 dynamically the length of a string in the DATA or IDATA field.

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.

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 value 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 value is the length, in bytes, of the name of the opaque type.

- 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 value is the length, in bytes, of the owner name for the opaque type.

Use these field names with the GET DESCRIPTOR statement to obtain information about an opaque column. For more information on the **sysxtdtypes** system catalog table, see Chapter 1 of the *Informix Guide to SQL: Reference*.

Describing a Distinct-Type Column

The DESCRIBE statement sets the following item-descriptor fields when the column to fetch has an 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 type you are setting. 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 **sqltypes.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 (page 1-699). 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.

Use these field names with the GET DESCRIPTOR statement to obtain information about a distinct-type column. For more information on the **sysxtdtypes** system catalog table, see Chapter 1 of the *Informix Guide to SQL: Reference*.

References

See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for more information about using dynamic SQL statements.

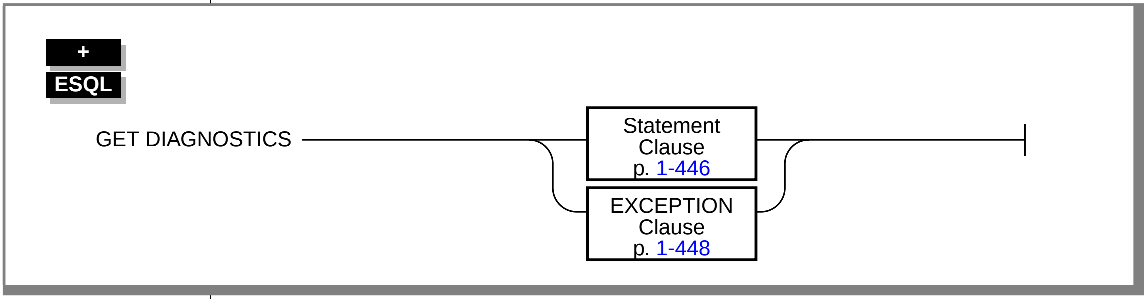
For more information about the system-descriptor area, see the *INFORMIX-ESQL/C Programmer's Manual*.

GET DIAGNOSTICS

Use the GET DIAGNOSTICS statement to return diagnostic information about executing an SQL statement. The GET DIAGNOSTICS statement uses one of two clauses, as the following list describes:

- The Statement clause determines count and overflow information about errors and warnings generated by the most recent SQL statement.
- The EXCEPTION clause provides specific information about errors and warnings generated by the most recent SQL statement.

Syntax



Usage

The GET DIAGNOSTICS statement retrieves selected status information from the diagnostics area and retrieves either count and overflow information or information on a specific exception. The diagnostics area is a data structure that stores diagnostic information about an executed SQL statement.

The GET DIAGNOSTICS statement never changes the contents of the diagnostics area.

Using the SQLSTATE Status Code

When an SQL statement executes, the database server sets a status code is automatically generated. This code represents one of the following exceptions:

- *Success* to indicate that the SQL executed without exceptions
- *Warning* to indicate that the SQL statement executed successfully but the database server encountered some condition that might limit the statement
- *End of Data* or *Not Found* to indicate that the SQL statement executed successfully but it either found no matching rows (Not Found) or it did not operate on a row (End of Data)
- *Error* to indicate that the SQL statement did not execute successfully

The database server stores this status code in a variable called **SQLSTATE**.

X/O
ANSI

The **SQLSTATE** status variable conforms to the ANSI and X/Open standards. ♦

Tip: *Informix database servers also store a status code in an Informix-specific variable called **SQLCODE** and exception information in the SQLCA structure. For more information, see the "INFORMIX-ESQL/C Programmer's Manual."*

Class and Subclass Codes

The **SQLSTATE** status code is a a five-character string that can contain only digits and capital letters. This string has the following two parts:

- The first two characters of the **SQLSTATE** status code indicate a *class*.
- The last three characters of the **SQLSTATE** status code indicate a *subclass*.

Figure 1-1 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 `server rejected the connection`.

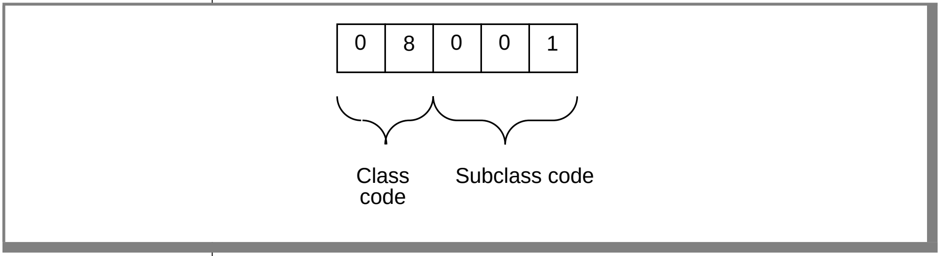


Figure 1-1
The Structure of the SQLSTATE Status Code

The following table is a quick reference for interpreting class code values.

SQLSTATE Class Code Value	Outcome
00	Success
01	Success with warning
02	End of data not found
> 02	Errors

Support for ANSI Standards

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 a `I` are Informix-specific warning messages.
- **SQLSTATE** code with a class code of `01` and a subclass code of `U01` indicates that the a user-defined routine has returned a warning message that has been defined by a user-defined routine.
- **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 a capital letter in the range I to Z indicate conditions that are currently undefined by ANSI. The only exception is that **SQLSTATE** codes whose class code is `IX` are Informix-specific error messages.
- **SQLSTATE** code of `U0001` indicates that a user-defined routine has returned an error message that has been defined by a user-defined routine.

List of SQLSTATE Codes

The following table describes the class codes, subclass codes, and the meaning of all valid warning and error codes associated with the **SQLSTATE** status 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	INFORMIX-Universal Server database selected
01	I05	Float to decimal conversion has been used
01	I06	Informix extension to ANSI-compliant standard syntax
01	I07	UPDATE/DELETE statement does not have a WHERE clause
01	I08	An ANSI keyword has been used as a cursor name
01	I09	Number of items in the select list is not equal to the number in the into list
01	I10	Database server running in secondary mode
01	I11	Dataskip is turned on
01	U01	User-defined routine has defined the warning message text
02	000	No data found

(1 of 4)

Class	Subclass	Meaning
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
07	009	Invalid descriptor index
08	000	Connection exception
08	001	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

(2 of 4)

Class	Subclass	Meaning
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
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
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

(3 of 4)

Class	Subclass	Meaning
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
U0	001	User-defined routine has defined the error message text

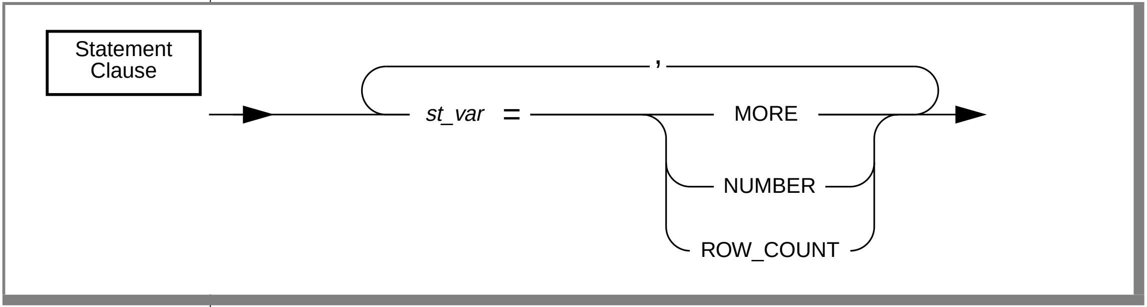
(4 of 4)

Using SQLSTATE in Applications

You can use a variable, called **SQLSTATE**, that you do not have to declare in your program. **SQLSTATE** contains the status code that is generated every time your program executes an SQL statement. This status code is essential for exception handling. 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 from the diagnostics area.

For an example of how to use an **SQLSTATE** variable in a program, see "Using GET DIAGNOSTICS for Error Checking" on page 1-456.

Statement Clause



Element	Purpose	Restrictions	Syntax
st_var	Host variable that receives status information about the most recent SQL statement. It receives information for the specified status field name.	Data type must match that of the requested field.	Variable name must conform to language-specific rules for variable names.

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. These three fields are represented by the following keywords.

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 it detected in the diagnostics area.

  The GET DIAGNOSTICS statement returns a value of `N`.
- Detected more exceptions than it stored in the diagnostics area.

  The GET DIAGNOSTICS statement returns a value of `Y`.

The value of MORE is always `N`.

Using the NUMBER Keyword

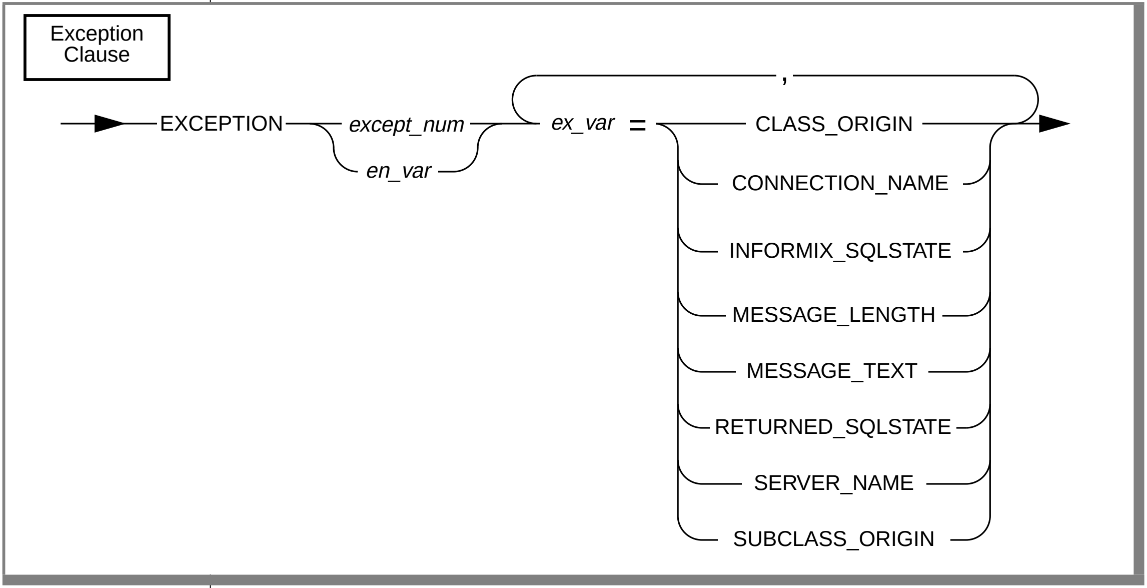
Use the NUMBER keyword to count the number of exceptions that the most recently executed SQL statement placed into the diagnostics area. The **NUMBER** field can hold a value from 1 to 35,000, to indicate how many exceptions are counted.

Using the ROW_COUNT Keyword

Use the ROW_COUNT keyword to count the number of rows the most recently executed statement processed. The **ROW_COUNT** field counts the following number of rows:

- Inserted into a table
- Updated in a table
- Deleted from a table

EXCEPTION Clause



Element	Purpose	Restrictions	Syntax
en_var	Host variable that specifies an exception number for a GET DIAGNOSTICS statement	Variable must contain an integer value limited to a range from 1 to 35,000. Variable data type must be INT or SMALLINT.	Variable name must conform to language-specific rules for variable names.
except_num	Literal integer value that specifies the exception number for a GET DIAGNOSTICS statement. The *except_num* literal indicates one of the exception values from the number of exceptions returned by the NUMBER field in the Statement clause.	Integer value is limited to a range from 1 to 35,000.	Literal Number, p. 1-997
ex_var	Host variable that you declare, which receives EXCEPTION information about the most recent SQL statement. Receives information for a specified exception field name.	Data type must match that of the requested field.	Variable name must conform to language-specific rules for variable names.

When GET DIAGNOSTICS retrieves exception information, it deposits 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 seven exception information fields are represented by the keywords described in the following table.

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]
INFORMIX_SQLCODE	Integer	SQLCODE value	long int
MESSAGE_TEXT	Character	String	char[8191]
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, *except_num*, or an integer host variable (an exact numeric with a scale of 0), *en_var*. 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 illegal 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

Use the RETURNED_SQLSTATE keyword to determine the **SQLSTATE** value that describes the exception.

Using the INFORMIX_SQLCODE Keyword

Use the INFORMIX_SQLCODE keyword to retrieve the value of the Informix-specific status code (**SQLCODE**) for the associated **SQLSTATE** (and **RETURNED_SQLSTATE**) value. The *Informix Error Messages* manual describes Informix-specific codes.

Using the CLASS_ORIGIN Keyword

Use the CLASS_ORIGIN keyword to retrieve the source of the class portion of the **RETURNED_SQLSTATE** (and **SQLSTATE**) value. Possible class origins include the following:

- If the International Standards Organization (ISO) standard defines the class, the value of **CLASS_ORIGIN** is equal to '`ISO 9075`'. ANSI SQL and ISO SQL are synonymous.
- If Informix defines the class value, the value of **CLASS_ORIGIN** is equal to '`IX000`'.
- If a user-defined routine has defined the message in the **MESSAGE_TEXT** field, the value of **CLASS_ORIGIN** is '`U0001`'. A routine can be an SPL routine or an external routine.

Using the SUBCLASS_ORIGIN Keyword

Use the SUBCLASS_ORIGIN keyword to define the source of the subclass portion of the **RETURNED_SQLSTATE** (and **SQLSTATE**) value. Possible subclass origins include the following:

- If the International Standards Organization (ISO) standard defines the subclass value, the value of **SUBCLASS_ORIGIN** is equal to 'ISO 9075'. ANSI SQL and ISO SQL are synonymous.
- If Informix defines the subclass value, the value of **SUBCLASS_ORIGIN** is equal to 'IX000'.
- If a user-defined routine has defined the message in the **MESSAGE_TEXT** field, the value of **SUBCLASS_ORIGIN** is 'U0001'. A routine can be an SPL routine or an external routine.

Using the MESSAGE_TEXT Keyword

Use the MESSAGE_TEXT keyword to determine the message text of the exception (for example, an error message). User-defined routines (such as stored routines and external routines) can define their own message text, which you can access text through the **MESSAGE_TEXT** field.

The following values indicate that a user-defined routine has returned a message that the routine has defined:

- The **RETURNED_SQLSTATE** field and **SQLSTATE** are as follows:
  - ❑ "01U01": when the user-defined routine returns a user-defined *warning* message
  - ❑ "U0001": when the user-defined routine returns a user-defined *error* message
- The **CLASS_ORIGIN** and **SUBCLASS_ORIGIN** fields are set to "U0001".

Using the MESSAGE_LENGTH Keyword

Use the MESSAGE_LENGTH keyword to determine the length of the current string in the **MESSAGE_TEXT** field.

Using the SERVER_NAME Keyword

Use the SERVER_NAME keyword to determine the name of the database server associated with the actions of a CONNECT or DATABASE statement.

When the SERVER_NAME Field Is Updated

The GET DIAGNOSTICS statement updates the **SERVER_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.
- A DISCONNECT ALL statement fails.

When the SERVER_NAME Field Is Not Updated

The **SERVER_NAME** field is not updated when:

- a CONNECT statement fails.
- a DISCONNECT statement fails (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	It 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	It contains the name of the database server to which you switch or fail to switch.
DISCONNECT	It 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	It sets the field to blank if the statement executes successfully. If the statement does not execute successfully, the **SERVER_NAME** field contains the names of all the database servers from which you did not disconnect. However, this information does not mean that the connection still exists.

If the CONNECT statement is successful, the **SERVER_NAME** field is set to one of the following values:

- The **INFORMIXSERVER** value if the connection is to a default database server (that is, the CONNECT statement does not list a database server)
- The name of the specific 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 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 statements.

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 CONNECTION_NAME Is Not Updated

The **CONNECTION_NAME** field is not updated when the following situations occur:

- A CONNECT statement fails.
- A DISCONNECT statement fails (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.

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	It contains the name of the connection, specified in the CONNECT statement, to which you connect or fail to connect. The field is blank if you do not have a current connection or if you make a default connection.
SET CONNECTION	It contains the name of the connection, specified in the CONNECT statement, to which you switch or fail to switch.
DISCONNECT	It contains the name of the connection, specified in the CONNECT statement, 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 **CONNECTION_NAME** field remains unchanged.
DISCONNECT ALL	The **CONNECTION_NAME** field is blank 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 the CONNECT is successful, the **CONNECTION_NAME** field is set to the following 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

The DATABASE Statement

When you execute a DATABASE statement, the **CONNECTION_NAME** field is blank.

Using GET DIAGNOSTICS for Error Checking

The GET DIAGNOSTICS statement returns information held in various fields of the diagnostic area. For each field in the diagnostic area that you want to access, you must supply a host variable with a compatible data type.

The following examples illustrate using the GET DIAGNOSTICS statement to display error information. The first 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[8191];
    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");
    }
```

References

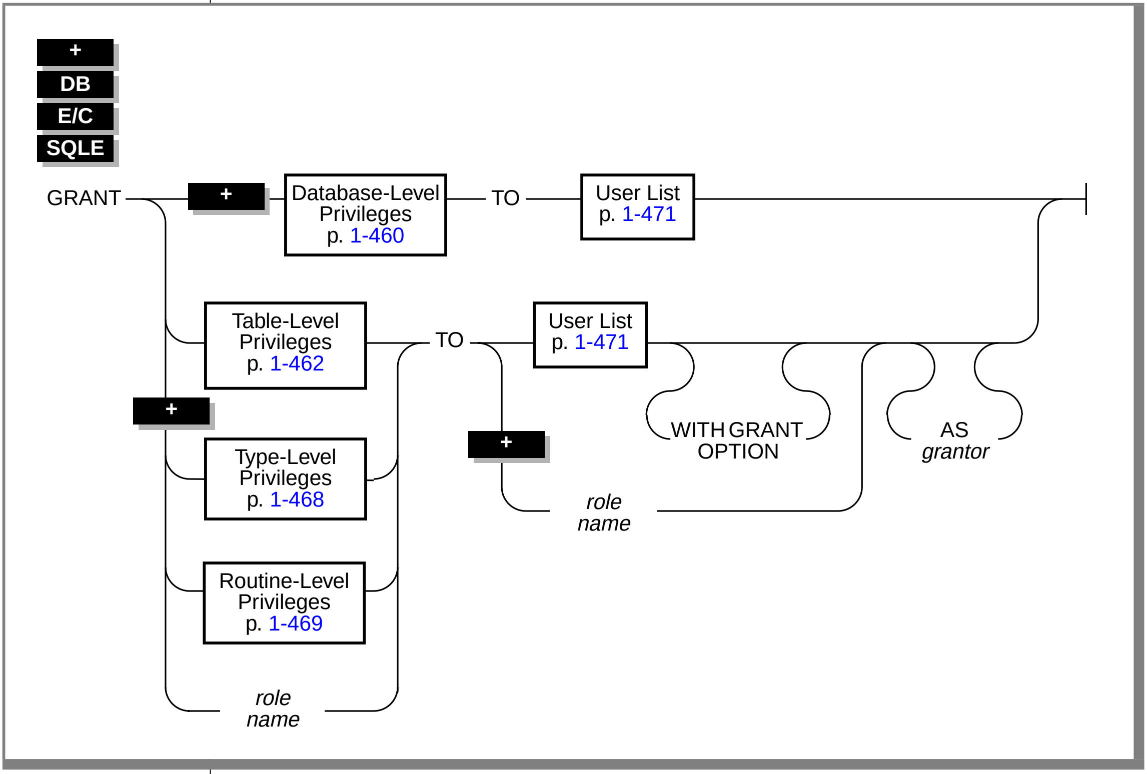
In Chapter 5 of the *Informix Guide to SQL: Tutorial*, see the discussion about error-code handling. In addition, refer to the exception-handling chapter of the *INFORMIX-ESQL/C Programmer’s Manual*.

GRANT

Use the GRANT statement to:

- authorize others to use, develop, or administrate a database that you create.
- allow others to view, alter, or drop a table, synonym, or view that you create.
- allow others to use a data type or execute a routine that you create.
- give a role name and its privileges to one or more users.

Syntax



+
DB
E/C
SQLE
GRANT
+
Database-Level Privileges p. 1-460
TO
User List p. 1-471
Table-Level Privileges p. 1-462
TO
User List p. 1-471
+
Type-Level Privileges p. 1-468
+
WITH GRANT OPTION
AS grantor
role name
Routine-Level Privileges p. 1-469
role name

Element	Purpose	Restrictions	Syntax
role name	A name that identifies users by their function. Use GRANT to: ■ give privileges to a role name. ■ specify the users who can use the privileges granted to the role.	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-962
grantor	A name that identifies who can REVOKE the effects of the current GRANT. By default, the login of the person who issues the GRANT statement identifies the grantor. To override the default, include the AS keyword followed by the login of your appointed grantor.	If you specify someone else as the grantor of the specified privilege, you cannot later revoke that privilege.	Identifier, p. 1-962

Usage

The GRANT statement extends privileges to other users that would normally accrue only to the DBA or to the creator of an object. Later GRANT statements do not affect privileges already granted to a user.

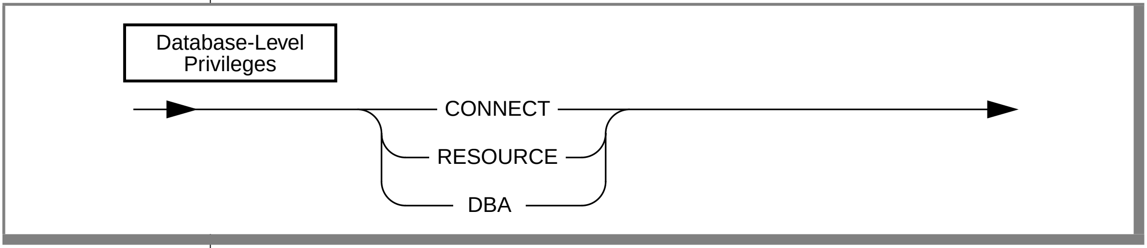
You can grant privileges to a previously created role. You can grant a role to individual users or to another role.

Privileges you grant remain in effect until you cancel them with a REVOKE statement. Only the grantor of a privilege or a DBA can revoke that privilege. The grantor is normally the person who issues the GRANT statement. To transfer the right to revoke, name another user as grantor when you issue a GRANT statement.

The keyword PUBLIC extends a GRANT to all users. If you want to restrict privileges to a particular user that **public** already has, you must first revoke the right of **public** to those privileges.

Database-Level Privileges

When you create a database with the CREATE DATABASE statement, you are the owner. As the database owner, you 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 1-462.

Database access levels are, from lowest to highest, Connect, Resource, and DBA. Use the corresponding keyword to grant a level of access privilege.

Privilege	Permissible Tasks
CONNECT	Any user with the Connect privilege can perform the following tasks: ■ 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 A user with the Connect privilege and appropriate table-level privileges can also do the following: ■ Alter or drop a table or index ■ Grant table-level privileges

(1 of 2)

Privilege	Permissible Tasks
RESOURCE	Gives you the ability to 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 tasks: ■ Create new tables ■ Create new indexes ■ Create new routines ■ Create new data types
DBA	Has all the capabilities of the Resource privilege as well as the ability to perform the following tasks: ■ Grant any database-level privilege, including the DBA privilege, to another user ■ Grant any table-level privilege to another user ■ Grant any table-level privilege to a role ■ Grant a role to a user or to another role ■ Execute the SET SESSION AUTHORIZATION statement ■ Use the NEXT SIZE keyword to alter extent sizes in the system catalog ■ Drop any object, regardless of its owner ■ Create tables, views, and indexes, and specify another user as owner of the objects ■ Restrict the Execute privilege to DBAs when registering a routine ■ Execute the DROP DATABASE statement ■ Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement ■ Insert, delete, or update rows of any system catalog table except **systables** Only user **informix** can update **systables**. You cannot grant this privilege.

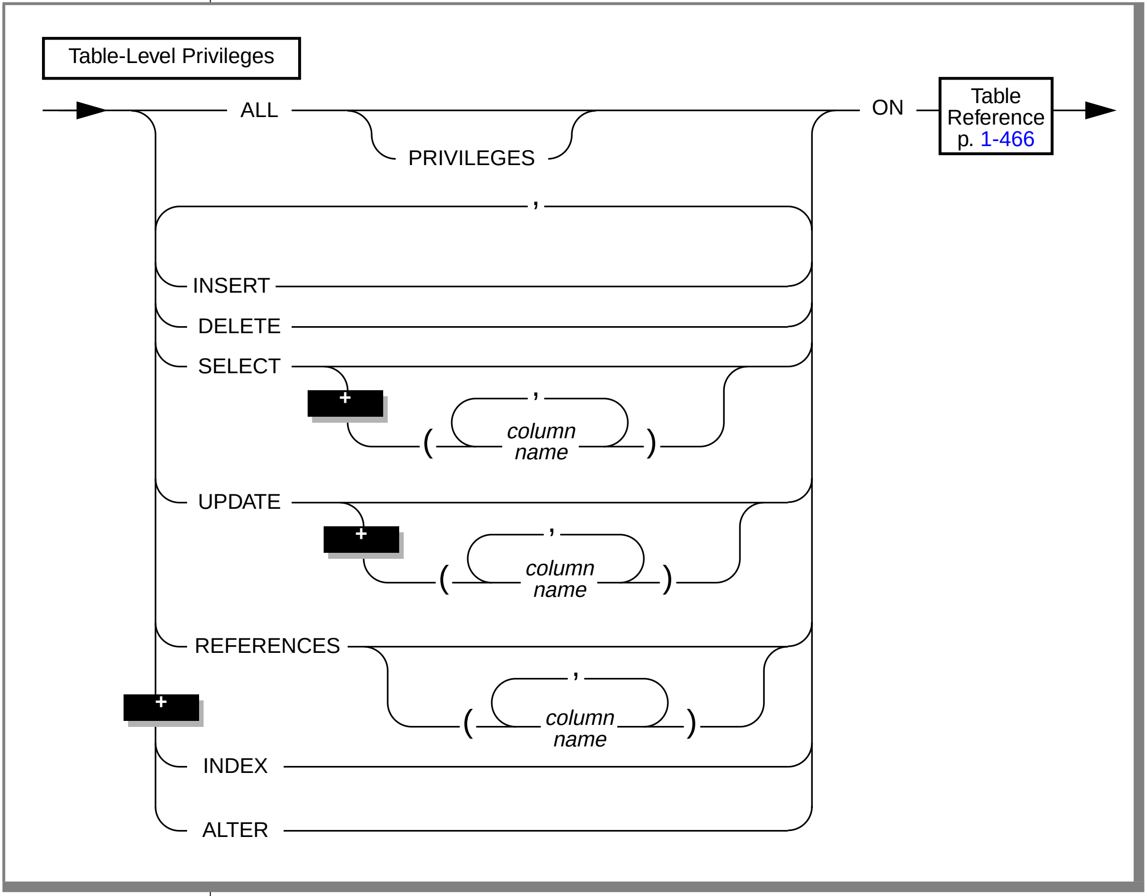
(2 of 2)

Warning: *Informix strongly recommends that you do not update, delete, or alter any rows in the system catalog tables. Modifying the system catalog tables can destroy the integrity of the database.*

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 table ownership to another user, but you can grant table-level privileges to another user or to a role.

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 name	The name of the column or columns to which a Select, Update, or References privilege is granted. If you omit *column name*, the privilege applies to all columns in the specified table.	The specified column or columns must exist.	Identifier, p. 1-962

The table that follows lists keywords for granting table-level privileges.

Privilege	Functions
INSERT	Enables the grantee to insert rows into a table, view, or synonym.
DELETE	Enables the grantee to delete rows from a table, view, or synonym.
SELECT	Enables the grantee to select and view data. By default, the grantee can specify any column names from your table in a SELECT statement or SELECT *. You can limit selection to only certain columns from your table if you explicitly list their column names in the GRANT SELECT statement.
UPDATE	Enables the grantee to change data. By default, the grantee can specify any column names from your table in an UPDATE statement. To enable the grantee to update only certain columns, explicitly list their column names in the GRANT UPDATE statement.
REFERENCES	Enables the grantee to: ■ reference columns of your table as foreign keys. ■ create constraints that cascade deletes to foreign keys. By default, a grantee can specify any column names from your table as a foreign key. To enable the grantee to reference only certain columns, explicitly list their column names in the GRANT REFERENCES statement. A grantee must also have the appropriate additional privileges to invoke the statement that creates, adds, or modifies a column to contain foreign keys. (For example, CREATE TABLE requires the Resource privilege.)

(1 of 2)

Privilege	Functions
INDEX	Enables the grantee to create permanent indexes on a table. Has no effect unless the grantee also has database-level Resource privilege.
ALTER	Enables the grantee to perform all the functions provided by the ALTER TABLE statement, such as: ■ add or delete columns. ■ modify column data types. ■ add or delete constraints on column values. ■ set the object mode of indexes, constraints, and triggers. ■ change the locking mode of the table from PAGE to ROW. ■ add or drop a corresponding row type name for your table. The Alter privilege has no effect unless the grantee also has the database-level Resource privilege. The Usage privilege is also required for any user-defined type effected by the ALTER TABLE statement.
ALL	Provides all table-level privileges with a single keyword. You can optionally use the longer form ALL PRIVILEGES. For some of the individual table-level privileges covered by ALL to take effect, the recipient needs additional authorization. (See "Behavior of the ALL Keyword" on page 1-465 for details.)

(2 of 2)

You can narrow the scope of a Select, Update, or References privilege by naming the specific columns to which the privilege applies.

Specify keyword PUBLIC as *user* if you want a GRANT statement to apply to all users.

Examples

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 shown in the following example:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
    ON customer TO PUBLIC
```

Behavior of the ALL Keyword

The ALL keyword grants all table-level privileges to the specified user. If any a grantee lacks additional privileges required to use a table-level privilege, the GRANT statement with the ALL keyword succeeds, but the following SQLSTATE warning is returned:

```
01007 - Privilege not granted.
```

For example, assume that the 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:

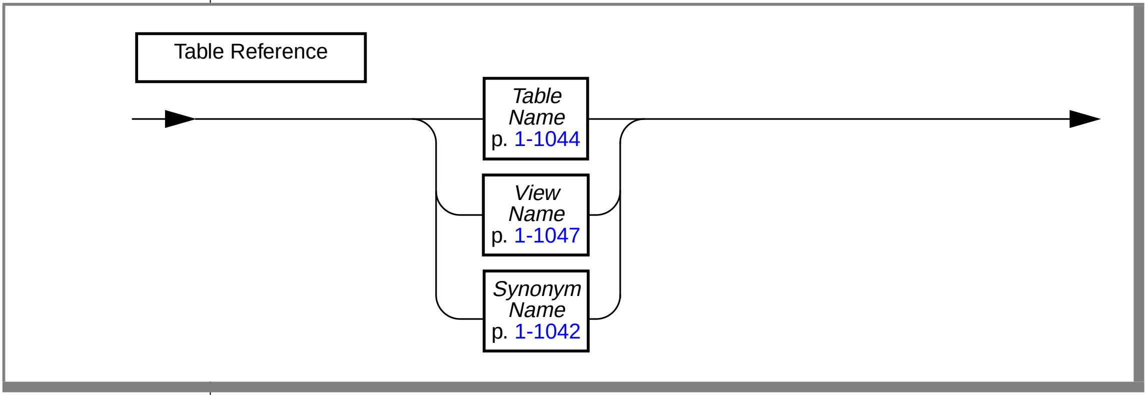
```
GRANT ALL ON customer TO tania
```

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.** User **ted** has those privileges and the right to grant those privileges to other users.
- User **ted** can not grant the Delete, Update, References, Index, and Alter privileges because he does not have them. User **tania** does not have them either.

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.



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

If you create a table in an ANSI-compliant database, only you, as table owner, have any table-level privileges until you explicitly grant privileges 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. 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 1-286.

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.

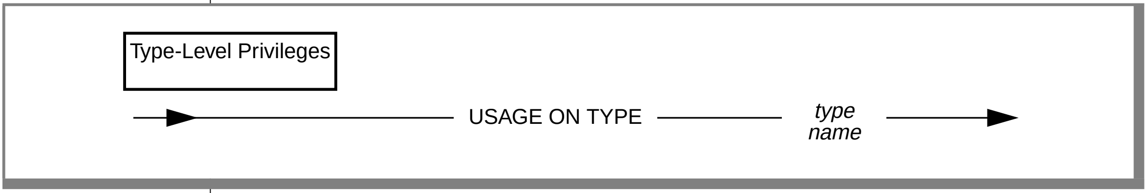
You can grant (or revoke) privileges on a view only if you are the owner of the underlying tables or if you received these privileges on the table with the right to grant them (the WITH GRANT OPTION keyword). You must explicitly grant those privileges within your authority; **public** does not automatically 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, or References privileges on a view (or the All privilege because All includes Index, References, and Alter).

Type-Level Privileges

You own a user-defined data type 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 data of your user-defined type.



Element	Purpose	Restrictions	Syntax
type name	The name of the user-defined data type to which the Usage privilege is granted	The specified data type must exist.	Data Type, p. 1-855

Without a GRANT statement, any user can create SQL statements that contain built-in data types. By 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.

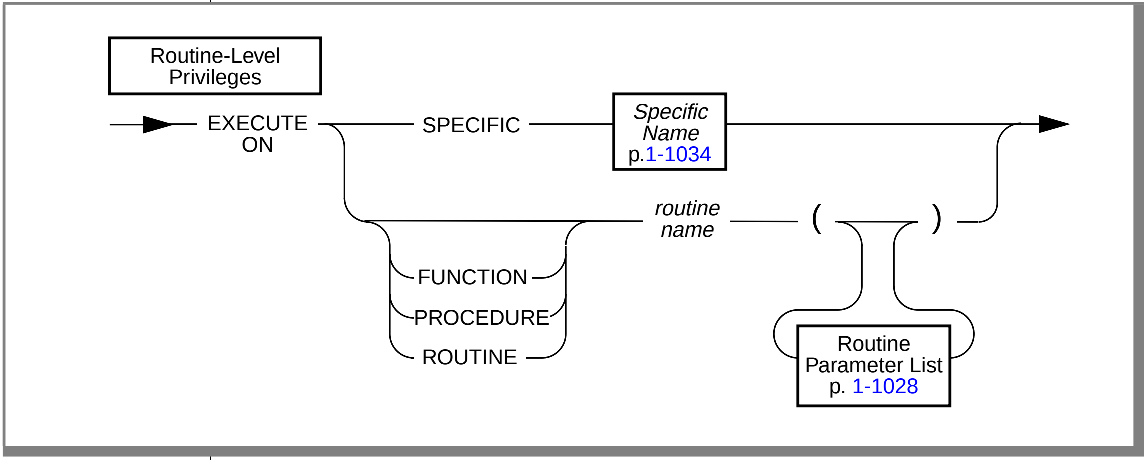
For more information about user-defined types, see CREATE OPAQUE TYPE and CREATE DISTINCT TYPE in this manual, the Chapter 2, "Data Types" in the *Informix Guide to SQL: Reference*, and Chapter 10, "Understanding Complex Data Types" in the *Informix Guide to SQL: Tutorial*.

Routine-Level Privileges

The generic term *user-defined routine* refers to both a user-defined function and a user-defined procedure. A function returns one or more values; a procedure does not.

When you create a user-defined routine with the CREATE FUNCTION or CREATE PROCEDURE statement, you own, and automatically receive the Execute privilege on, that routine. The Execute privilege allows you to invoke the user-defined routine with an EXECUTE FUNCTION or EXECUTE PROCEDURE statement, or with a CALL statement in an SPL routine. The Execute privilege also permits use of a function in an expression, as in the following example:

```
SELECT * FROM table WHERE in_stock(partnum) < 20
```



Element	Purpose	Restrictions	Syntax
routine name	The name given to the user-defined routine in a CREATE FUNCTION or CREATE PROCEDURE statement	The identifier must refer to an existing user-defined routine. In an ANSI-compliant database, specify the owner as the prefix to the routine name.	Function Name, p. 1-959 or Procedure Name, p. 1-1004

Privilege	Functions
SPECIFIC	Grants the Execute privilege for the routine identified by *specific name*.
FUNCTION	Grants the Execute privilege for any user-defined function with the specified *routine name* (and parameter types that match *routine parameter list,* if supplied).
PROCEDURE	Grants the Execute privilege for any user-defined procedure with the specified *routine name (*and parameter types that match *routine parameter list,* if supplied).
ROUTINE	Grants Execution privilege for user-defined functions and user-defined procedures with the specified *routine name (*and parameter types that match *routine parameter list,* if supplied).

If both a function and a 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 keyword FUNCTION, PROCEDURE, or SPECIFIC.

To limit the Execute privilege to a user-defined routine that accepts particular data types as arguments, include the data types as the routine parameter list in the GRANT statement or use the SPECIFIC keyword if a specific name exists for that routine and parameter list.

The requirement to grant the Execute privilege explicitly depends on the following conditions:

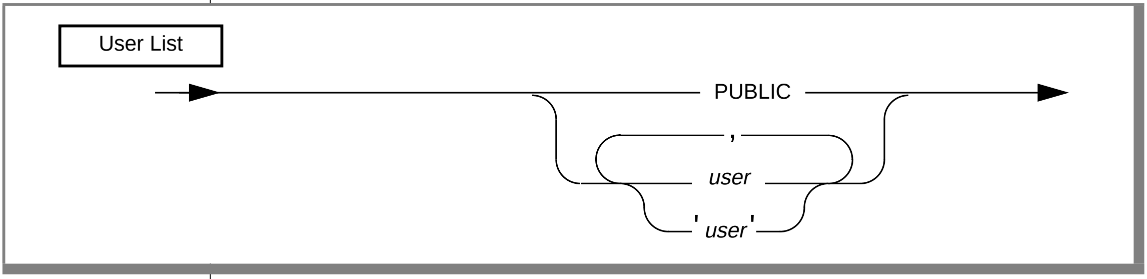
- If you have DBA-level privileges, you can use the DBA keyword of CREATE FUNCTION or CREATE PROCEDURE to restrict the default Execute privilege to users with the DBA database-level privilege. You must explicitly grant the Execute privilege on that routine to users who do not have the DBA privilege.
- If you have the Resource database-level privilege, but not the DBA privilege, you cannot use the DBA keyword when you create a routine:
  - When you create a routine in a database that is *not* ANSI compliant, **public** can execute that routine. 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 routine until you explicitly grant the Execute privilege.

ANSI

- In an ANSI-compliant database, the creator of a routine must explicitly grant the Execute privilege on that routine. ♦

User List

You can grant privileges to an individual user or a list of users. You can also use the PUBLIC keyword to grant privileges to all users.



Element	Purpose	Restrictions	Syntax
user	The login name to receive the role or privilege granted	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 grant a role or privilege to all authorized users.	Identifier, p. 1-962

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
```

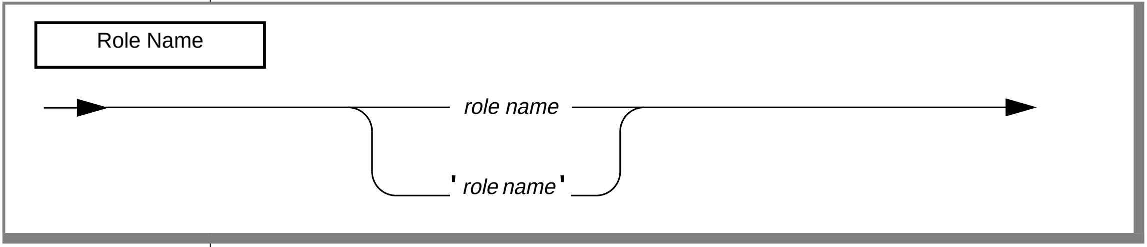
ANSI

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	The name of the role that is granted, or the name of the role to which a privilege or another role is granted	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-962

Granting a Role to a User or Another Role

The CREATE ROLE statement (page 1-190) must add a role name to the database before anyone can use that role name in a GRANT statement.

A DBA has the authority to grant a new role to another user. If a user receives a role WITH GRANT OPTION (page 1-474), 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 the SET ROLE statement enables it. A role grantor or a role grantee can issue the SET ROLE.*

The following example shows the sequence 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-, type-, 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.

When you grant a privilege to a role:

- you can specify the AS *grantor* clause (page 1-475). In this way, the people who have the role can revoke these same privileges.
- 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 Clause

The WITH GRANT OPTION clause creates a chain of grantors. When you include this clause to a GRANT statement, you grant privileges to *user* and authorize *user* to grant the same privileges to others.

If you use the WITH GRANT OPTION to grant privileges, you forfeit control over the future dissemination of those privileges.

If you revoke a privilege you granted with the WITH GRANT OPTION, you revoke the privilege from all users who received it as a result of the WITH GRANT OPTION chain that you initiated. (See "Revoking Privileges Granted WITH GRANT OPTION" on page 1-584 for examples.)

If you want to create a chain of privileges with another user as the source of the privilege, use the AS *grantor* clause. In that case, *grantor* can revoke all privileges along the WITH GRANT OPTION chain.

AS grantor Clause

If you issue a GRANT command with the AS *grantor* clause, you relinquish the right to revoke the privileges that you grant to the named *grantor*. The login given in the AS *grantor* clause replaces your login in the **systabauth** system catalog table.

***Important:** You cannot reverse the AS grantor clause. Once you commit a GRANT naming another as grantor, that person retains the sole right to revoke that GRANT.*

The remaining code fragments in this section illustrate the effects of AS *grantor*.

As owner of the **items** table, you grant all privileges to the user **tom**.

```
REVOKE ALL ON items FROM PUBLIC;
GRANT ALL ON items TO tom
```

The system catalog **systabauth** shows your login as grantor; you retain the right to revoke all privileges on **items** from the user **tom**.

You also grant Select and Update privileges to the user **jim**, but you specify **tom** as grantor.

```
GRANT SELECT, UPDATE ON items TO jim AS tom
```

The system catalog **systabauth** shows **tom** as grantor; only **tom** can revoke Select and Update privileges on **items** from the user **jim**. Later, you decide to revoke privileges on the **items** table from the user **tom**, so you issue the following statement:

```
REVOKE ALL ON items FROM tom
```

When you try to revoke the privileges on **items** from the user **jim**, the database server returns an error, as the following example shows:

```
REVOKE SELECT, UPDATE ON items FROM jim

580: Cannot revoke permission.
```

The database server issues the error because it has **tom** recorded as the original grantor, which you cannot change. Even a table owner cannot revoke a privilege that another user granted.

References

See the GRANT FRAGMENT, REVOKE, and REVOKE FRAGMENT statements in this manual.

For more information about routines and parameter lists, see the CREATE FUNCTION and CREATE PROCEDURE in this manual.

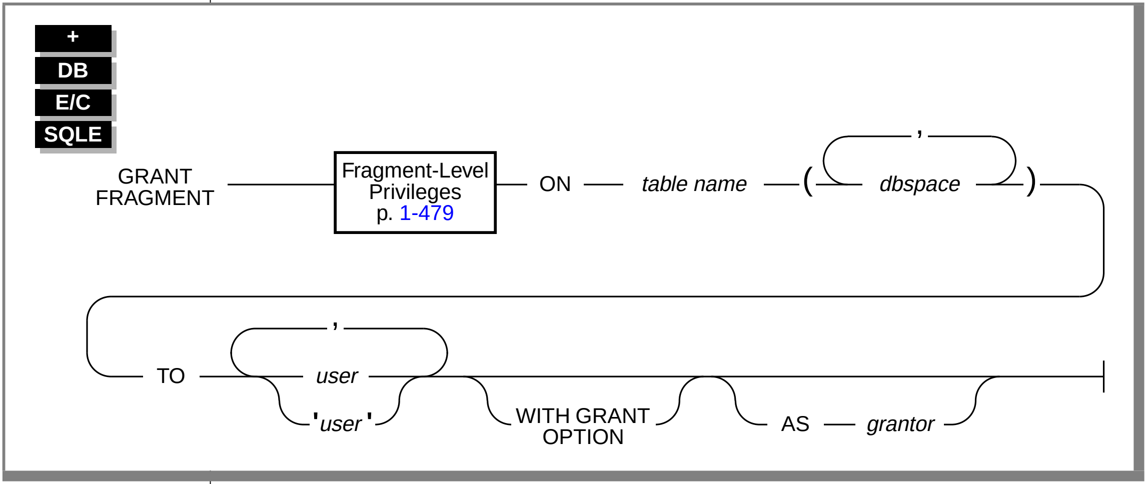
For information on roles, see the CREATE ROLE, DROP ROLE, and SET ROLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of database-level privileges and table-level privileges in Chapter 4 and the discussion of privileges and security in Chapter 11.

GRANT FRAGMENT

The GRANT FRAGMENT statement enables you to grant Insert, Update, and Delete privileges on individual fragments of a fragmented table.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where the fragment is stored. Use this parameter to specify the fragment or fragments on which privileges are to be granted. There is no default value.	You must specify at least one dbspace. The specified dbspaces must exist.	Identifier, p. 1-962
grantor	The name of the user who is to be listed as the grantor of the specified privileges in the **grantor** column of the **sysfragauth** system catalog table. The user who issues the GRANT FRAGMENT statement is the default grantor of the privileges.	The user specified in *grantor* must be a valid user.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
table name	The name of the table that contains the fragment or fragments on which privileges are to be granted. There is no default value.	The specified table must exist and must be fragmented by expression.	Table Name, p. 1-1044
user	The name of the user or users to whom the specified privileges are to be granted. There is no default value.	If you put quotes around *user*, the name of the user is stored exactly as you typed it. In an ANSI-compliant database, the name of the user is stored as uppercase letters if you do not use quotes around *user*.	Identifier, p. 1-962

(2 of 2)

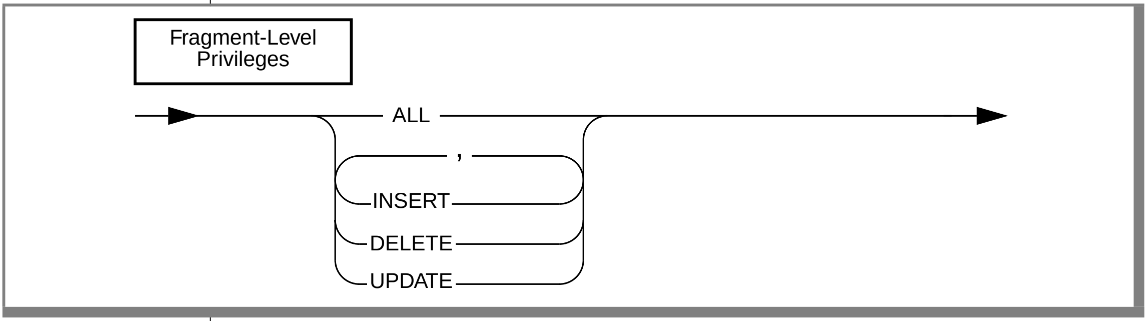
Usage

The GRANT FRAGMENT statement is similar to the GRANT statement. Both statements grant privileges to users. The difference between the two statements is that you use GRANT to grant privileges on a table while you use GRANT FRAGMENT to grant privileges on table fragments.

Use the GRANT FRAGMENT statement to grant the Insert, Update, or Delete privilege on one or more fragments of a table to one or more users.

The GRANT FRAGMENT statement is valid only for tables that are fragmented according to an expression-based distribution scheme. For an explanation of expression-based distribution schemes, see the ALTER FRAGMENT statement on page 1-27.

Fragment-Level Privileges



The following table defines each of the fragment-level privileges.

Privilege	Functions
ALL	Grants Insert, Update, and Delete privileges on a table fragment.
INSERT	Grants Insert privilege on a table fragment. This privilege gives the user the ability to insert rows in the fragment.
DELETE	Grants Delete privilege on a table fragment. This privilege gives the user the ability to delete rows in the fragment.
UPDATE	Grants Update privilege on a table fragment. This privilege gives the user the ability to update rows in the fragment and to name any column of the table in an UPDATE statement.

Definition of Fragment-Level Authority

When a fragmented table is created in an ANSI-compliant database, the table owner implicitly receives all table-level privileges on the new table, but no other users receive privileges.

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 new 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.

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.

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 granted by the GRANT FRAGMENT statement 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 authority lets users 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 INFORMIX-Universal 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 authority necessary for the operation attempted. If the table authority exists, the command continues processing.
2. If the table authority does 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 authority necessary for the operation attempted. If the fragment authority exists, the command continues processing. If the fragment authority does 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 Authority

The duration of fragment-level authority 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 authorities that exist for the affected fragments. Similarly, if you drop a dbspace, you also drop any authorities 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 authorities that the former fragment or fragments had when they were part of the fragmented table. Instead, such tables assume the default table authorities.

If a table with fragment authorities defined on it is changed to a table with a round-robin strategy or some other expression strategy, the fragment authorities are also dropped, and the table assumes the default table authorities.

Granting Privileges on One Fragment or a List of Fragments

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 the 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 the user **millie**:

```
GRANT FRAGMENT ALL ON customer (dbsp1, dbsp2) TO millie
```

Granting Privileges on All Fragments of a Table

If you want 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. However, 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 the 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 the 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 the 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 the 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 the 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 the 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 the 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.

Including the AS grantor Clause

When you include the AS *grantor* clause in the GRANT FRAGMENT statement, you specify that the user who is named in the *grantor* parameter is listed as the grantor of the privilege in the **grantor** column of the **sysfragauth** system catalog table.

In the following example, the DBA grants the Delete privilege on the fragment of the **customer** table in **dbsp3** to the user **martha**. In the GRANT FRAGMENT statement, the DBA uses the AS *grantor* clause to specify that the 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 a GRANT FRAGMENT statement does not include the AS *grantor* clause, the user who issues the statement is the default grantor of the privileges that are specified in the statement.

In the following example, the user grants the Update privilege on the fragment of the **customer** table in **dbsp3** to the 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
```

Consequences of the AS grantor Clause

If you omit the AS *grantor* clause, or if you specify your own user name in the *grantor* parameter, you can later revoke the privilege that you granted to the specified user. 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**.

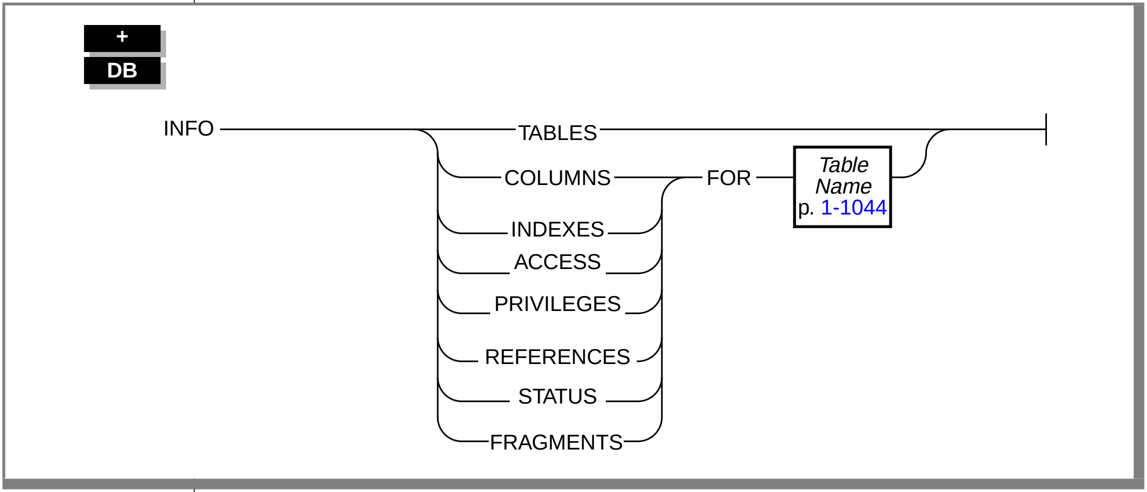
References

See the GRANT and REVOKE FRAGMENT statements in this manual.

INFO

Use the INFO statement to display a variety of information about databases and tables.

Syntax



Usage

You can use keywords in the INFO statement to display the following information.

Information Displayed	INFO Keyword
List of tables in the current database	TABLES
Column information for a specified table	COLUMNS
Index information for a specified table	INDEXES
Fragment strategy for a table	FRAGMENTS
User access privileges for a specified table	ACCESS or PRIVILEGE

(1 of 2)

Information Displayed	INFO Keyword
Reference privileges for the columns of a specified table	REFERENCES
Status information for a specified table	STATUS

(2 of 2)

Instead of using the INFO statement, you can use the Info options on the SQL menu or the TABLE menu to display the same and additional information.

TABLES Keyword

Use the TABLES keyword to display a list of the tables and views in the current database. The name of a table can appear in one of the following ways:

- 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 owner's name precedes the table name, such as **'june'.cust_calls**.

INFO statement requesting table information for the stores7 database

```
INFO TABLES
```

Display of table information

```
Table name

call_typecatalogcust_callscustomer
custviewitemslog_recordmanufact
orderssomeordersstatestock
```

In this display, the TABLES keyword provides information for the user-defined tables and views of the **stores7** database. It does not display the system catalog tables and system catalog views.

COLUMNS Keyword

Use the COLUMNS keyword to display the names and data types of the columns in a specified table and whether null values are allowed. The following examples show an INFO statement and the resulting display of information about the columns in a table:

INFO statement requesting column information

```
INFO COLUMNS FOR cust_calls
```

Display of column information

```
Column name          Type                                   Nulls

customer_num         integer                                yes
call_dtime           datetime year to minute                yes
user_id              char(18)                               yes
call_code            char(1)                                yes
call_descr           char(240)                              yes
res_dtime            datetime year to minute                yes
res_descr            char(240)                              yes
```

The COLUMNS keyword provides information for built-in data types (such as INTEGER, CHAR, and DATETIME) as well as user-defined data types (such as collection types, row types, and opaque types).

INDEXES Keyword

Use the INDEXES keyword to display the following information for each index on a table: the index name, the index owner, the index type (unique or duplicate), whether the index is clustered, the index access method used (functional, B-tree, and so on), and the names of the columns that are indexed.

The following examples show an INFO statement and the resulting display of information about the indexes of a table.

INFO statement requesting index information

```
INFO INDEXES FOR cust_calls
```

Display of index information

```
Index name        Owner     Type/Clstr    Access Method    Columns

c_num_dt_ix       velma     unique/No     B-Tree           customer_num
                                                           call_dtime
c_num_cus_ix      velma     dupls/No      B-Tree           customer_num
```

FRAGMENTS Keyword

Use the FRAGMENTS keyword to display the dbspace names where fragments are located for a specified table. The following examples show an INFO statement and the resulting display of fragments for a table that is fragmented with a round-robin distribution scheme. An INFO statement that is executed on a table that is fragmented with an expression-based distribution scheme would show the expressions and the dbspaces.

INFO statement requesting fragment information

```
INFO FRAGMENTS FOR new_accts
```

Display of fragment information

```
dbsp1

dbsp2

dbsp3
```

Displaying Privileges, References, and Status

You can use keywords in your INFO statement to display information about the access privileges (including the References privilege) or the status of a table.

ACCESS or PRIVILEGES Keyword

Use the ACCESS or PRIVILEGES keywords to display user access privileges for a specified table. The following examples show an INFO statement and the resulting display of user privileges for a table:

INFO statement requesting privileges information

```
INFO PRIVILEGES FOR cust_calls
```

Display of privileges information

```
User     Select             Update             Insert  Delete  Index  Alter

public    All                All                Yes     Yes     Yes    No
```

REFERENCES Keyword

Use the REFERENCES keyword to display the References privilege for users for the columns of a specified table. The following examples show an INFO statement and the resulting display:

INFO statement requesting References privilege information

```
INFO REFERENCES FOR newtable
```

Display of References privilege information

```
User        Column References

betty       col1
            col2
            col3
wilma       All
public      None
```

The output indicates that the user **betty** can reference columns **col1**, **col2**, and **col3** of the specified table; the user **wilma** can reference all the columns in the table; and **public** cannot access any columns in the table.

If you want information about database-level privileges, you must use a SELECT statement to access the **sysusers** system catalog table.

See the GRANT and REVOKE statements for more information about database and table-access privileges.

STATUS Keyword

Use the STATUS keyword to display information about the owner, row length, number of rows and columns, and creation date for a specified table. The following example displays status information for the **cust_calls** table:

INFO statement requesting status information

```
INFO STATUS FOR cust_calls
```

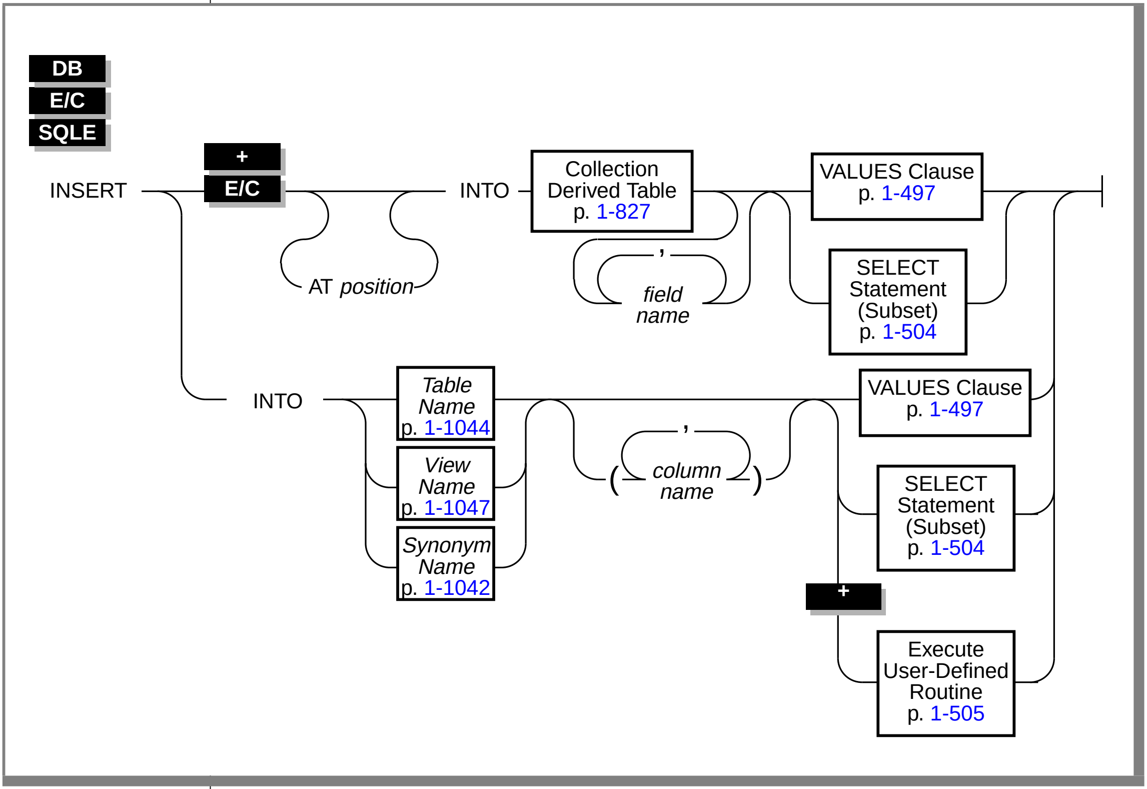
Display of status information

```
Table Name        cust_calls
Owner             velma
Row Size          517
Number of Rows    7
Number of Columns 7
Date Created      01/28/1993
```

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 INFORMIX-ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column that receives a new column value, or a list of columns that receive new values. If you specify a column list, values are inserted into columns in the order in which you list the columns. If you do not specify a column list, values are inserted into columns in the column order that was established when the table was created or last altered.	The number of columns you specify must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. 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.	Identifier, p. 1-962
position	The position at which you want to insert an element in a LIST	The *position* can be a literal number or a procedure variable of type INT or SMALLINT.	
field name	The name of a field of a named or unnamed row type	The row type must already be defined in the database.	"Extended Field Definition" on page 1-199 and "Unnamed Row Types" on page 1-870

Usage

Use the INSERT statement to create either of the following types of objects:

- A row in a table: either a single new row of column values or a group of new rows using data selected from other tables
- An element in a collection variable ♦

E/C

SPL

For information on how to insert an element into a collection variable, see "Inserting Into a Collection Variable" on page 1-506. The other sections of this INSERT statement describe how to create a row in a table.

To insert data into a table, you must either own the table or have the Insert privilege for the table (see the GRANT statement on page 1-458). 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 1-494.

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.

ESQL

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. (For more information about the DESCRIBE statement, see page 1-335.) ♦

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 columns, the columns receive data in the order in which you list them. 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.

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, refer to the WITH CHECK OPTION discussion under the CREATE VIEW statement on page 1-286.

If several users are entering sensitive information into a single table, the 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)
```

Inserting Rows with a Cursor

ESQL

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 following the last successfully inserted rows are discarded. ♦

Inserting Rows into a Database Without Transactions

If you are inserting rows into a database without transactions, you must take explicit action to restore inserted rows after a failure. For example, if the INSERT statement fails after you insert some rows, the successfully inserted rows remain in the table. You cannot recover automatically from a failed insert.

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.

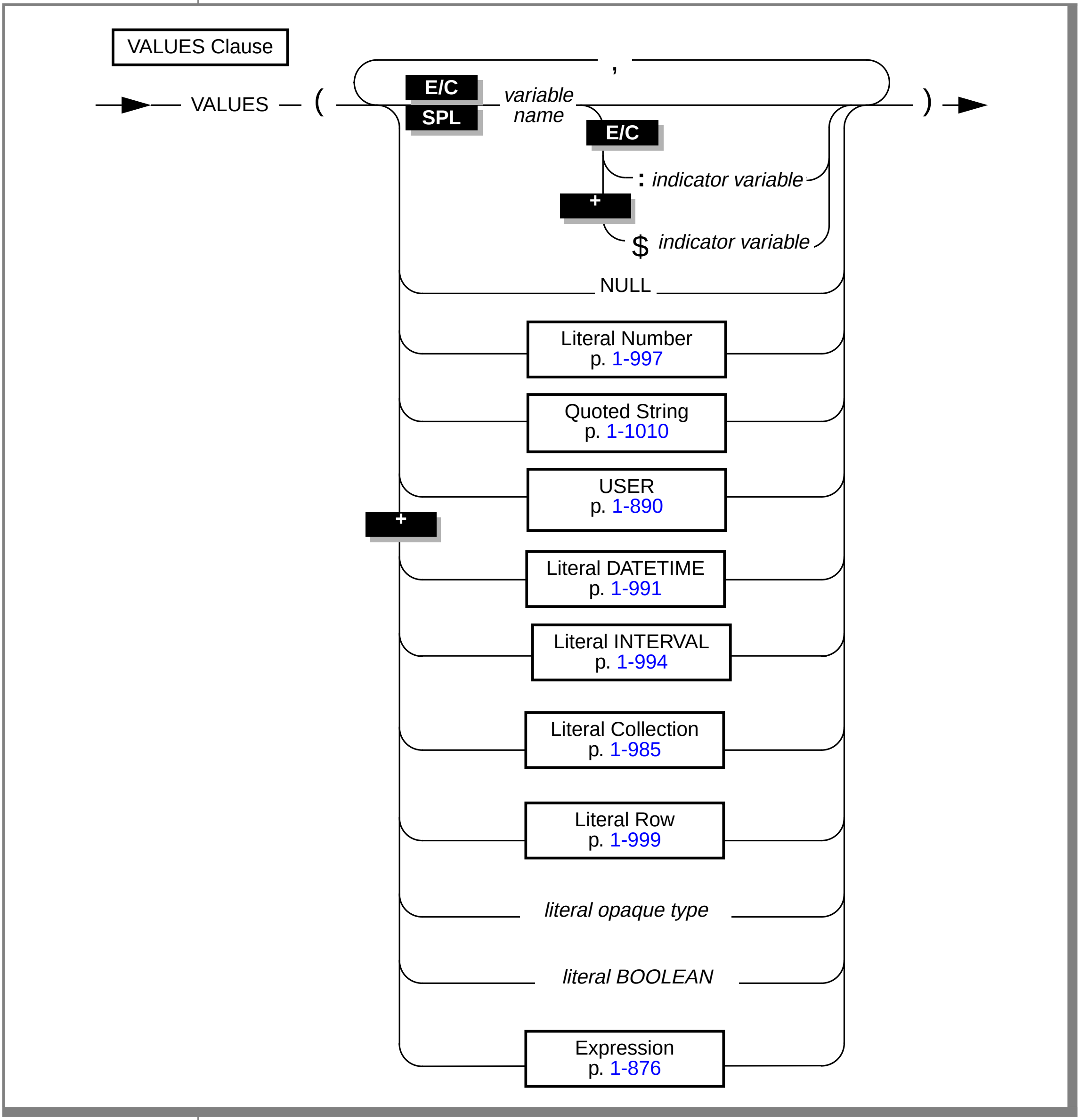
ANSI

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.

When you use INFORMIX-Universal Server within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

Rows that you insert within 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



Element	Purpose	Restrictions	Syntax
indicator variable	A program variable associated with *variable name* that indicates when an SQL API statement returns a null value to *variable name*	See your SQL API manual for the restrictions that apply to indicator variables in a particular language.	The name of the indicator variable must conform to language-specific rules for naming indicator variables.
literal opaque type	The literal representation for an opaque data type	Must be a literal that is recognized by the input support function for the associated opaque type.	The literal representation is defined by the developer of the opaque type.
literal BOOLEAN	The literal representation of a BOOLEAN value	A literal BOOLEAN value can only be `'t'` (TRUE) or `'f'` (FALSE) and must be specified as a quoted string.	Quoted String, p. 1-1010
variable name	A host variable that specifies a value to be inserted into a column	You can specify in *variable name* any other value option listed in the VALUES clause (NULL, Literal Number, and so on). If you specify a quoted string in *variable name*, the string can be longer than the 32-kilobyte maximum that applies to your specified quoted strings.	The name of the host variable must conform to language-specific rules for variable names.

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 of the string is 256 bytes. If you insert a value greater than `256`, the database server returns an error.

ESQL

If you are using variables, you can insert quoted strings longer than 256 bytes into a table. ♦

Data Type Compatibility and Casting

The value that you insert into a column does not have to be of the same data type as the column that receives it. However, these two data types 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. For a summary of the casting that the database server provides, see Chapter 2 of the *Informix Guide to SQL: Reference*. For information on how to create a user-defined cast, see the CREATE CAST statement in this manual and the *Extending INFORMIX-Universal Server: Data Types* manual.

Inserting Values into Character Columns

E/C

ANSI

With INFORMIX-ESQL/C, if you use a host variable to insert a value in a character column (CHAR, VARCHAR, or LVARCHAR) of a database that is ANSI compliant, the string within the host variable must be null terminated. The database server generates an error if you try to insert a string that is not null terminated. For more information, refer to the chapter on character data types in the *INFORMIX-ESQL/C Programmer's Manual*.

Inserting Values into TEXT and BYTE Columns

You can use the INSERT statement on tables with TEXT or BYTE columns if you:

- insert a NULL value into a TEXT or BYTE column.

  For example, the following INSERT statement inserts a new row into the **catalog** table:

  ```
  INSERT INTO catalog
  VALUES (0, 1, "HRO", NULL, NULL,
     "description of new catalog item")
  ```

- list all columns *except* the TEXT or BYTE columns before the VALUES clause.

  For example, the following INSERT statement inserts the same new row into the **catalog** table:

  ```
  INSERT INTO catalog (catalog_num, stock_num,
     manu_code, cat_advert)
  VALUES (0, 1, "HRO", "description of new catalog item")
  ```

You cannot use literal values within an INSERT statement to put simple large-object data within a TEXT or BYTE column. To insert values into a TEXT or BYTE column, you can use any of the following methods:

- Use the LOAD statement from within DB-Access to load the simple large-object data from an operating system file.

  For more information, see the description of LOAD in this chapter.
- Use **loc_t** host variables within an INFORMIX-ESQL/C client application.

  For more information, see the chapter on simple large objects in the *INFORMIX-ESQL/C Programmer's Manual*.

Inserting Values into SERIAL and SERIAL8 Columns

If you want to insert consecutive serial values in a SERIAL or SERIAL8 column in the table, specify a zero for a SERIAL or SERIAL8 column in the INSERT statement. When a SERIAL or SERIAL8 column is set to zero, the database server assigns the next highest value. If you want to enter an explicit value in a SERIAL or SERIAL8 column, specify the nonzero value after you first verify that the value does not duplicate one already in the table. If the SERIAL or SERIAL8 column is uniquely indexed or has a unique constraint, and you try to insert a value that duplicates one already in the table, an error occurs. For more information about the SERIAL and SERIAL8 data types, see Chapter 2 of the *Informix Guide to SQL: Reference*.

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 very large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign()**. When you execute the INSERT statement 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 the *Extending INFORMIX-Universal Server: Data Types* manual.

Inserting Values into Collection Columns

You can use the VALUES clause to insert literal values into a collection column, which can be a LIST, MULTISET, or SET. 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 adds literal values as a row in the **tab1** table:

```
EXEC SQL insert into tab1 values
    (
        5,
        "LIST{ROW(1, 'abcde'),
            ROW(2, 'fghij'),
            ROW(3, 'klmno')}",
        3.5
    )
```

The collection column, **list1**, in the **tab1** row has three elements, each element is an unnamed row type with an INTEGER field and a CHAR(5) field. For more information on the syntax for literal collection values, see "Literal DATETIME" on page 1-991.

E/C

You can use ESQL/C host variables to insert:

- an entire collection into a column.

  Use a **collection** variable as a *variable name* in the VALUES clause to insert an entire collection. For example, the following ESQL/C code fragment inserts the elements of the **a_set** host variable into the **set_col** column of the **tab_a** table:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
     client collection set(smallint not null) a_set;
  EXEC SQL END DECLARE SECTION;
  ...
  EXEC SQL insert into tab_a (set_col) values (:a_set);
  ```

- individual elements in a collection.

  To insert non-literal values into a collection column, you must first insert the elements in a **collection** variable and then specify the **collection** variable in the SET clause of an UPDATE statement. For information on how to insert values into a **collection** variable, see "Inserting Into a Collection Variable" on page 1-506. ♦

Important: *A collection column cannot contain NULL elements.*

Inserting Values into Row-Type Columns

You can use the VALUES clause to insert literal and nonliteral values in a named row type or unnamed row type column. For example, suppose you define the following named row type and table:

```
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 following INSERT statement inserts *literal* values in the **name** and **address** columns of the **employee** table:

```
INSERT INTO employee VALUES
    (
        ROW('John', 'Williams'),
        ROW('103 Baker St', 'Tracy','CA', 94060)::address_t
    )
```

The INSERT statement uses ROW constructors to generate values for the **name** column (an unnamed row type) and the **address** column (a named row type). When you specify a value for a named row type, you must use the CAST AS keyword or the double colon (::) operator, in conjunction with the name of the named row type, to cast the value to the named row type.

For more information on the syntax for ROW constructors, see "Constructor Expressions" on page 1-895 in the Expression segment. For information on literal values for named row types and unnamed row types, see the Literal Row segment on page 1-999.

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 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, you can first insert 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 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 1-510. ♦

Using Expressions in the VALUES Clause

You can insert any type of expression into a column. For example, you can insert a cast expression or a function that returns the current date, date and time, login name of the current user, or database server name of the current Universal Server instance.

The TODAY keyword returns the system date. The CURRENT keyword returns the system date and time. The USER keyword returns an eight-character 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. The following example uses the CURRENT and USER keywords to insert a new row into the **cust_calls** table:

```
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 the Expression segment on page 1-876.

Inserting Nulls with the VALUES Clause

When you execute an INSERT statement, a null value is inserted into any column for which you do not provide a value as well as for all columns that do not have default values associated with them, which are not listed explicitly. You also can use the NULL keyword 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.

Subset of SELECT Statement

You can insert the rows of data that result from a SELECT statement into a table if the insert data is selected from another table or tables.

If this statement has a WHERE clause that does not return rows, **sqlca** returns SQLNOTFOUND (100) for ANSI-compliant databases. In databases that are not ANSI compliant, **sqlca** returns (0). When you insert as a part of a multi-statement prepare, and no rows are inserted, **sqlca** returns SQLNOTFOUND (100) for both ANSI databases and databases that are not ANSI compliant. The following SELECT clauses are not supported:

- INTO TEMP
- ORDER BY
- UNION

In addition, the FROM clause of the SELECT statement cannot contain the same table name as the table into which you are inserting rows, as shown in the following example:

```
INSERT INTO newtable
    SELECT item_num, order_num, quantity, stock_num,
           manu_code,  total_price
        FROM items
```

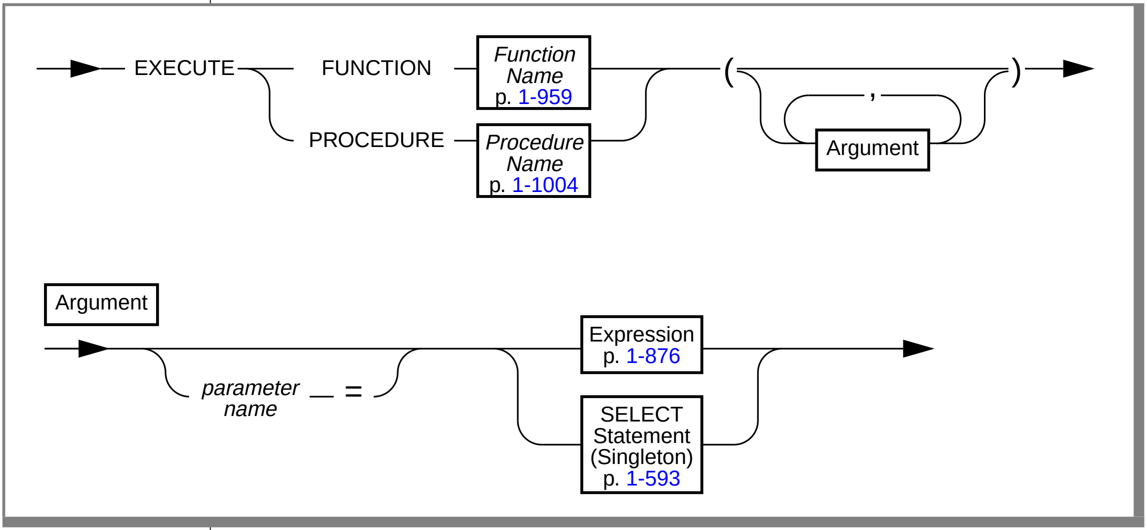
For detailed information on SELECT statement syntax, see page 1-593.

Using INSERT as a Dynamic Management Statement

E/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 *INFORMIX-ESQL/C Programmer's Manual* manual. ♦

Inserting Data with a User-Defined Routine



Element	Purpose	Restrictions	Syntax
parameter name	The name of an input parameter to the procedure	The input parameter must have been defined in the CREATE FUNCTION or CREATE PROCEDURE statement for the specified user-defined routine.	Expression, p. 1-876

You can execute the following types of routines to generate values to be inserted into a column:

- A user-defined function
- A legacy stored procedure

Inserting Data With a User-Defined Function

You can specify the EXECUTE FUNCTION statement instead of a VALUES clause in the INSERT statement to insert into a table values that a user-defined function returns. The values that the user-defined function returns must match those expected by the column list in number and data type.

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. ♦

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. ♦

SPL

Inserting Data With a Legacy Stored Procedure

Universal Server supports use of the EXECUTE PROCEDURE statement in an INSERT statement to insert the rows of data that result from a call to a legacy stored procedure. The values that the stored procedure returns must match those expected by the column list in number and data type. The number and data types of the columns must match those that the column list expects. Informix recommends that you use the EXECUTE FUNCTION statement to insert data from all new user-defined functions. ♦

E/C

SPL

Inserting Into a Collection Variable

The INSERT statement with the Collection Derived Table segment allows you to insert elements into a collection variable. The Collection Derived Table segment identifies the collection variable in which to insert the elements. For more information on syntax of the Collection Derived Table segment, see page 1-827.

E/C

In an INFORMIX-ESQL/C program, declare a host variable of type **collection** for a collection variable. This **collection** variable can be typed or untyped. ♦

SPL

In an SPL routine, declare a variable of type COLLECTION, LIST, MULTISET, or SET for a collection variable. This collection variable can be typed or untyped. ♦

To insert new elements, follow these steps:

1. Create a collection variable in your SPL routine or ESQL/C program.
2. Add collection element(s) to the collection variable with the INSERT statement and the Collection Derived Table segment.
3. Once the collection variable contains the correct elements, you then use the INSERT or UPDATE statement on a table or view name to save the collection variable in a collection column (SET, MULTISET, or LIST).

The INSERT statement and the Collection Derived Table segment allow you to perform the following operations on a collection variable:

- Insert *one* element into the collection

  Use the INSERT statement with the Collection Derived Table segment.

E/C

- Insert *one or more* elements into the collection

  Associate the INSERT statement and the Collection Derived Table segment with a cursor to declare a collection cursor for the **collection** variable.

  For information on how to use a collection cursor to add one or more elements to an ESQL/C **collection** variable, see "Inserting into a Collection Cursor" on page 1-560 in the PUT statement. ♦

The INSERT statement and the Collection Derived Table segment allow you to insert *one* element in a collection. For SET and MULTISET collections, the position of the new element is undefined because the elements of these collections are not ordered. However, LIST collections have elements that are ordered. If the column is the LIST type, you can use the AT clause to specify the position within the list at which you wish to add the new element. For more information, see "AT Clause" on page 1-509.

E/C

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);
```

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. For information on how to use **collection** host variables in an ESQL/C program, see the *INFORMIX-ESQL/C Programmer's Manual*. ♦

SPL

You can perform a similar insert with an SPL routine, as in the following example:

```
CREATE PROCEDURE test2()

    DEFINE a_multiset MULTISET(INT NOT NULL);

    SELECT multiset_col INTO a_multiset FROM table1
        WHERE id = 107;
    INSERT INTO TABLE(a_multiset) VALUES( 1423231 );
.
.
.
END PROCEDURE;
```

You can insert into the collection variable **a_multiset** without using a cursor, because the collection variable contains a MULTISET. The elements of a MULTISET are not listed in a particular order, and the position of the new element that is inserted is undefined. For more information on how to use SPL collection variables, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.♦

After you insert a new value into a collection variable, you need to store the new collection in the database. For more information, see "Saving the Collection Variable" on page 1-510. You can also use a collection variable as a *variable name* in the VALUES clause to insert elements into a collection. For more information, see "Inserting Values into Collection Columns" on page 1-501. ♦

AT Clause

By default, Universal Server adds a new element at the end of a LIST collection. The AT clause does provide the ability to insert LIST elements at a specified position. If you specify a position that is greater than the number of elements in the list, Universal Server adds the element to the end of the list. You must specify a position value of at least one because the first element in the list is at position 1.

E/C

Suppose the ESQL/C host variable **a_list** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection list(smallint NOT NULL) a_list;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement adds a new list element of 9 as the new third element of **a_list**:

```
EXEC SQL insert at 3 into table(:a_list) values (9);
```

Suppose that before this INSERT, **a_list** contained the elements {1,8,4,5,2}. After this INSERT, this variable contains the elements {1,8,9,4,5,2}. For more information on how to insert values into ESQL/C **collection** variables , see the chapter on complex data types in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

SPL

You can perform a similar insert with an SPL routine, as the following example shows:

```
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 );
.
.
.
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 has been inserted at position 3 in the list. For more information on how to insert values into SPL collection variables, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

After you insert a new value into a collection variable, you need to store the new collection in the database. For more information, see “Saving the Collection Variable”.

Saving the Collection Variable

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 collection column with one of the following SQL statements:

- To update the collection column in the table with the **collection** variable, use an UPDATE statement on a table or view name and specify the collection variable in the SET clause.

  For more information, see “Updating Collection Columns” on page 1-786 in the UPDATE statement.

- To insert a collection in a column, use the INSERT statement on a table or view name and specify the collection variable in the VALUES clause.

  For more information, see “Inserting Values into Collection Columns” on page 1-501.

E/C

SPL

Inserting into a Row Variable

The INSERT statement does not support a row variable in the Collection Derived Table segment. However, you can use the UPDATE statement 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 1-502 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 1-798. ♦

References

See the SELECT statement in this manual. See also the CLOSE, DECLARE, DESCRIBE, EXECUTE, FLUSH, OPEN, PREPARE, and PUT statements in Chapter 1 of this manual for specific information about dynamic management statements. See also the FOREACH statement in Chapter 2 of this manual.

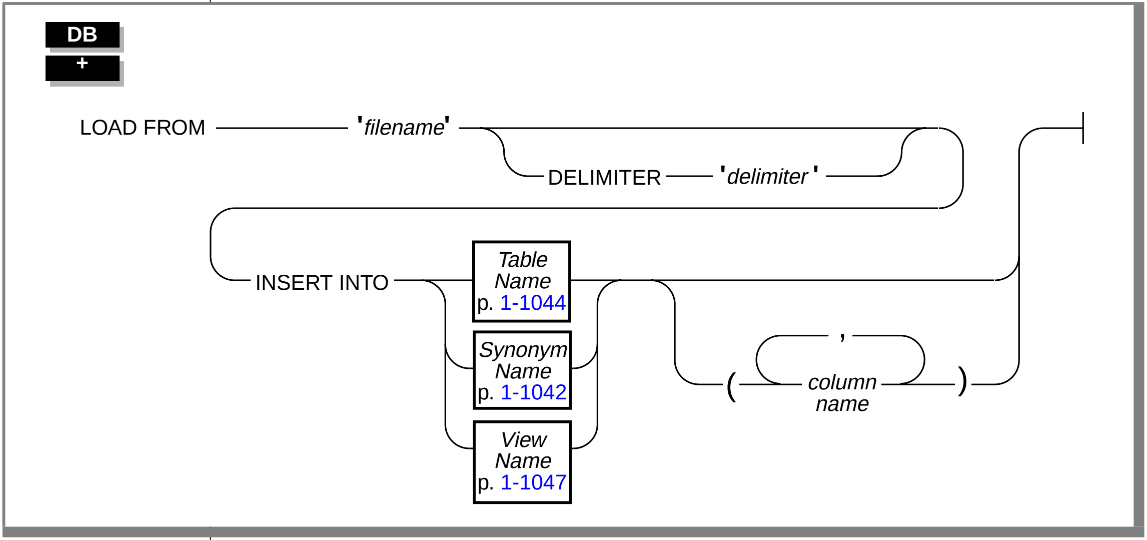
In the *Informix Guide to SQL: Tutorial*, see the discussion of inserting data in Chapter 4 and Chapter 6. In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the INSERT statement.

For information on how to access row and collections with ESQL/C host variables, see the chapter on complex types in the *INFORMIX-ESQL/C Programmer's Manual*. For information on how to access row and collections with SPL variables, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.

LOAD

Use the LOAD statement to insert data from an operating-system file into an existing table, synonym, or view.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column or columns that receive data values from the load file during the load operation	You must specify the columns that receive data if you are not loading data into all columns. You must also specify columns if the order of the fields in the load file does not match the default order of the columns in the table (the order established when the table was created).	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
delimiter	A quoted string that identifies the character to use as the delimiter in the LOAD FROM file. The delimiter is a character that separates the data values in each line of the LOAD FROM file.	If you do not specify a delimiter character, the database server uses the value of the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the vertical bar (|). You cannot use the following items as delimiter characters: backslash (\\), new-line character (=CTRL-J), and hexadecimal numbers (0 to 9, a to f, A to F).	Quoted String, p. 1-1010
filename	A quoted string that identifies the pathname and filename of the load file. The load file contains the data to be loaded into the specified table or view. The default pathname for the load file is the current directory.	If you do not include a list of columns in the *column name* parameter, the fields in the load file must match the columns specified for the table in number, order, and type. You must also observe restrictions about the same number of fields in each line, the relationship of field lengths to column lengths, the representation of data types in the file, the use of the backslash character (\\) with certain special characters, and special rules for VARCHAR and BLOB data types. See "The LOAD FROM File" on page 1-514 for information on these restrictions.	Quoted String, p. 1-1010. The pathname and filename specified in the quoted string must conform to the conventions of your operating system.

(2 of 2)

Usage

The LOAD statement adds 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 on page 1-458.

The LOAD FROM File

The LOAD FROM file contains the data to add to a table. You can use the file that the UNLOAD statement creates as the LOAD FROM 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 your Informix product expects you to represent the data types in the LOAD file (when they use the default locale, U.S. English).

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	A collection must have its values surrounded by braces ({}) and a field delimiter separating each element. For more information, see "Loading Complex Types" on page 1-519.
date	A 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. See the *Guide to GLS Functionality* for more information about these environment variables.

(1 of 3)

Type of Data	Input Format
MONEY	A value that can include currency notation: a leading currency symbol ($), a comma (,) as the thousands separator, and a period (.) as the decimal separator. You can use a different currency notation if you indicate this notation with the **DBMONEY** environment variable. For more information on this environment variable, see the *Guide to GLS Functionality*.
NULL	Nothing between the delimiters.
row types (named and unnamed)	A row type must have its values surrounded by parentheses and a field delimiter separating each element. For more information, see "Loading Complex Types" on page 1-519.
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 1-518.
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 1-518.
time	A character string in the following format: *year-month-day hour:minute:second.fraction*. You cannot use type specification or qualifiers for DATETIME or INTERVAL values. The year must be a four-digit number, and the month must be a two-digit number. You can specify a different date and time format with the **GL_DATETIME** or **DBTIME** environment variable. See the *Guide to GLS Functionality* for more information on these environment variables.

(2 of 3)

Type of Data	Input Format
user-defined data formats (opaque types)	The 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 if the data is in binary format. The data in the LOAD FROM file must correspond to the format that the import or importbinary support function expects. The associated opaque type must have an assign support function if special processing is required before the data is written in the database.

(3 of 3)

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 *Guide to GLS Functionality*. ♦

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 (\):

- Backslash
- Delimiter
- New-line character anywhere in the value of a VARCHAR or NVARCHAR column
- New-line character at end of a value for a TEXT value

Do not use the backslash character (\) as a field delimiter. It serves as an escape character to inform the LOAD statement that the next character is to be interpreted as part of the data.

The following example shows the contents of a hypothetical 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 vertical bar (|), the default delimiter character
- 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 delimiter characters with nothing between them.

The following statement loads the values from the **new_custs** file into the **customer** table owned by **jason**:

```
LOAD FROM 'new_custs' INSERT INTO jason.customer
```

For more information about the format of the input file, see the discussion of the **dbload** utility in the *Informix Migration Guide*.

Loading Character Data

The 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 columns that are the VARCHAR data type, 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.

GLS

These restrictions on character columns also apply to NCHAR and NVARCHAR columns. For more information on these data types, see the *Guide to GLS Functionality*. ♦

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 embedded 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.

GLS

For TEXT columns, the database server handles any required code-set conversions for the data. For more information, see the *Guide to GLS Functionality*. ♦

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 *Informix Guide to SQL: Reference*.

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. It copies all smart-large-object values into a single file. Each BLOB or CLOB value is appended to the current file. The database server might create several files if the values are extremely large or there any many values.

In a LOAD FROM file, a CLOB or BLOB column value appears as follows:

```
start_off, end_off, client_path
```

In this format, *start_off* is the starting offset of the smart-large-object value within the file, *end_off* is the length of the BLOB or CLOB value, and *client_path* is the pathname for the client file. For example, to load a CLOB value that is 2048 bytes long and stored at the beginning of the **/usr/apps/clob_val** file, the database server expects the following value in the LOAD FROM file to appear as follows:

```
|0, 2048, /usr/apps/clob_value|
```

The preceding example assumes a default field delimiter of the vertical bar.

Loading Complex Types

In a LOAD FROM file, complex types appear as follows:

- Collections are introduced with the appropriate constructor SET, MULTISET, 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) have their fields enclosed with parentheses and separated with the field separator, as follows:

  ```
  (val1 | val2 | ... )
  ```

  For example, to load the ROW values `(1, 'abc')`, the corresponding field of the LOAD FROM file appears as:

  ```
  |(1 | abc)|
  ```

The preceding examples use the default field separator, the vertical bar (|).

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 very 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 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 the *Extending INFORMIX-Universal Server: Data Types* manual.

DELIMITER Clause

Use the DELIMITER clause to specify the delimiter that separates the data contained in each column in a row in the LOAD FROM file. If you omit this clause, your Informix product checks the **DBDELIMITER** environment variable.

If the **DBDELIMITER** environment variable has not been set, the default delimiter is the vertical bar (|). See Chapter 3 in the *Informix Guide to SQL: Reference* for information about how to set the **DBDELIMITER** environment variable.

You can specify TAB (CTRL-I) or <blank> (= ASCII 32) as the delimiter symbol. You cannot use the following items as the delimiter symbol:

- Backslash (\)
- New-line character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

The following statement identifies the semicolon (;) as the delimiter character:

```
LOAD FROM '/a/data/ord.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. (See the discussion of Synonym Name, Table Name, and View Name that begins on page 1-1042 for details.)

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 (determined when the table was created).

The following example identifies the **price** and **discount** columns as the only columns in which to add data:

```
LOAD FROM '/tmp/prices' DELIMITER ','
    INSERT INTO norman.worktab(price,discount)
```

References

See the UNLOAD and INSERT statements in this manual.

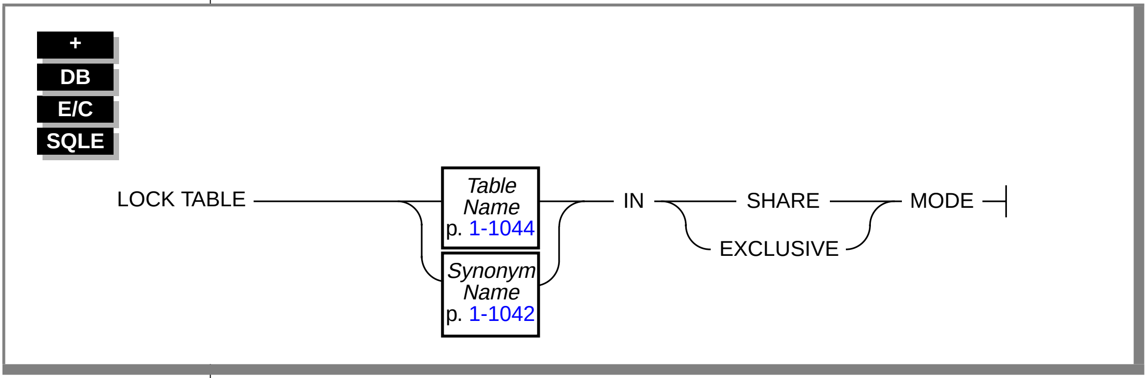
In the *Informix Migration Guide*, see the task-oriented discussion of the LOAD statement and other utilities for moving data.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the LOAD statement.

LOCK TABLE

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax



Usage

You can lock a table if you own the table or have the 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 an exclusive lock is attempted while another user has locked the table in share mode.

The SHARE keyword locks a table in shared mode. Shared mode allows 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. Exclusive mode denies other processes both *read* and *write* access to the table.

Exclusive-mode locking automatically occurs when you execute the ALTER INDEX, CREATE INDEX, DROP INDEX, RENAME COLUMN, RENAME TABLE, and ALTER 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, 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 Universal Server configuration.
- All row and table locks release automatically after a transaction is completed. The UNLOCK TABLE statement fails within a database that uses transactions.

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
```

Databases Without Transactions

In a database that was created without transactions, table locks set by using the LOCK TABLE statement are released after any of the following occurrences:

- 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
```

References

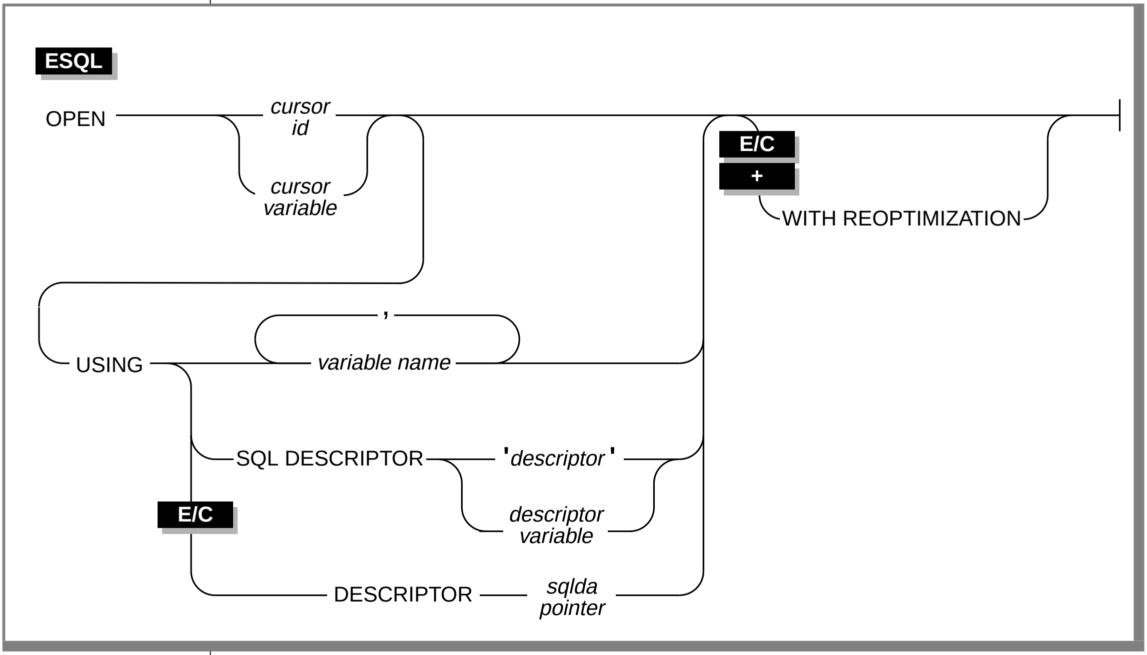
See the BEGIN WORK, SET ISOLATION, SET LOCK MODE, COMMIT WORK, ROLLBACK WORK, and UNLOCK TABLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of locks in Chapter 7.

OPEN

Use the OPEN statement to activate a cursor.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for a cursor	Cursor must have been previously created by a DECLARE statement.	Identifier, p. 1-962
cursor variable	Host variable that identifies a cursor	Host variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Variable name must conform to language-specific rules for variable names
descriptor	Quoted string that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-1010

(1 of 2)

Element	Purpose	Restrictions	Syntax
descriptor variable	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-1010
sqlda pointer	Pointer to an **sqlda** structure that defines the type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement	You cannot begin an *sqlda pointer* with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-335
variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character or collection data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The OPEN statement activates 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 statement
- An insert cursor: a cursor that is associated with the INSERT statement
- A collection cursor: a select or insert cursor that operates on a **collection** variable

You create a cursor with the DECLARE statement (see page 1-300). When the program opens the cursor with OPEN, the associated SELECT, INSERT, or EXECUTE FUNCTION statement is passed to the database server, which begins execution. The specific actions that the database server takes differ, depending on whether the cursor is associated with an INSERT statement (an insert cursor), or with a SELECT statement (a select cursor) or EXECUTE FUNCTION statement (a function cursor). When the program has retrieved or inserted all the rows it needs, close the cursor by using the CLOSE statement.

When you associate the SELECT, INSERT, or EXECUTE FUNCTION statement directly with a cursor (that is, you do not use PREPARE to prepare it before the DECLARE statement), associated with a cursor is by the OPEN statement implicitly prepares the statement. The total number of prepared objects and open cursors that are allowed in one program at any time is limited by the available memory. You can use the FREE statement to free the cursor and release the database server resources.

ANSI

You receive an error code if you try to open a cursor that is already open. ♦

Opening a Select Cursor

When you open a select cursor (a read-only or an update cursor), the SELECT statement is passed to the database server along with any values that the USING clause of the OPEN statement specifies. (If the statement was previously prepared, the statement passed to the database server when it was prepared.) 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 for a select cursor:

```
EXEC SQL declare s_curs cursor for
    select * from orders;
EXEC SQL open s_curs;
```

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 keyword. (See the DECLARE statement on page 1-300.)

Because the database server is seeing the query for the first time, it might detect errors in the query. In this case, the database server does not actually return the first row of data, but it sets a return code in the **SQLCODE** field of the **sqlca** structure (**sqlca.sqlcode**).

The **SQLCODE** value is either negative or zero, as the following table describes.

Return Code Value	Meaning
Negative	Shows an error has been detected in the SELECT statement.
Zero	Shows the SELECT statement is valid.

If the SELECT statement is valid, but no rows match its search 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** value might exist. Check the GET DIAGNOSTICS statement for information about how to get and interpret the **SQLSTATE** value.*

Opening a Function Cursor

When you open a function cursor, the EXECUTE FUNCTION statement is passed to the database server along with any values that the USING clause of the OPEN statement specifies. The values in the USING clause are passed as arguments to the user-defined function that the EXECUTE FUNCTION executes. 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 user-defined function to the point where it returns the first set of values.

The following example illustrates a simple OPEN statement for a function cursor in INFORMIX-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;
```

In the above example, the database server is seeing the EXECUTE FUNCTION statement for the first time when it executes the OPEN function. Therefore, it might detect syntactic errors in the statement. In this case, the database server does not actually return the first row of data, but it sets a return code in the **SQLCODE** field of the **sqlca** structure (**sqlca.sqlcode**).

The **SQLCODE** value is either negative or zero, as the following table describes.

Return Code Value	Meaning
Negative	Shows that an error has been detected in the EXECUTE FUNCTION statement.
Zero	Shows that the EXECUTE FUNCTION statement is valid.

If the EXECUTE FUNCTION statement is valid, but the user-defined function returns no rows, the first FETCH statement returns a value of 100 (SQLNOTFOUND), which means no values returned.

Tip: *When you encounter an* ***SQLCODE*** *error, a corresponding* ***SQLSTATE*** *value might exist. See the GET DIAGNOSTICS statement for information about the* ***SQLSTATE*** *status variable.*

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 the DECLARE statement on page 1-300.)

An OPEN statement for an insert cursor cannot include a USING clause. The following INFORMIX-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;
```

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. See the discussion of the PUT statement on page 1-552 for information about checking errors and counting inserted rows.

Opening a Collection Cursor

You can declare both select and insert cursors on **collection** variables. Such cursors are called collection cursors. You can use the OPEN statement to open these cursors. The OPEN statement allocates resources that the collection cursor needs. (For more information, see the DECLARE statement on page 1-300.)

You can use the name of a **collection** variable in the USING clause of the OPEN statement. For more information on the USING clause, see "USING Clause". For more information on the use of OPEN...USING with a **collection** variable, see "Fetching From a Collection Cursor" on page 1-419 and "Inserting into a Collection Cursor" on page 1-560.

USING Clause

The USING clause of the OPEN statement is required when the cursor is associated with a prepared statement that includes question-mark (?) placeholders, as follows:

- A SELECT statement that contains input parameters in its WHERE clause
- An EXECUTE FUNCTION statement that contains input parameters in as arguments of its user-defined function
- An INSERT statement that contains input parameters in its VALUES clause

(See the PREPARE statement on page 1-538.) You can supply values for these parameters in one of the following ways:

- You can specify host variables in the USING clause.
- You can specify a system-descriptor area in the USING SQL DESCRIPTOR clause.
- You can specify an **sqlda** structure in the USING DESCRIPTOR clause.

Naming Variables in USING

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 statement question-mark (?) parameters in a one-to-one correspondence, from left to right.

You must supply one host variable name for each placeholder. The data type of each variable must be compatible with the corresponding value that the prepared statement requires. The following example illustrates the USING clause of the OPEN statement with a SELECT statement in an INFORMIX-ESQL/C code fragment:

```
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_custnum;
```

The following example illustrates the USING clause of the OPEN statement with an EXECUTE FUNCTION statement in an INFORMIX-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;
```

You cannot include indicator variables in the list of variable names. To use an indicator variable, you must include the SELECT or EXECUTE FUNCTION statement as part of the DECLARE statement.

USING SQL DESCRIPTOR 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. A system-descriptor area describes the data type and memory location of one or more values.

X/O

You can also use an **sqlda** structure to dynamically supply parameters. However, a system-descriptor area conforms to the X/Open standards. ♦

To specify a system-descriptor area as the location of parameters, use the USING SQL DESCRIPTOR clause of the OPEN statement. This clause allows you to associate input values from a system-descriptor area when you open a cursor.

The following example shows the OPEN...USING SQL DESCRIPTOR statement:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL open selcurs using sql descriptor 'desc1';
```

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 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 further information, refer to the discussion of the system-descriptor area in the *INFORMIX-ESQL/C Programmer's Manual*.

USING DESCRIPTOR Clause

E/C

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. To specify an **sqlda** structure as the location of parameters, use the USING DESCRIPTOR clause of the OPEN statement. This clause allows you to associate input values from an **sqlda** structure when you open a cursor.

The following example shows the OPEN...USING DESCRIPTOR statement in INFORMIX-ESQL/C:

```
struct sqlda *sdp;
...
EXEC SQL open selcurs using descriptor sdp;
```

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.

For further information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

WITH REOPTIMIZATION Clause

The WITH REOPTIMIZATION clause allows you to reoptimize your query-design plan. When you prepare a SELECT statement or an EXECUTE FUNCTION statement, Universal Server uses a query-design plan to optimize that query. If you later modify the data that is associated with a prepared SELECT statement or the data that is associated with an EXECUTE FUNCTION statement, you can compromise the effectiveness of the query-design plan for that statement. In other words, if you change the data, you can deoptimize your query. To ensure optimization of your query, you can prepare the SELECT or EXECUTE FUNCTION statement again or open the cursor again using the WITH REOPTIMIZATION clause.

Informix recommends that you use the WITH REOPTIMIZATION clause because it provides the following advantages over preparing a statement again:

- Rebuilds only the query-design plan rather than the entire statement
- Uses fewer resources
- Reduces overhead
- Requires less time

The WITH REOPTIMIZATION clause also makes your database server optimize your query-design plan before processing the OPEN cursor statement. The following example shows the WITH REOPTIMIZATION clause in INFORMIX-ESQL/C:

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

Reopening a 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.

A subsequent OPEN statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new active set that is based on the current values of the variables in the USING clause. If the program 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. For these queries, the database server does not pre-fetch 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.

However, 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.

Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you release resources with a FREE *cursor id* or FREE *cursor variable* statement, you cannot use the cursor unless you declare the cursor again. If you execute a FREE *statement id* or FREE *statement id variable* statement, you cannot open the cursor that is associated with the statement id or statement id variable unless you prepare the statement id or statement id variable again.

References

See the CLOSE, DECLARE and FREE statements in this manual for general information about cursors. See the PUT and FLUSH statements in this manual for information about insert cursors. See the FETCH statement in this manual for information about select and function cursors.

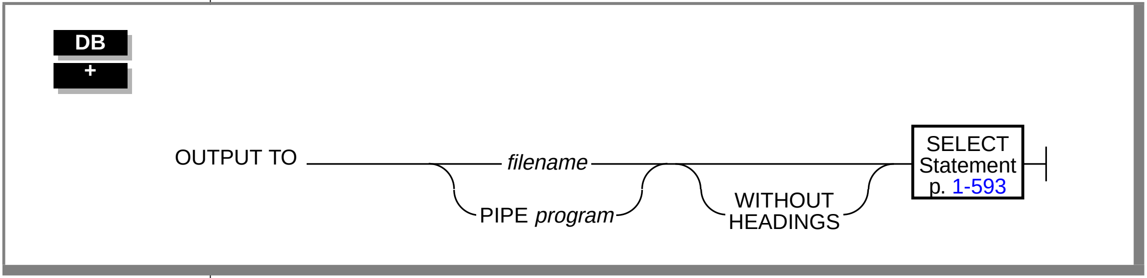
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for more information about dynamic SQL statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the OPEN statement in Chapter 5. Refer also to the *INFORMIX-ESQL/C Programmer's Manual* for more information about the system-descriptor area and the **sqlda** structure.

OUTPUT

Use the OUTPUT statement to send query results directly to an operating-system file or to pipe it to another program.

Syntax



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of an operating-system file where the results of the query are written. The default pathname is the current directory.	You can specify a new or existing file in *filename*. If the specified file exists, the results of the query overwrite the current contents of the file.	The pathname and filename must conform to the conventions of your operating system.
program	The name of a program where the results of the query are sent	The program must exist and must be known to the operating system. The program must be able to read the results of a query.	The name of the program must conform to the conventions of your operating system.

Usage

You can send the results of a query to an operating-system file by specifying the full pathname for the file. If the file already exists, the output overwrites the current contents, as the following example shows:

```
OUTPUT TO /usr/april/query1
    SELECT * FROM cust_calls WHERE call_code = 'L'
```

You can display the results of a query without column headings by using the WITHOUT HEADINGS keywords, as the following example shows:

```
OUTPUT TO /usr/april/query1
    WITHOUT HEADINGS
    SELECT * FROM cust_calls WHERE call_code = 'L'
```

You also 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
```

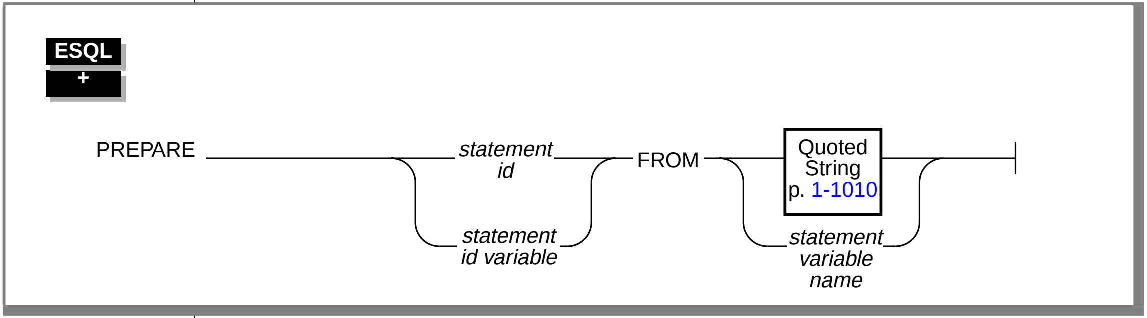
References

See the SELECT and UNLOAD statements in this manual.

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements in an INFORMIX-ESQL/C program at runtime.

Syntax



Element	Purpose	Restrictions	Syntax
statement id	A statement identifier that is a data structure representing the text of a prepared SQL statement	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.	Identifier, p. 1-962
statement id variable	Host variable that contains the statement identifier	This variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
statement variable name	Host variable whose value is a character string that consists of one or more SQL statements	This variable must be a character data type. For restrictions on the statements in the character string, see "SQL Statements Permitted in Single-Statement Prepares" on page 1-543 and "Restrictions for Multistatement Prepares" on page 1-550. A statement variable name cannot be used if the SQL statement contains the Collection Derived Table segment.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The PREPARE statement permits your program to assemble the text of an SQL statement at runtime 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.

The number of prepared objects in a single program is limited by the available memory. This limit includes both statement identifiers that are named in PREPARE statements (*statement id* or *statement id variable*) and cursor declarations that incorporate SELECT, EXECUTE FUNCTION, or INSERT statements. To avoid exceeding the limit, use a FREE statement to release some statements or cursors.

Using a Statement Identifier

A PREPARE statement sends the statement text to the database server where it is analyzed. 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 resources that were allocated to the statement. After you release the database server resources, you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.

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. See the manual for your SQL API for more information, restrictions, and performance issues when preprocessing with the **-local** option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or sequence of statements at a time. You can execute a new PREPARE statement with an existing statement identifier if you wish to bind a given statement 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 character-string variable. In the following ESQL/C examples, the first example shows a statement identifier that was prepared as an SQL API host variable; the second example shows a statement identifier that was prepared as a character-string constant:

```
stcopy ("query2", stmtid);
EXEC SQL prepare :stmtid from
    'select * from customer';

EXEC SQL prepare query2 from
    'select * from customer';
```

A statement ID variable must be the character data type. In C, it must be defined 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 curly brackets ({ }).

  These comment symbols represent SQL comments. For more information on SQL comment symbols, see "How to Enter SQL Comments" on page 1-9.
- The text can contain either a single SQL statement or a sequence of statements that are separated by semicolons.

  For more information on preparing a single SQL statement, see "SQL Statements Permitted in Single-Statement Prepares" on page 1-543. For more information on preparing a sequence of SQL statements, see "Preparing Sequences of Multiple SQL Statements" on page 1-549.
- Names of host-language variables are not recognized as such in prepared text.

  Therefore, you cannot prepare a SELECT statement that contains an INTO clause or an EXECUTE FUNCTION that contains 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 further information on using identifiers in statement text, see "Preparing Statements with SQL Identifiers" on page 1-546.
- Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes.

  For further information on using question marks as placeholders, see "Preparing Statements That Receive Parameters" on page 1-544.
- The text cannot include an embedded SQL statement prefix or terminator, such as a dollar sign ($) or the words EXEC SQL.

The following example shows a PREPARE statement in INFORMIX-ESQL/C that takes statement text as a quoted string:

```
EXEC SQL prepare new_cust from
    'insert into customer(fname,lname) values(?,?)';
```

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 *INFORMIX-ESQL/C Programmer's Manual*.

Preparing and Executing User-Defined Routines

The way to prepare a user-defined routine (SPL routine or external routine) depends on whether the routine is a procedure or a function:

- To prepare an SPL or an external *procedure*, prepare the EXECUTE PROCEDURE statement that executes the procedure.

  To execute the prepared procedure, use the EXECUTE statement.
- To prepare an SPL or an external *function*, prepare the EXECUTE FUNCTION statement that executes the function.

  You cannot include the INTO clause of EXECUTE FUNCTION in the PREPARE statement. The way to execute a prepared user-defined function depends on whether the function returns only one group of values or multiple groups of values. Use the EXECUTE statement for functions that return only one group of values. To execute functions that return more than one group of return values, you must associate the EXECUTE FUNCTION statement with a cursor.

For information on how to create and execute SPL routines, see Chapter 14 of the *Informix Guide to SQL: Tutorial*. For more information on how to execute user-defined routines dynamically, see the *INFORMIX-ESQL/C Programmer's Manual*.

SQL Statements Permitted in Single-Statement Prepares

You can prepare any single SQL statement *except* the ones in the following list.

ALLOCATE COLLECTION
ALLOCATE DESCRIPTOR
ALLOCATE ROW
CLOSE
CONNECT
DEALLOCATE COLLECTION
DEALLOCATE DESCRIPTOR
DEALLOCATE ROW
DECLARE
DESCRIBE
DISCONNECT
EXECUTE IMMEDIATE
EXECUTE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
INFO
LOAD
OPEN
OUTPUT
PREPARE
PUT
SET CONNECTION
SET DESCRIPTOR
UNLOAD
WHENEVER

You can prepare a SELECT statement. If the SELECT statement 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 or FOR READ ONLY clause. These clauses are normally used with the DECLARE statement to create an update cursor or read-only cursor, respectively. The following example shows a SELECT statement with a FOR UPDATE clause in INFORMIX-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 needed information is known at the time the statement is prepared. The following example in INFORMIX-ESQL/C shows two statements that are 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 INFORMIX-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. You cannot use a question-mark (?) placeholder to represent an SQL identifier except as noted in "Preparing Statements with SQL Identifiers" on page 1-546.

The following example of an INFORMIX-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 'stores7';
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 (for statements that are associated with a cursor) and EXECUTE (all other prepared statements) 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 cannot use question-mark (?) placeholders for SQL identifiers. You must specify these identifiers in the statement text when you prepare the statement.

However, in a few special cases, you can use the question mark (?) placeholder for an SQL identifier. These cases are as follows:

- You can use the *?* placeholder for the database name in the DATABASE statement.
- You can use the *?* placeholder for the dbspace name in the IN *dbspace* clause of the CREATE DATABASE statement

E/C

- You can use the *?* placeholder 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 INFORMIX-ESQL/C example prompts the user for the name of a table and uses that name in a SELECT statement. Because the table 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 using 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 'stores7';

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);
}
```

For an explanation of how to use an **sqlda** structure for statement values, see the *INFORMIX-ESQL/C Programmer's Manual*.

Preparing Sequences of 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 block.

In most situations, compiled products return error-status information on the first error in the multistatement text. No indication exists of which statement in the sequence causes an error. You can use **sqlca** to find the offset of the ESQL/C error in **sqlca.sqlerrd[4]**. For more information about **sqlca** and error-status information, see the *INFORMIX-ESQL/C Programmer's Manual*.

In a multistatement prepare, if no rows are returned from a WHERE clause in the following statements, you get SQLNOTFOUND (100) in both ANSI-compliant databases and databases that are not ANSI compliant:

- UPDATE ... WHERE ...
- SELECT INTO TEMP ... WHERE ...
- INSERT INTO ... WHERE ...
- DELETE FROM ...WHERE ...

In the following example, four SQL statements are prepared into a single INFORMIX-ESQL/C string that is 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;
```

In the preceding code fragment, the semicolons (;) are required as SQL statement-terminator symbols between each SQL statement in the text that **query** holds.

Restrictions for Multistatement Prepares

In addition to the statements listed in "SQL Statements Permitted in Single-Statement Prepares" on page 1-543, you cannot use the following statements in text that contains multiple statements that are separated by semicolons.

CLOSE DATABASE	DROP DATABASE
CREATE DATABASE	SELECT (except SELECT INTO TEMP)
DATABASE	

You cannot use regular SELECT statements in multistatement prepares. The only form of the SELECT statement allowed in a multistatement prepare is a SELECT statement with an INTO TEMP clause.

In addition, the statements that could cause the current database to be closed in the middle of executing the sequence of statements are not allowed in a multistatement prepare.

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 INFORMIX-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;
            }
    }
```

References

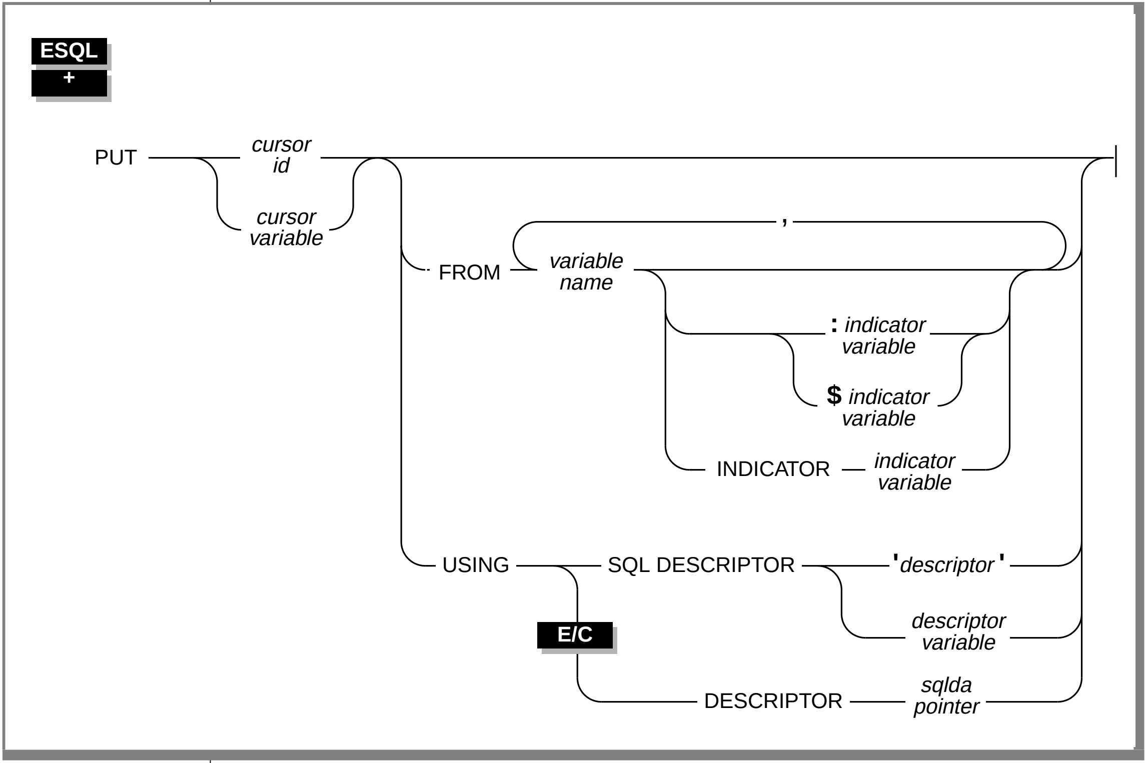
See the CLOSE, DECLARE, DESCRIBE, EXECUTE, FREE, and OPEN statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the PREPARE statement and dynamic SQL in Chapter 5.

PUT

Use the PUT statement to store a row in an insert buffer for later insertion into the database.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for an insert cursor into which the rows are to be stored	A DECLARE statement must have previously created the insert cursor and the OPEN statement must have previously open it.	Identifier, p. 1-962
cursor variable	Host variable that holds the value of *cursor id*	The host variable must be a character data type. The cursor identified in *cursor variable* must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Variable name must conform to language-specific rules for variable names.
descriptor	Quoted string that identifies the system-descriptor area that defines the type and memory location of values that correspond to the question-mark (?) placeholder in a prepared INSERT statement	The system-descriptor area must have been allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-1010
descriptor variable	Host variable name that holds the value of *descriptor*	The system-descriptor area that is identified in *descriptor variable* must have been allocated with the ALLOCATE DESCRIPTOR statement.	Variable name must conform to language-specific rules for variable names.
indicator variable	Host variable that you set to indicate that null-value data has been placed in the corresponding *variable name*	This parameter is optional, but use an indicator variable if the possibility exists that *variable name* might contain null-value data. If you specify the indicator variable without the INDICATOR keyword, you cannot put a space between *variable name* and *indicator variable*. The rules for placing a prefix before *indicator variable* are language-specific. See your SQL API manual for further information on indicator variables. Variable cannot be a DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
sqlda pointer	Points to an **sqlda** structure that defines the type and memory location of values that correspond to the question-mark (?) placeholder in a prepared INSERT statement	You cannot begin an **sqlda** pointer with a dollar sign ($) or a colon (:).	See the discussion of **sqlda** structure in the INFORMIX-ESQL / C Programmer's Manual.
variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared INSERT statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The PUT statement is one of four statements that are used for inserts that send more than one row to the database. The four statements, DECLARE, OPEN, PUT, and CLOSE, are used in the following sequence:

1. Declare an cursor to control the rows to insert.
2. Open the cursor to create the insert buffer.
3. Put the contents of each row into the cursor.
4. Close the cursor to send the rows to the database server and to break the association between the cursor and the active set.

Each PUT statement stores a row in an insert buffer that was created when *cursor name* was 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 INFORMIX-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

PUT is not an X/Open SQL statement. Therefore, you get a warning message if you compile a PUT statement in X/Open mode in INFORMIX-ESQL/C. For details on compiling in X/Open mode, see the *INFORMIX-ESQL/C Programmer's Manual*. ♦

Supplying Inserted Values

The values that reside 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 that are named in the FROM clause of the PUT statement
- Values that are prepared dynamically by an **sqlda** structure or a system-descriptor area and then named in the USING clause of the PUT statement

Using Constant Values in INSERT

The VALUES clause of the INSERT statement lists the values of the inserted columns. One or more of these values might 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 INFORMIX-ESQL/C example, 99 empty customer rows 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 the cursor declaration (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 in 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 INFORMIX-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 in the **customer** table. The VALUES clause names 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 function that is not defined in the example obtains customer information from an interactive user and leaves 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;
for (; (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. See the *INFORMIX-ESQL/C Programmer's Manual* for more information about indicator variables.

Naming Program Variables in FROM Clause of PUT

When the INSERT statement is prepared (see the PREPARE statement on page 1-538), you cannot use program variables in its VALUES clause. However, you can represent values using a question-mark (?) placeholder. List the names of program variables in the FROM clause of the PUT statement to supply the missing values.

The following INFORMIX-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 'stores7';
    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;
}
```

Using 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 insert values from a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values.

X/O

You can also use an **sqlda** structure to supply parameters dynamically. However, a system-descriptor area conforms to the X/Open standards. ♦

To specify a system-descriptor area as the location of parameters, use the USING SQL DESCRIPTOR clause of the PUT statement. Use the SET DESCRIPTOR statement to transfer the insert values for the PUT statement in the system-descriptor area. The USING SQL DESCRIPTOR clause allows you to obtain insert values from a system-descriptor area to put in an insert cursor.

The following INFORMIX-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';
```

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 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 on how to use a system-descriptor area, see the *INFORMIX-ESQL/C Programmer's Manual*.

Using an sqlda Structure

E/C

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate insert 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. To specify an **sqlda** structure as the location of parameters, use the USING DESCRIPTOR clause of the PUT statement. This clause allows you to obtain insert values from an **sqlda** structure to put into an insert cursor.

Each time the PUT statement executes, the values that the **sqlda** 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.

The following example shows the PUT ... USING DESCRIPTOR statement:

```
struct sqlda *pointer2;
...
EXEC SQL put selcurs using descriptor pointer2;
```

For further information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

Inserting into 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 INSERT statement that you associate with the cursor. Once you open the collection cursor with the OPEN statement, the cursor allows you to put elements in the **collection** variable.

For more information on the Collection Derived Table segment, see page 1-827. For more information how to declare a collection cursor for an INSERT statement, see "An Insert Cursor For a Collection Variable" on page 1-320.

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 collection column with the INSERT or UPDATE statement.*

Suppose you have a table called **children** with the following structure:

```
CREATE TABLE children
(
    age         SMALLINT,
    name        VARCHAR(30),
    fav_colors  SET(VARCHAR(20)),
)
```

The following ESQL/C code 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 code fragment saves the new collection into the **fav_colors** column of the database. Without this UPDATE statement, the collection column never has the new collection added.

Writing Buffered Rows

When the OPEN statement opens an insert cursor, an insert buffer is created. The PUT statement puts a row into this insert buffer. The block of buffered rows is 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:

- The 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 executes, naming the cursor.

  When the OPEN statement is applied to an open cursor, it closes the cursor before reopening it; this implied CLOSE statement flushes the buffer.
- A COMMIT WORK statement executes.
- The buffer contains blob 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.

Checking the Result of PUT

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 fields of the **sqlca**, as the following table shows.

ESQL/C
sqlca.sqlcode, SQLCODE
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, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The **SQLCODE** variable 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 are successfully inserted into the database:

- If a row is put into the insert buffer, and buffered rows are *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 corresponding **SQLSTATE** error value might exist. You can use the **SQLSTATE** variable to check the result of each PUT statement. You can also use the GET DIAGNOSTICS statement to examine the **RETURNED_SQLSTATE** field. See the GET DIAGNOSTICS statement in this manual for more information.*

Counting Total and Pending Rows

To count the number of rows that were actually inserted in the database and the number not yet inserted, follow these steps:

- Prepare two integer variables (for example, **total** and **pending)**.
- When the cursor is opened, set both variables to 0.
- Each time a PUT statement executes, increment both **total** and **pending**.
- 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 that were actually inserted. If all commands are successful, **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.

References

See the ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, and SET DESCRIPTOR statements in this manual for further information about using the PUT statement with dynamic management statements.

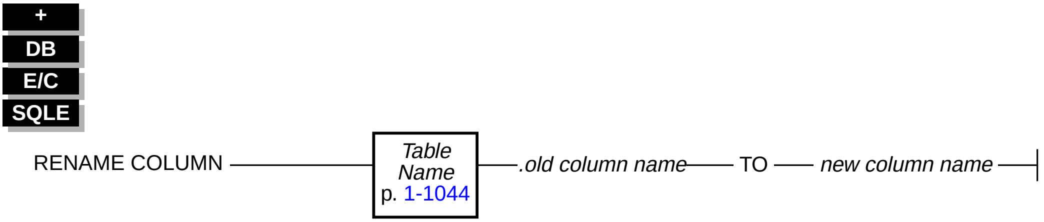
In the *Informix Guide to SQL: Tutorial*, see the discussion of the PUT statement in Chapter 6.

For further information about error checking, the system-descriptor area, and the **sqlda** structure, see the *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 name	The new name to be assigned to the column	The new name of the column must be unique within the table. If you rename a column that appears within a trigger definition, the new column name replaces the old column name in the trigger definition only if certain conditions are met. See "How Triggers Are Affected" on page 1-566 for more information on this restriction.	Identifier, p. 1-962
.old column name	The current name of the column you want to rename	The column must exist within the table. The column name must be preceded by a period. You can put a space between the table name and *.old column name*, or you can omit the space.	Identifier, p. 1-962

Usage

You can rename a column of a table if any of the following conditions are true:

- You own the table.
- You have the DBA privilege on the database.
- You have the Alter privilege on the table.

When you rename a column, choose a column name that is unique within the table.

How Views and Check Constraints Are Affected

If you rename a column that a view in the database references, the text of the view in the **sysviews** system catalog table is updated to reflect the new column name.

If you rename a column that a check constraint in the database references, 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 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, it returns an error.

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
```

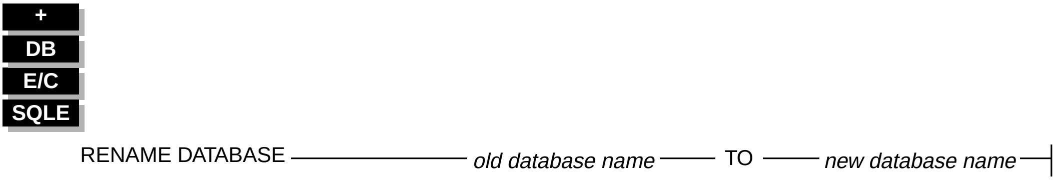
References

See the ALTER TABLE, CREATE TABLE, and RENAME TABLE statements in this manual.

RENAME DATABASE

Use the RENAME DATABASE statement to change the name of a database.

Syntax



Element	Purpose	Restrictions	Syntax
new database name	The new name that you want to assign to the database	Name must be unique. You cannot rename the current database. The database to be renamed must not be opened by any users when the RENAME DATABASE command is issued.	Database Name, p. 1-852
old database name	The name of the database that you want to rename	The database name must exist.	Database Name, p. 1-852

Usage

You can rename a database if *either* of the following statements 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 a stored procedure.

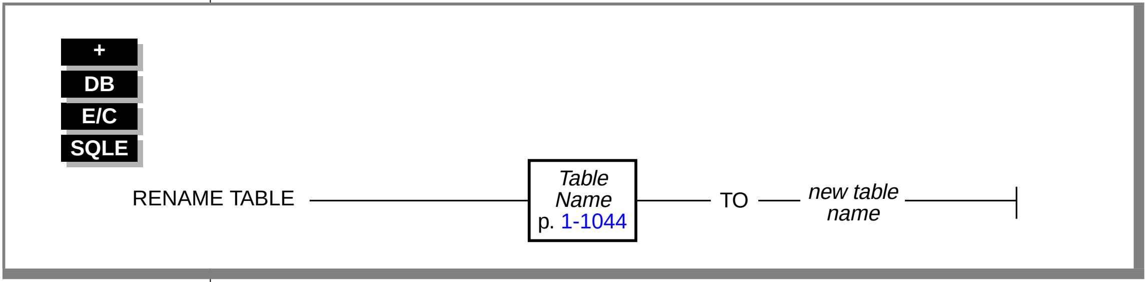
References

See the CREATE DATABASE statement in this manual.

RENAME TABLE

Use the RENAME TABLE statement to change the name of a table.

Syntax



Element	Purpose	Restrictions	Syntax
new table name	The new name that you want to assign to the table	You cannot use the *owner.* convention in the new name of the table.	Identifier, p. 1-962

Usage

You can rename a table if any of the following statements are true:

- You own the table.
- You have the DBA privilege on the database.
- You have the Alter privilege on the table.

You cannot change the table owner by renaming the table. You can use the *owner.* convention in the old name of the table, but an error occurs during compilation if you try to use the *owner.* convention in the new name of the table.

ANSI

In an ANSI-compliant database, you must use the *owner.* convention in the old name of the table if you are referring to a table that you do not own. ♦

You cannot use the RENAME TABLE statement to move a table from the current database to another database or to move a table from another database to the current database. The table that you want to rename must reside in the current database. The renamed table that results from the statement remains in the current database.

Renaming Tables That Views Reference

If a view references the table that was renamed, and the view resides in the same database as the table, the database server updates the text of the view in the **sysviews** system catalog table to reflect the new table name. See the *Informix Guide to SQL: Reference* for further information on the **sysviews** system catalog table.

Renaming Tables That Have Triggers

If you rename a table that has a trigger, it produces the following results:

- The database server replaces the name of the table in the trigger definition.
- The table name is *not* replaced where it appears inside any triggered actions.
- The database server 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.

Example of Renaming a Table

The following example reorganizes the **items** table. The intent is to move the **quantity** column from the fifth position to the third. The example illustrates 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_num   INTEGER,
    quantity    SMALLINT,
    stock_num   SMALLINT,
    manu_code   CHAR(3),
    total_price MONEY(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
```

References

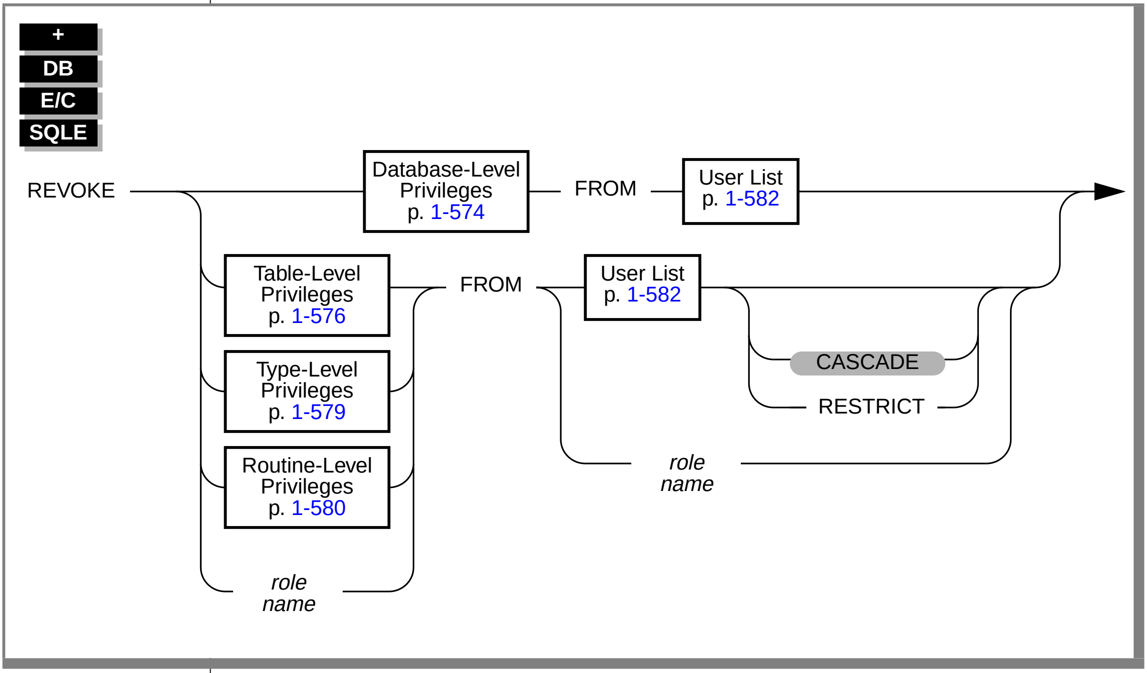
See the ALTER TABLE, CREATE TABLE, DROP TABLE, and RENAME COLUMN statements in this manual.

REVOKE

Use the REVOKE statement to cancel any of the following for specific users or for a role:

- Privileges on a database
- Privileges on a table, synonym, or view
- Privileges on a user-defined data type or routine
- A role name

Syntax


+
DB
E/C
SQLE
REVOKE
Database-Level Privileges p. 1-574
FROM
User List p. 1-582
Table-Level Privileges p. 1-576
Type-Level Privileges p. 1-579
Routine-Level Privileges p. 1-580
role name
FROM
User List p. 1-582
CASCADE
RESTRICT
role name

Element	Purpose	Restrictions	Syntax
role name	A name that identifies users by their function. Use REVOKE to remove either: ■ privileges granted to a role name. ■ specific users or another role name from those identified with the role. ■ the role name itself.	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-962

Usage

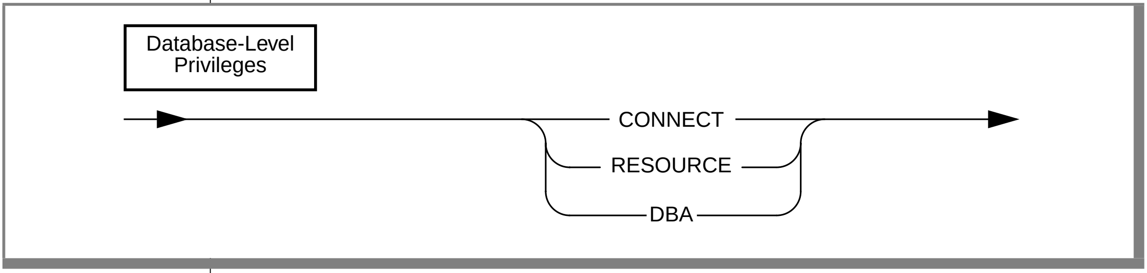
You can revoke privileges if:

- you granted them and did not name 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.

You cannot revoke privileges from yourself. You cannot revoke privileges you granted if you named another user as grantor, nor can you revoke the status as grantor from the other user.

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.



The following table lists the appropriate keyword for each database-level privilege.

DBA	If you revoke the DBA privilege from a user, you cancel the user's ability to perform the following tasks in this database: ■ Grant any database-level privilege to another user ■ Grant a role ■ Grant table-level privileges on a table that another user owns ■ Create permanent indexes on a table that another user owns ■ Change the owner of an object ■ Drop an object that another user owns ■ Execute the DROP DATABASE statement ■ Execute the SET SESSION AUTHORIZATION statement ■ Use the NEXT SIZE keyword to alter extent sizes in the system catalog tables ■ Insert, delete, or update rows of any system catalogs When you revoke the DBA privilege, you also revoke the Resource privilege.

(1 of 2)

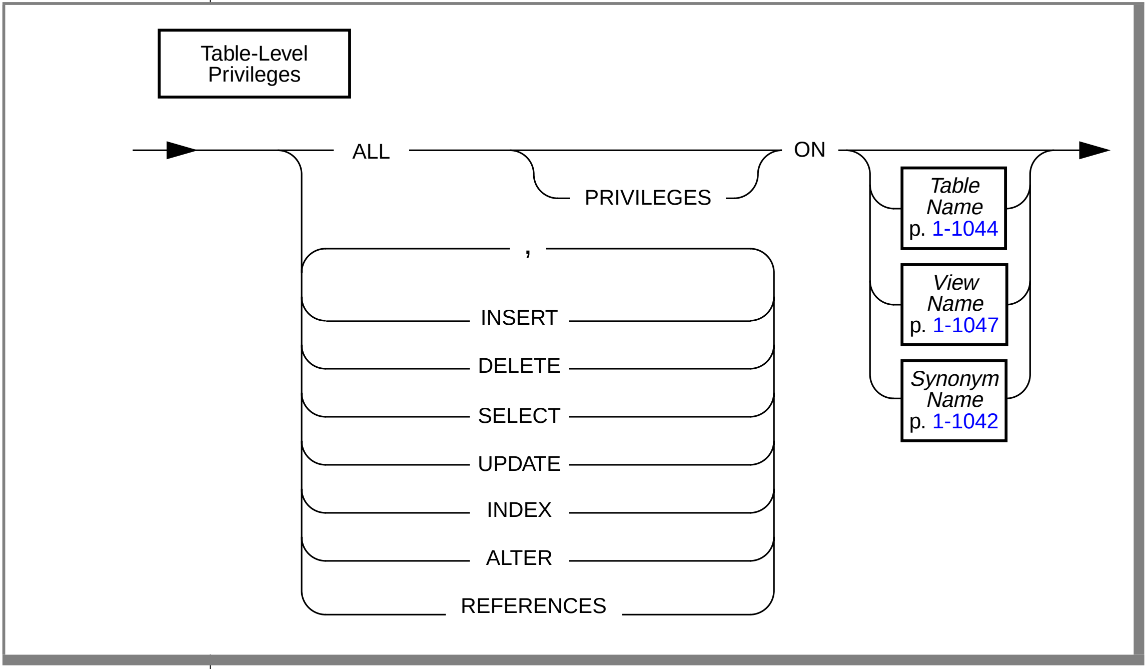
RESOURCE	If you revoke the Resource privilege from a user, you cancel that user's ability to perform the following tasks in this database: ■ Create new tables ■ Create new indexes ■ Create new routines ■ Create new data types
CONNECT	If you revoke the Connect privilege from a user, you cancel the user's ability to open the database or access any of its objects. Specifically, the user can no longer do the following: ■ Connect to the database ■ Execute SELECT, INSERT, UPDATE, and DELETE statements ■ Create views or synonyms ■ Create temporary tables and indexes on the temporary tables ■ Use privileges granted to the user, to a role, or to **public** ■ Grant privileges authorized by a GRANT statement that contained the WITH GRANT OPTION clause

(2 of 2)

You must revoke the most powerful privilege that a user has first:

- You cannot successfully revoke either the Resource or the Connect privilege from a user who still has the DBA privilege.
- When you revoke the DBA privilege, the former DBA loses the Resource privilege but retains the Connect privilege.
- You cannot successfully revoke the Connect privilege from a user who still has the Resource privilege.
- After you revoke a DBA or Resource privilege, you can revoke the Connect privilege with a separate REVOKE statement.

Table-Level Privileges



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	Functions
INSERT	Removes the ability to insert rows into a table, view, or synonym
DELETE	Removes the ability to delete rows from a table, view, or synonym
SELECT	Removes the ability to issue a SELECT statement on a table, view, or synonym
UPDATE	Removes the ability to change any column of the table, view, or synonym using UPDATE statements

(1 of 2)

Privilege	Functions
INDEX	Removes other users' ability to create permanent indexes on your table, even if those users have the Resource privilege. Using REVOKE does not remove the ability to create indexes on temporary tables, a function of the Connect database-level privilege.
ALTER	Removes the authorization to issue an ALTER statement on your table, such as the ability to: ■ add or delete columns. ■ modify column data types. ■ add or delete constraints. ■ set the object modes of indexes, constraints or triggers.
REFERENCES	Removes the ability to reference columns in your table as foreign keys. Revoke the References privilege to disallow cascading deletes.
ALL	Provides all the preceding privileges. You can optionally follow ALL with the PRIVILEGES keyword.

(2 of 2)

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 the grantees and to remove some columns from table-level privileges.

Replacing PUBLIC With Specified Users

If **public** can select from your table, you cannot revoke the Select privilege from users by name.

For example, assume **public** has default Select privileges on your **customer** table. You issue the following statement in an attempt to exclude **ted** from accessing your table:

```
REVOKE ALL ON customer TO ted
```

The REVOKE 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 the appropriate users. The following example revokes the Index and Alter privileges from all users for the **customer** table and grants 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.

In the following 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
```

The following more typical example shows how to restrict privileges for **public** to certain columns:

```
REVOKE ALL ON customer FROM PUBLIC
GRANT SELECT (fname, lname, company, city)
    ON customer TO PUBLIC
```

Behavior of the ALL Keyword

The ALL keyword revokes all table-level privileges available to the users or role specified in the REVOKE statement.

The ALL keyword can execute successfully when a user does not have a table-level privilege, but the REVOKE statement returns the following SQLSTATE code:

```
01006 - Privilege not revoked
```

For example, assume that the user **hal** has the Select and Insert privileges on the **customer** table. User **jocelyn** revokes all table-level privileges from user **hal** with the following REVOKE statement:

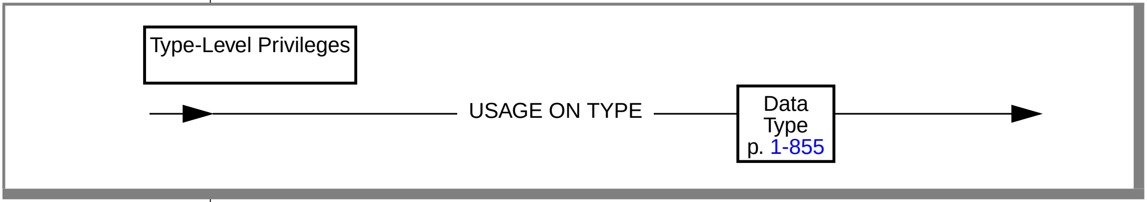
```
REVOKE ALL ON customer FROM hal
```

The statement succeeds in revoking the Select and Insert privileges from user **hal** because user **hal** had those privileges. Simultaneously, the statement alerts you that it could not revoke privileges implied by the ALL keyword that **hal** did not have, such as Delete, Update, and others.

Type-Level Privileges

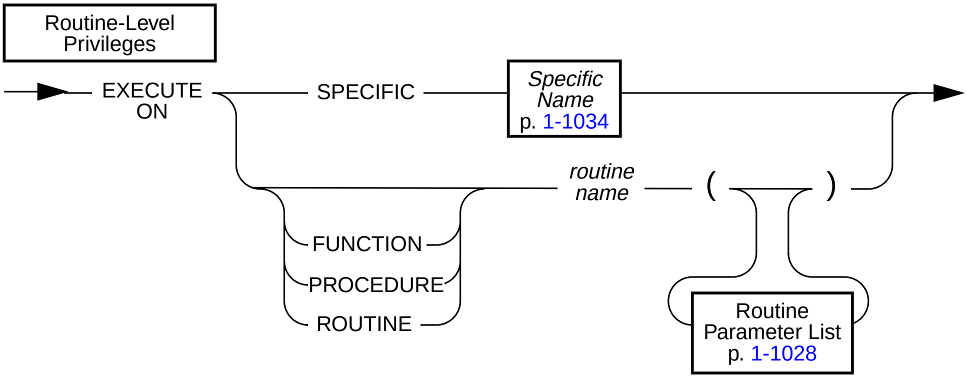
Any user can reference a built-in data type in an SQL statement, but not a distinct data type based on a built-in data type. The creator of a user-defined data type or a DBA must explicitly grant the Usage privilege on that new type, including a distinct data type based on a built-in data type.

REVOKE with the USAGE ON TYPE keywords removes the Usage privilege that you granted earlier to another user or role.



Routine-Level Privileges

The generic term *routine* refers to both a function and a procedure. A user with the Execute privilege on your routine can invoke your routine with an EXECUTE FUNCTION or EXECUTE ROUTINE statement, or a CALL statement using SPL. If you create a function, a user with the Execute privilege on your function can also use it in an expression.



Element	Purpose	Restrictions	Syntax
routine name	The name given to the user-defined routine in a CREATE FUNCTION or CREATE PROCEDURE statement	The identifier must refer to an existing user-defined routine. In an ANSI-compliant database, specify the owner as the prefix to the routine name.	Function Name, p. 1-959 or Procedure Name, p. 1-1004

When you create a routine under any of the following circumstances, you must explicitly grant the Execute privilege before you can revoke it:

ANSI

- You create a routine 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.

Commutators or negators for the routine require separate, explicit REVOKE statements if you granted the Execute privilege to them.

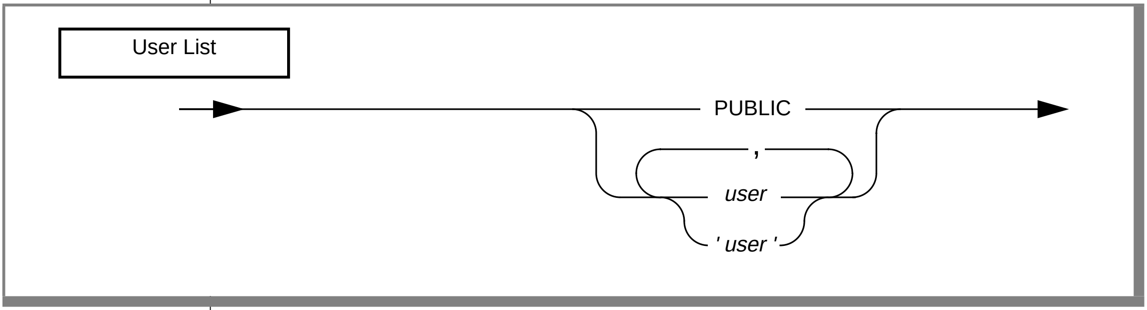
When you create a routine without any of the preceding conditions in effect, **public** can execute your routine without a GRANT statement. To limit who executes your routine, revoke the privilege using the keywords FROM PUBLIC and then grant it to a user list (see page 1-582) or role (see page 1-583).

If two or more routines have the same routine name, use the appropriate keyword from the following list to specify which of those routines a user can no longer execute.

Privilege	**Functions**
SPECIFIC	Prevents a user from executing a specific combination of the routine name and parameter list 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).

User List

In the user list, you identify who loses the privileges you are revoking. The user list can consist of a single user's login or multiple users' logins, 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	The login name to receive the role or privilege granted	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 grant a role or privilege to all authorized users.	Identifier, p. 1-962

When the user list contains specific logins, you can combine the REVOKE statement with the GRANT statement to selectively secure tables, columns, routines, types, and so forth. For examples, see "When to Use REVOKE Before GRANT" on page 1-577.

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.

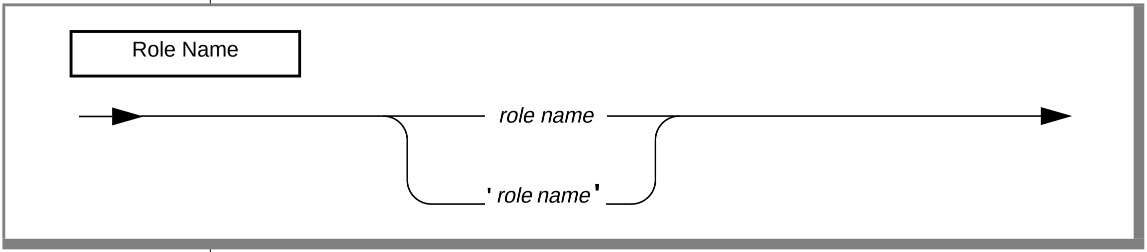
ANSI

In an ANSI-compliant database, if you do not use quotes around *user*, the name of the user is stored in uppercase letters. ♦

Role Name

Only the DBA or a user 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 1-584 explains revoking such a role.



Element	Purpose	Restrictions	Syntax
role name	Name of the role that: ■ loses a privilege assigned to it. ■ loses the use of another role. ■ a user or another role loses.	The role must have been created with the CREATE ROLE statement and granted with the GRANT statement.	Identifier, p. 1-962

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* the 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:

- 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 the user **mary**:

```
REVOKE ALL ON items FROM PUBLIC
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION
```

The 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 the users **mary**, **cathy**, and **paul**.

The CASCADE keyword has the same effect as this default condition.

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, as the following example shows:

Assume that the user **clara** uses the WITH GRANT OPTION clause to grant the Select privilege on the **customer** table to the user **ted**.

Assume that user **ted**, in turn, grants the Select privilege on the **customer** table to user **tania**. The following REVOKE statement issued by **clara** 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
```

References

See the GRANT, GRANT FRAGMENT, and REVOKE FRAGMENT statements in this manual.

For information about roles, see the CREATE ROLE, DROP ROLE, and SET ROLE statements in this manual.

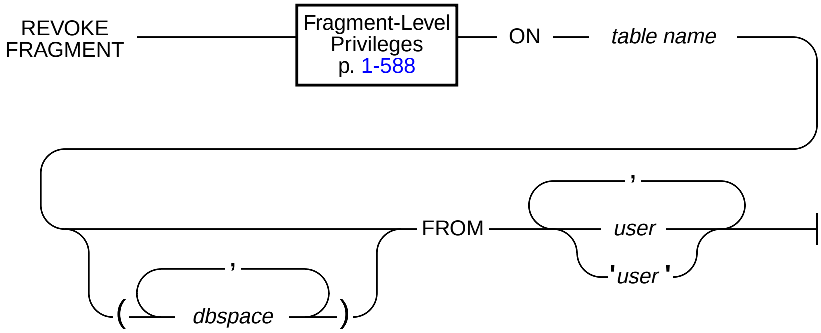
See the discussion of privileges and security in the *Informix Guide to SQL: Tutorial*.

REVOKE FRAGMENT

The REVOKE FRAGMENT statement enables you to revoke privileges that have been granted on individual fragments of a fragmented table. You can use this statement to revoke the Insert, Update, and Delete fragment-level privileges from users.

Syntax

+
DB
E/C
SQLE



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where the fragment is stored. Use this parameter to specify the fragment or fragments on which privileges are to be revoked. If you do not specify a fragment, the REVOKE statement applies to all fragments in the specified table that have the specified privileges.	The specified dbspace or dbspaces must exist.	Identifier, p. 1-962
table name	The name of the table that contains the fragment or fragments on which privileges are to be revoked. There is no default value.	The specified table must exist and must be fragmented by expression.	Table Name, p. 1-1044
user	The name of the user or users from whom the specified privileges are to be revoked. There is no default value.	The user must be a valid user.	Identifier, p. 1-962

Usage

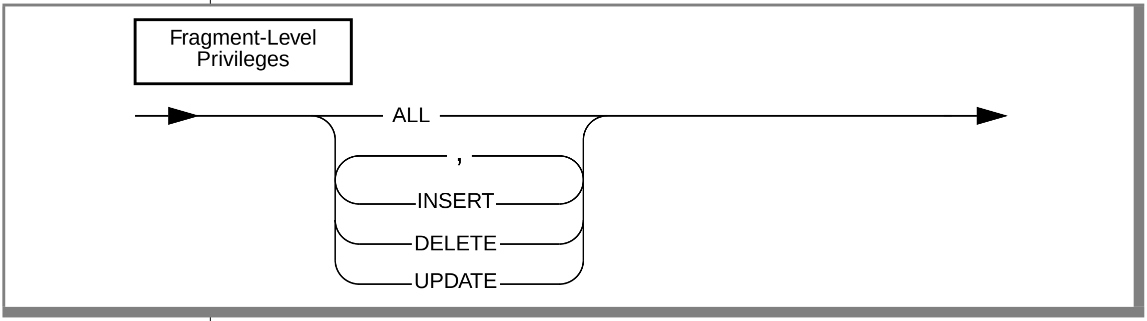
Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete privilege on one or more fragments of a fragmented table from one or more users.

The REVOKE FRAGMENT statement is only valid for tables that are fragmented according to an expression-based distribution scheme. See the ALTER FRAGMENT statement on page 1-27 for an explanation of expression-based distribution schemes.

You can specify one fragment or a list of fragments in the REVOKE FRAGMENT statement. To specify a fragment, name the dbspace in which the fragment resides.

You do not have to specify a particular fragment or a list of fragments in the REVOKE FRAGMENT statement. If you do not specify any fragments in the statement, 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 keywords are described in the following list.

Privilege	Functions
ALL	Revokes all privileges currently granted on a table fragment
INSERT	Revokes Insert privilege on a table fragment. This privilege gives the user the ability to insert rows in the fragment.
DELETE	Revokes Delete privilege on a table fragment. This privilege gives the user the ability to delete rows in the fragment.
UPDATE	Revokes Update privilege on a table fragment. This privilege gives the user the ability to update rows in the fragment and to 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 have 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.

Examples of the REVOKE FRAGMENT Statement

The examples that follow are based on the **customer** table. All the examples 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 the 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 the user **susan**:

```
REVOKE FRAGMENT UPDATE, INSERT ON customer (dbsp1) FROM susan
```

Revoking All Privileges

The following statement revokes all privileges currently granted to the 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 the 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 the 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 the user **mel** on all fragments for which this user currently has privileges:

```
REVOKE FRAGMENT ALL ON customer FROM mel
```

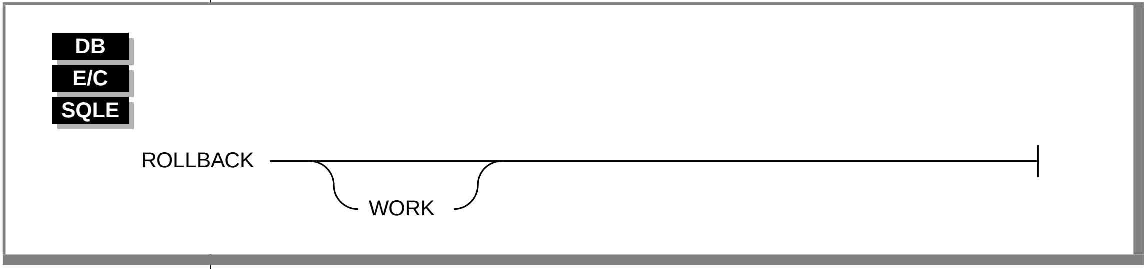
References

See the REVOKE and GRANT FRAGMENT statements in this manual.

ROLLBACK WORK

Use the ROLLBACK WORK statement to cancel a transaction and undo any changes that occurred since the beginning of the transaction.

Syntax



Usage

The ROLLBACK WORK statement is valid only in databases with transactions.

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 the ROLLBACK WORK statement 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. Transactions start after each COMMIT WORK or ROLLBACK WORK statement. If you issue a ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect. ♦

ESQL

The ROLLBACK WORK statement closes all open cursors except those that are declared with hold, which remain open despite transaction activity.

If you use the ROLLBACK WORK statement within a routine that a WHENEVER statement calls, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. Specifying these before the ROLLBACK WORK statement prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

References

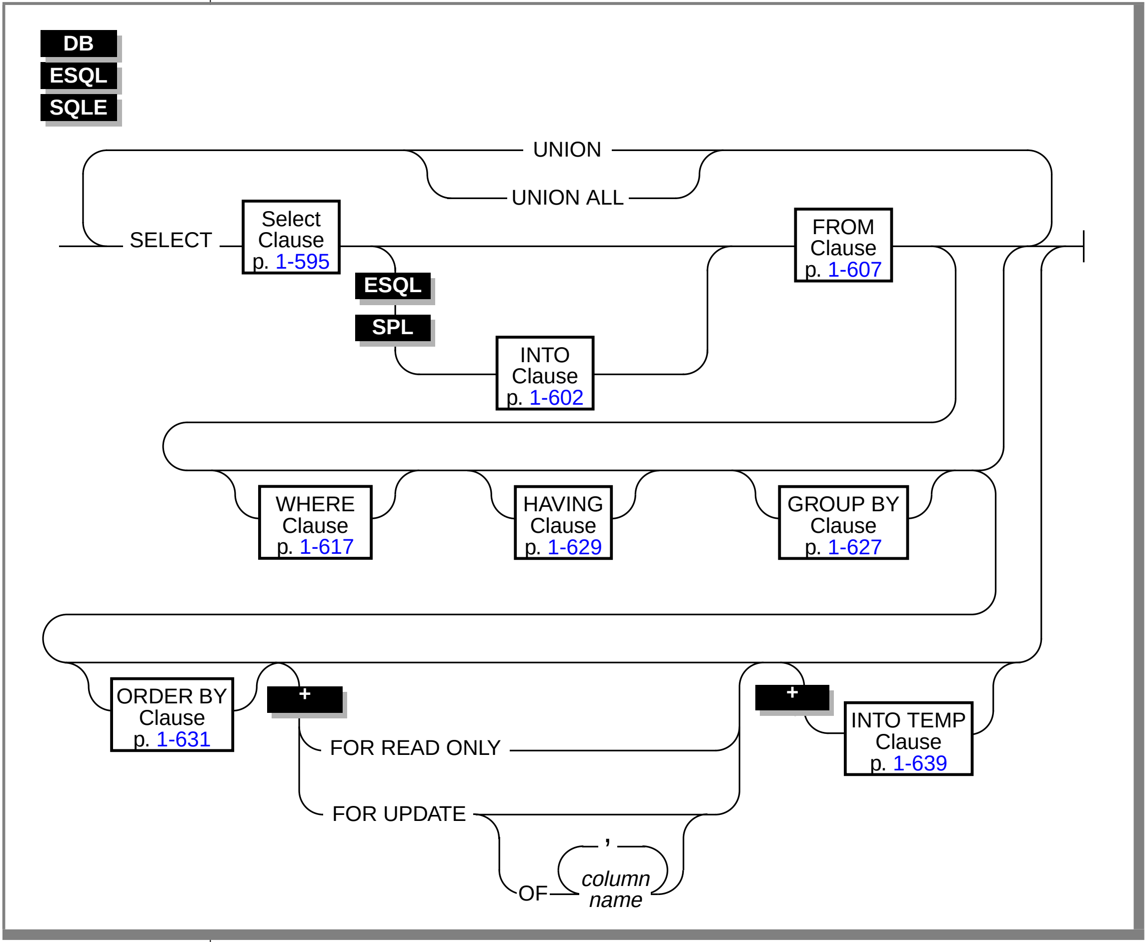
See the BEGIN WORK and COMMIT WORK statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of ROLLBACK WORK in Chapter 5.

SELECT

Use the SELECT statement to query a database or the contents of an SPL or INFORMIX-ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column that can be updated after a fetch	The specified column must be in the table, but it does not have to be in the select list of the SELECT clause.	Identifier, p. 1-962

Usage

You can query the tables in the current database, a database that is not current, or a database that is on a different database server from your current database.

The SELECT statement comprises many basic clauses. Each clause is described in the following list.

ESQL

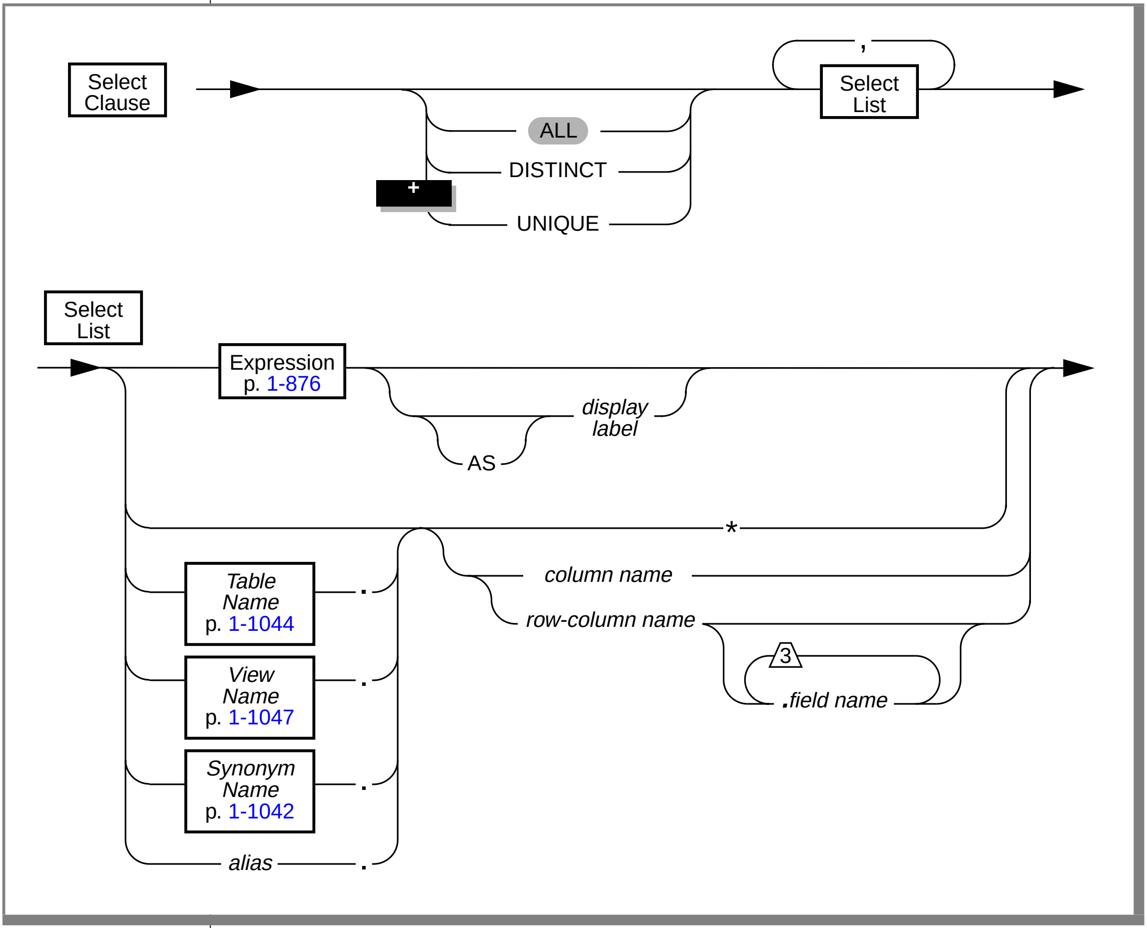
SPL

ESQL

Clause	Purpose
SELECT	Names a list of items to be read from the database
INTO	Specifies the program variables, host variables, or procedure variables that receive the selected data ♦
FROM	Names the tables that contain the selected columns
	Names the ESQL/C **collection** variable that contains the selected elements ♦
WHERE	Sets conditions on the selected rows
GROUP BY	Combines groups of rows into summary results
HAVING	Sets conditions on the summary results
ORDER BY	Orders the selected rows
INTO TEMP	Creates a temporary table in the current database and puts the results of the query into the table
FOR UPDATE	Specifies that the values returned by the SELECT statement can be updated after a fetch
FOR READ ONLY	Specifies that the values returned by the SELECT statement cannot be updated after a fetch

SELECT Clause

The SELECT clause contains the list of database objects or expressions to be selected, as shown in the following diagram.



Element	Purpose	Restrictions	Syntax
*	The asterisk (*) signifies that all columns in the specified table or view are to be selected.	Use this symbol whenever you want to retrieve all the columns in the table or view in their defined order. If you want to retrieve all the columns in some other order, or if you want to retrieve a subset of the columns, you must specify the columns explicitly in the SELECT list.	The asterisk (*) is a literal value that has a special meaning in this statement.
alias	A temporary alternative name for a table or view within the scope of a SELECT statement. You can use aliases to make a query shorter.	You must specify a table name and the table alias in the FROM clause. See "FROM Clause" on page 1-607 for further information on this restriction.	Identifier, p. 1-962
column name	The name of a column from one of the tables or views to be joined. Rows from the tables or views are joined when there is a match between the values of the specified columns.	When the specified columns have the same name in the tables or views to be joined, you must distinguish the columns by preceding each column name with the name or alias of the table or view in which the column resides.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
display label	A temporary name that you assign to a column. In DB-Access, the display label appears as the heading for the column in the output of the SELECT statement. In ESQL, the value of *display label* is stored in the **sqlname** field of the **sqlda** structure. For more information on the *display label* parameter, see "Using a Display Label" on page 1-601.	You can assign a display label to any column in your select list. If you are creating a temporary table with the SELECT...INTO TEMP clause, 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 in creating a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead. If your display label is also a keyword, you can use the AS keyword with the display label to clarify the use of the word. You must use the AS keyword with the display label to use any of the following words as a display label: UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION.	Identifier, p. 1-962
field name	The name of the row field that you are accessing in the row column	The field must be a component of the row that *row-column name* or *field name* (for nested rows) specifies.	Identifier, p. 1-962
row-column name	The name of the row column that you specify	The column must be a named row type or unnamed row type.	Identifier, p. 1-962

(2 of 2)

In the SELECT clause, specify exactly what data is being selected as well as whether you want to omit duplicate values.

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 keywords, all the rows are returned by default.

Keyword	Meaning
ALL	Specifies that all selected values are returned, regardless of whether duplicates exist. ALL is the default state.
DISTINCT	Eliminates duplicate rows from the query results
UNIQUE	Eliminates duplicate rows from the query results. UNIQUE is a synonym for DISTINCT.

For example, the following query lists the **stock_num** and **manu_code** of all items that have been ordered, excluding duplicate items:

```
SELECT DISTINCT stock_num, manu_code FROM items
```

You can use the DISTINCT or UNIQUE keywords once in each level of a query or subquery. For example, the following query uses DISTINCT in both the query and 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, function, aggregate function, and procedure), or combination thereof, in the select list. The expression types are described in “Expression” on page 1-876.

The following sections present examples of using each type 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. However, if you combine a column expression and an aggregate function, you must include the column expression in the GROUP BY clause.

You cannot use variable names (for example, host variables in an external application or SPL variables) in the select list by themselves. You can include a variable name in the select list, however, if an arithmetic or concatenation operator connects it to a constant.

Selecting Columns

Column expressions are the most commonly used expressions in a SELECT statement. See "Column Expressions" on page 1-881 for a complete description of the syntax and use of column expressions.

The following examples show column expressions within 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. See "Constant Expressions" on page 1-887 for a complete description of the syntax and use of constant expressions.

The following examples 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 Function Expressions

A function expression uses a function that is evaluated for each row in the query. All 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 function expressions within a select list:

```
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 Expressions

An aggregate function returns one value for a set of queried rows. The aggregate functions take on values that depend on the set of rows that the WHERE clause of the SELECT statement returns. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows that the FROM clause forms.

The following examples 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 SPL Function Expressions

SPL 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** function for each **customer_num** and displays the output of the function under the `n_orders` label:

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
    FROM customer
```

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. The following examples show expressions that use arithmetic operators within a select list:

```
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
```

Selecting Row Fields

You can select a particular field of a row-type column (named or unnamed row type) with dot notation, which uses 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 expressions 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;
```

For more information, see "Column Expressions" on page 1-881 in the Expression segment.

Using a Display Label

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.

DB

A display label appears as the heading for that column in the output of the SELECT statement. ♦

ESQL

The value of *display label* is stored in the **sqlname** field of the **sqlda** structure. See your SQL API product manual for more information on the **sqlda** structure. ♦

If you are using the SELECT statement in creating a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead.

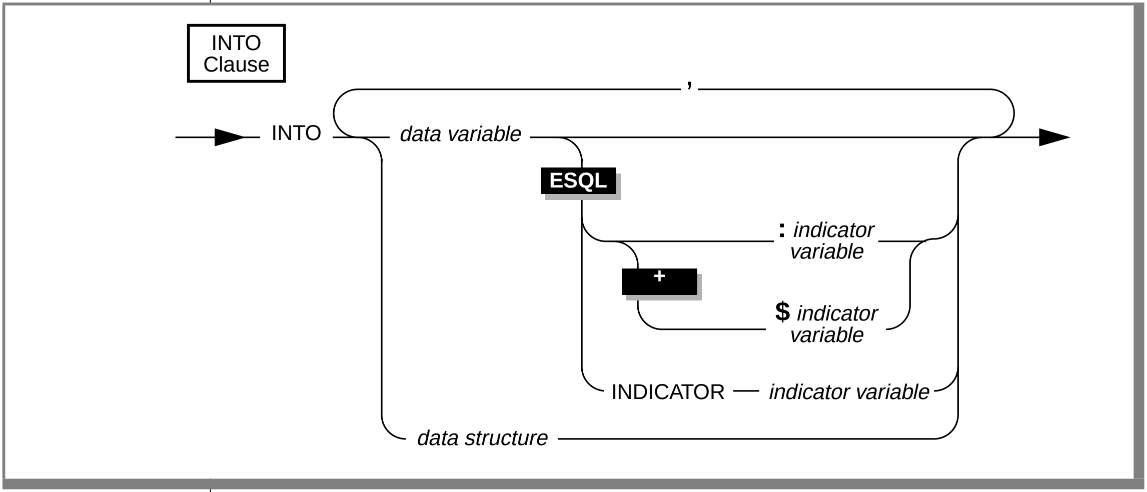
Using the AS Keyword

If your display label is also a keyword, you can use the AS keyword with the display label to clarify the use of the word. If you want to use the word UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION as your display label, you must use the AS keyword with the display label. The following example shows how to use the AS keyword with **minute** as a display label:

```
SELECT call_dtime AS minute FROM cust_calls
```

INTO Clause

Use the INTO clause within a routine to specify the program variables or host variables to receive the data that the SELECT statement retrieves. The following diagram shows the syntax of the INTO clause.



Element	Purpose	Restrictions	Syntax
data variable	A variable that receives the value returned by a function	If you issue this statement within an ESQL/C program, the *receiving variable* must be a host variable. If you issue this statement within an SPL routine, the *receiving variable* must be an SPL variable. If you issue this statement within a CREATE TRIGGER statement, the receiving variables must be column names within the triggering table or another table.	The name of a receiving variable must conform to language-specific rules for variable names. For the syntax of SPL variables, see Identifier, p. 1-962. For the syntax of column names, see Identifier, p. 1-962.
data structure	A structure that has been declared as a host variable	The individual elements of the structure must be matched appropriately to the data type of values being selected.	The name of the data structure must conform to language-specific rules for data structures.
indicator variable	A program variable that receives a return code if null data is placed in the corresponding *data variable*	This parameter is optional, but you should use an indicator variable if the possibility exists that the value of the corresponding *data variable* is null.	The name of the indicator variable must conform to language-specific rules for indicator variables.

You must specify an INTO clause with SELECT to name the variables that receive the values that the query returns. If the query returns more than one value, the values are returned into the list of variables in the order in which you specify them.

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 following example shows a SELECT statement in INFORMIX-ESQL/C:

```
EXEC SQL select fname, lname, company_name
    into :p_fname, :p_lname, :p_coname
    where customer_num = 101;
```

INTO Clause with Indicator Variables

ESQL

You should use an indicator variable if the possibility exists that data returned from the SELECT statement is null. See your SQL API product manual for more information about indicator variables. ♦

INTO Clause with Cursors

ESQL

SPL

If the SELECT statement returns more than one row, you must use a select 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 cannot put it in both. ♦

E/C

In an INFORMIX-ESQL/C program, use the DECLARE statement to declare the function cursor and the FETCH statement to fetch the rows individually from the function cursor. The following INFORMIX-ESQL/C code examples show different ways you can use the INTO clause:

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;
```

♦

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 the FOREACH statement on page 2-27. ♦

Preparing a SELECT...INTO Query

ESQL

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. ♦

Using Array Variables with the INTO Clause

ESQL

If you use a DECLARE statement with a SELECT statement that contains an INTO clause, and the program variable is an array element, you can identify individual elements of the array with integer constants or with variables. The value of the variable that is used as a subscript is determined when the cursor is declared, so afterward the subscript variable acts as a constant.

The following INFORMIX-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

ESQL

If the number of variables that are listed in the INTO clause differs from the number of items in the SELECT clause, a warning is returned in the **sqlwarn3** field of the **sqlca.sqlwarn** structure. The actual number of variables that are transferred is the lesser of the two numbers. See the *INFORMIX-ESQL/C Programmer's Manual* for information about the **sqlwarn** structure. ♦

ANSI

If the number of variables that are listed in the INTO clause differs from the number of items in the SELECT clause, you receive an error. ♦

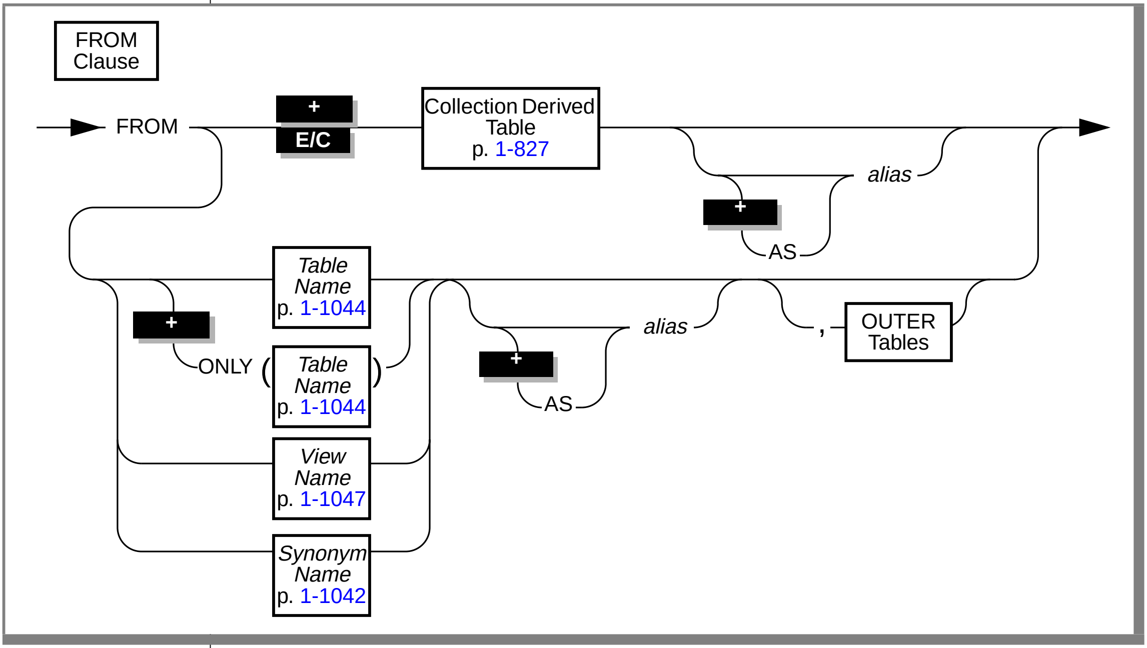
ESQL
SPL

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. If the conversion is impossible, an error occurs, and a negative value is returned in the **SQLCODE** (**sqlca.sqlcode**) status variable. In this case, the value in the program variable is unpredictable. ♦

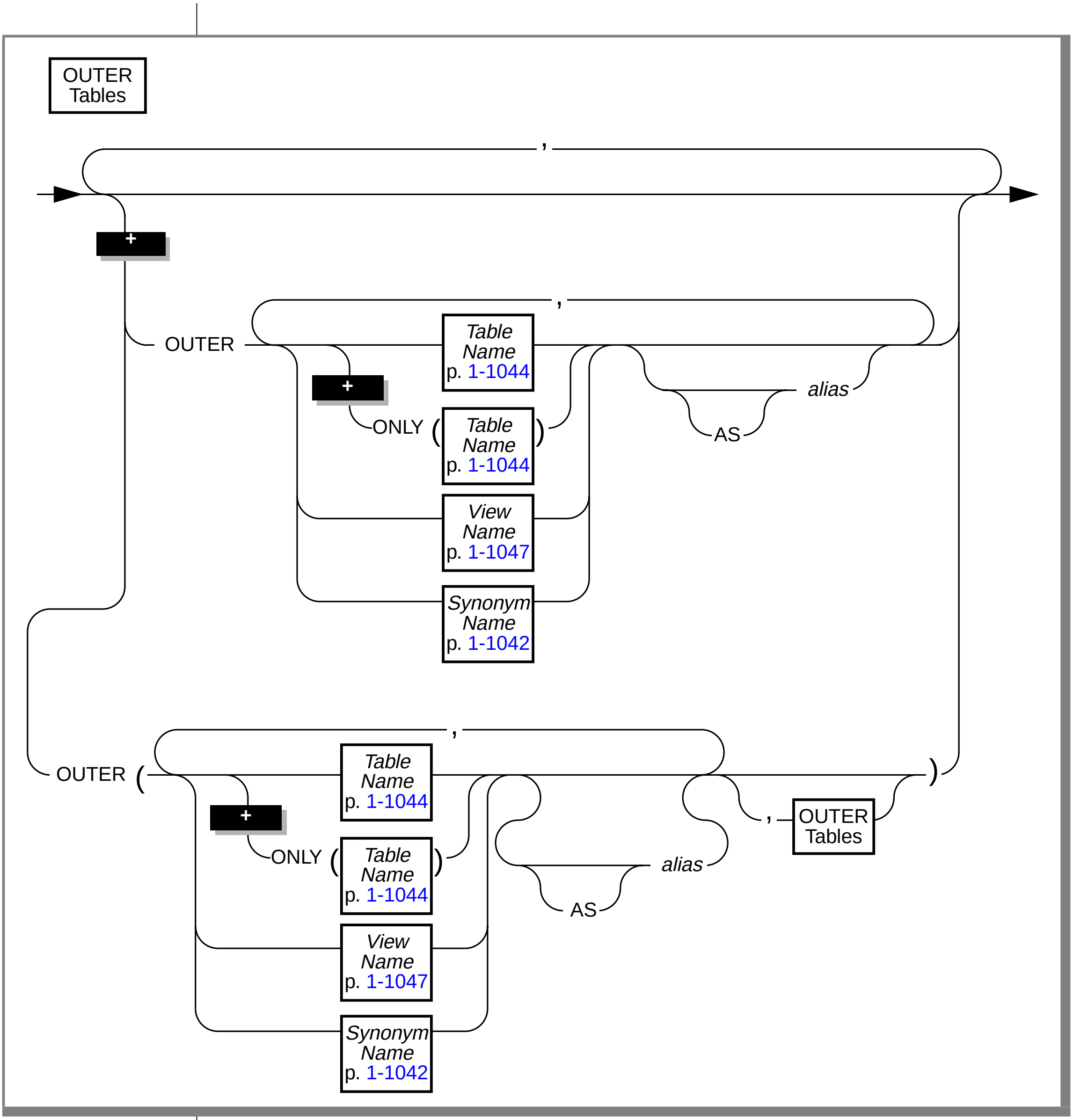
Tip: *When you encounter an* ***SQLCODE*** *error, a corresponding* ***SQLSTATE*** *error might also exist. See the GET DIAGNOSTICS statement for information about the* ***SQLSTATE*** *status variable.*

FROM Clause

The FROM clause lists the table or tables from which you are selecting the data. The following diagram shows the syntax of the FROM clause.



Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement. You can use aliases to make a query shorter.	If the SELECT statement is a self-join, you must list the table name twice in the FROM clause and assign a different alias to each occurrence of the table name. If you use a potentially ambiguous word as an alias, you must precede the alias with the keyword AS. See "AS Keyword with Aliases" on page 1-610 for further information on this restriction. Aliasing is not allowed for a collection of ROW types.	Identifier, p. 1-962


OUTER Tables
,
+
OUTER
,
Table Name p. 1-1044
+
ONLY
(
Table Name p. 1-1044
)
View Name p. 1-1047
Synonym Name p. 1-1042
alias
AS
OUTER
(
,
Table Name p. 1-1044
+
ONLY
(
Table Name p. 1-1044
)
View Name p. 1-1047
Synonym Name p. 1-1042
alias
AS
,
OUTER Tables
)

Usage

Use the keyword OUTER to form outer joins. Outer joins preserve rows that otherwise would be discarded by simple joins. See Chapter 3 of the *Informix Guide to SQL: Tutorial* for more information on outer joins.

The FROM clause cannot have a join when one of the tables to be joined is a collection.

If you use the SELECT statement to query 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 shown in the following example:

```
SELECT *
FROM ONLY(super_tab)
```

You can supply an alias for a table name or view name. You can use the alias to refer to the table or view in other clauses of the SELECT statement. This is especially useful with a self-join. (See the WHERE clause on page 1-617 for more information about self-joins.)

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 following example is the same as the second query in the preceding example, except that it establishes aliases for the tables 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
```

The following example uses the OUTER keyword to create an outer join and produce a list of all customers and their orders, regardless of whether they have placed orders:

```
SELECT c.customer_num, lname, order_num
    FROM customer c, OUTER orders o
    WHERE c.customer_num = o.customer_num
```

AS Keyword with Aliases

To use potentially ambiguous words as an alias for a table or view, you must precede them with the keyword AS. Use the AS keyword if you want to use the words ORDER, FOR, AT, GROUP, HAVING, INTO, UNION, WHERE, WITH, CREATE, or GRANT as an alias for a table or view.

E/C

SPL

Selecting From a Collection Variable

The SELECT statement 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. For more information on the Collection Derived Table segment, see page 1-827.

E/C

In an INFORMIX-ESQL/C program, declare a host variable of type **collection** for a collection variable. This **collection** variable can be typed or untyped. ♦

SPL

In an SPL routine, declare a variable of type COLLECTION, LIST, MULTISET, or SET for a collection variable. This collection variable can be typed or untyped. ♦

To select elements, follow these steps:

1. Create a collection variable in your SPL routine or ESQL/C program.
2. Optionally, fill the collection variable with elements.

   You can select a collection column into the collection variable with the SELECT statement (without the Collection Derived Table segment). Or you can insert elements into the collection variable with the INSERT statement and the Collection Derived Table segment.

3. Select a collection element from the collection variable with the SELECT statement and the Collection Derived Table segment.
4. Once the collection variable contains the correct elements, you can then use the INSERT or UPDATE statement on a table or view name to save the contents of the collection variable in a collection column (SET, MULTISET, or LIST).

The SELECT statement and the Collection Derived Table segment allow you to perform the following operations on a collection variable:

- Select *one* element from the collection

  Use the SELECT statement with the Collection Derived Table segment.

- Select *one or more* elements into the collection

  Associate the SELECT statement and the Collection Derived Table segment with a cursor to declare a collection cursor for the collection variable.

  E/C

  For information on how to use a collection cursor to select one or more elements from an ESQL/C **collection** variable, see "Associating a Cursor With a Collection Variable" on page 1-317 in the DECLARE statement. ♦

  SPL

  For information on how to use a collection cursor to select one or more elements from an SPL collection variable, see "Using a SELECT...INTO Statement" on page 2-30 of the FOREACH statement. ♦

The SELECT statement and the Collection Derived Table segment allow you to select *one* element into a collection. 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 element type of the collection.

The SELECT statement on a collection variable has the following restrictions:

- The select list of the SELECT statement cannot contain expressions.
- The select list must be an asterisk (*) if the collection contains elements of opaque, distinct, built-in, or other **collection** data types.

- Column names in the select list must be simple column names.

  These columns cannot use the following syntax:

  ```
  database@server:table.column
  ```
- The following SELECT clauses and options are not allowed: GROUP BY, HAVING, INTO TEMP, ORDER BY, WHERE, and WITH REOPTIMIZATION
- The FROM clause has no provisions to do a join.

SPL

In addition to the preceding list of restrictions, a SELECT...INTO that is associated with the FOREACH statement (called a collection query) 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*.
- You must use a collection query within a FOREACH loop.
- You cannot use the WITH HOLD option on the FOREACH statement.
- The data type of the element variable must be the same as the element type of the collection.
- The element variable can have any opaque, distinct, or collection data type, or any built-in data type except SERIAL, SERIAL8, TEXT, BYTE, CLOB or BLOB.

For more information a collection query, see the description of the FOREACH statement on page 2-27. For more information on how to use SPL routines to handle collections, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

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.

The following ESQL/C program uses a **collection** variable as a collection derived table:

```
main
{
    EXEC SQL BEGIN DECLARE SECTION;
        int a;
        client 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;
}
```

For information on how to use **collection** host variables in an ESQL/C program, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer’s Manual*. ♦

You can modify the collection variable with the Collection Derived Table segment and the INSERT, UPDATE, or DELETE statements. 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 collection variable in the collection column with one of the following statements:

- To update the collection column in the table with the collection variable, use an UPDATE statement on a table or view name and specify the collection variable in the SET clause.

  For more information, see "Updating Collection Columns" on page 1-786 in the UPDATE statement.
- To insert a collection in a column, use the INSERT statement on a table or view name and specify the collection variable in the VALUES clause.

  For more information, see "Inserting Values into Collection Columns" on page 1-501 in the INSERT statement.

Selecting From a Row Variable

E/C

The SELECT statement with the Collection Derived Table segment allows you to select fields from a **row** variable. The Collection Derived Table segment identifies the **row** variable from which to select the fields. For more information on the Collection Derived Table segment, see page 1-827.

To select fields, follow these steps:

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). Or 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 then use the INSERT or UPDATE statement on a table or view name to save the contents of the **row** variable in a row column (named and unnamed).

The SELECT statement and the Collection Derived Table segment allow you to select a particular field or group of fields in the **row** variable. The INTO clause identifies the variable that holds the field value selected from the **row** variable. The data type of the host variable in the INTO clause must be compatible with the field type.

For example, the following code fragment puts the value of the **width** field 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.
- Row columns cannot be specified in a comparison condition in a WHERE clause.
- The select list must be an asterisk (*) if the row-type contains fields of opaque, distinct, or built-in data types.
- Column names in the select list must be simple column names.

  These columns cannot use the syntax *database@server:table.column*
- The following SELECT 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. However, 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 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 1-785 in the UPDATE statement.

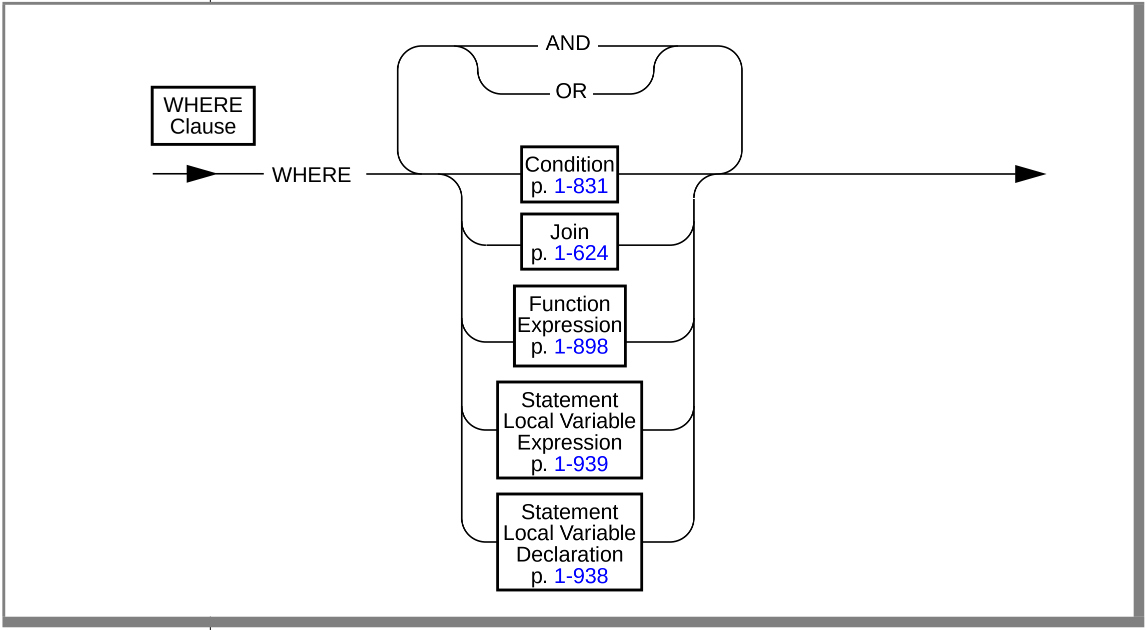
- To insert a row in a column, use the INSERT statement on a table or view name and specify the row variable in the VALUES clause.

  For more information, see "Inserting Values into Row-Type Columns" on page 1-502 in the INSERT statement.

For more information on how to use SPL row variables, see Chapter 14 of the *Informix Guide to SQL: Tutorial*. For more information on how to use ESQL/C **row** variables, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

WHERE Clause

Use the WHERE clause to specify search criteria and join conditions on the data that you are selecting.


WHERE Clause
WHERE
AND
OR
Condition p. 1-831
Join p. 1-624
Function Expression p. 1-898
Statement Local Variable Expression p. 1-939
Statement Local Variable Declaration p. 1-938

Using a Condition in the WHERE Clause

You can use the following kinds of simple conditions or comparisons in the WHERE clause:

- Relational-operator condition
- BETWEEN
- IN
- IS NULL
- LIKE or MATCHES

You also can use a SELECT statement within the WHERE clause; using a SELECT statement this way is called a subquery. The following list contains the kinds of subquery WHERE clauses:

- IN
- EXISTS
- ALL / ANY / SOME

Examples of each type of condition are shown in the following sections. For more information about each kind of condition, see the Condition segment on page 1-831.

You cannot use an aggregate function in the WHERE clause unless it is part of a subquery or if the aggregate is on a correlated column that originates from a parent query and the WHERE clause is within a subquery that is within a HAVING clause.

Relational-Operator Condition

For a complete description of the relational-operator condition, see page 1-836.

A relational-operator condition is satisfied when the expressions on either side of the relational operator fulfill the relation that the operator set up. The following SELECT statements use the greater than (>) and equal (=) relational operators:

```
SELECT order_num FROM orders
    WHERE order_date > '6/04/94'

SELECT fname, lname, company
    FROM customer
    WHERE city[1,3] = 'San'
```

BETWEEN Condition

For a complete description of the BETWEEN condition, see page 1-837.

The BETWEEN condition is satisfied when the value to the left of the BETWEEN keyword lies in the inclusive range of the two values on the right of the BETWEEN keyword. The first two queries in the following example use literal values after the BETWEEN keyword. The third query uses the CURRENT function and a literal interval. It looks 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 '6/1/93' AND '9/1/93'

SELECT * FROM cust_calls WHERE call_dtime
    BETWEEN (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT
```

IN Condition

For a complete description of the IN condition, see page 1-838.

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 )
```

IS NULL Condition

For a complete description of the IS NULL condition, see page 1-840.

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
```

LIKE or MATCHES Condition

For a complete description of the LIKE or MATCHES condition, see page 1-840.

The LIKE or MATCHES condition is satisfied when either of the following tests 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 following 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 (%). The backslash (\) is used as the standard escape character for the wildcard percent sign (%). 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 character is used as an escape character for the wildcard 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 standard escape character for the wildcard asterisk (*). 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 used as an escape character for the wildcard asterisk (*).

```
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'
```

IN Subquery

For a complete description of the IN subquery, see page 1-845.

With the IN subquery, more than one row can be returned, but only one column can be returned. The following 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)
```

EXISTS Subquery

For a complete description of the EXISTS subquery, see page 1-846.

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 * statement in the subquery in place of the column names because you are testing for the existence of a row or rows.

ALL/ANY/SOME Subquery

For a complete description of the ALL / ANY / SOME subquery, see page 1-846.

In the following example, the SELECT statements 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 statement uses the ALL subquery, and the second SELECT statement 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)
```

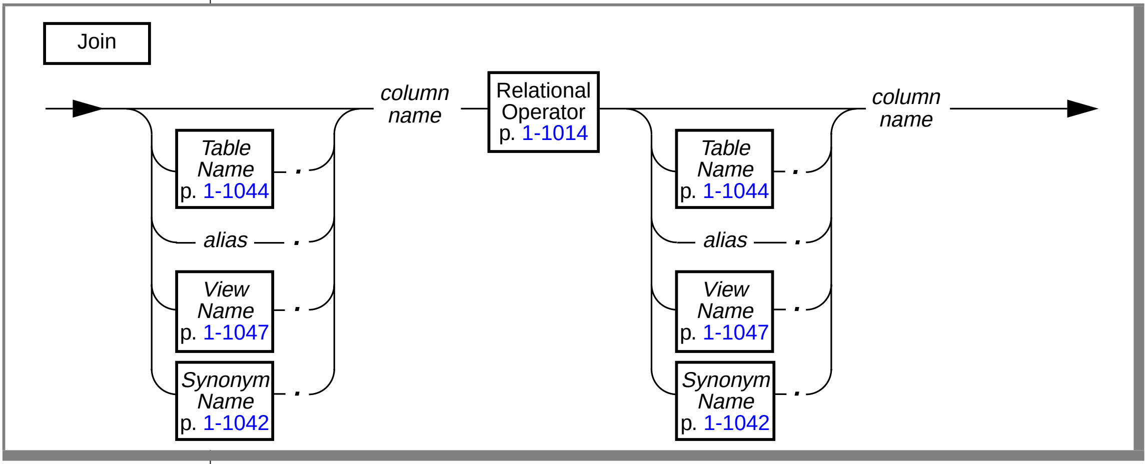
You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery returns exactly one value. If you omit ANY, ALL, or SOME, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
    WHERE stock_num = 9 AND quantity =
        (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

Using a Join in the WHERE Clause

You join two tables when you create a relationship in the WHERE clause between at least one column from one table and at least one column from another table. The effect of the join is to create 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. You can create two-table joins, multiple-table joins, and self-joins.

The following diagram shows the syntax for a join.



Element	Purpose	Restrictions	Syntax
alias	The alias assigned to the table or view in the FROM clause. See "FROM Clause" on page 1-607 for more information on aliases for tables and views.	If the tables to be joined are the same table (that is, if the join is a self-join), you must refer to each instance of the table in the WHERE clause by the alias assigned to that table instance in the FROM clause.	Identifier, p. 1-962
column name	The name of a column from one of the tables or views to be joined. Rows from the tables or views are joined when there is a match between the values of the specified columns.	When the specified columns have the same name in the tables or views to be joined, you must distinguish the columns by preceding each column name with the name or alias of the table or view in which the column resides.	Identifier, p. 1-962

Two-Table Joins

The following example shows a two-table join:

```
SELECT order_num, lname, fname
    FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
```

Tip: *You do not have to select the column where the two tables are joined.*

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 distinguish them by preceding the name with its associated table or table alias, as in *table.column*. See "Table Name" on page 1-1044 for the full syntax of a table name.

The following multiple-table join yields the company name of the customer who ordered an item as well as the stock number and manufacturer code of the item:

```
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 following 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
```

Outer Joins

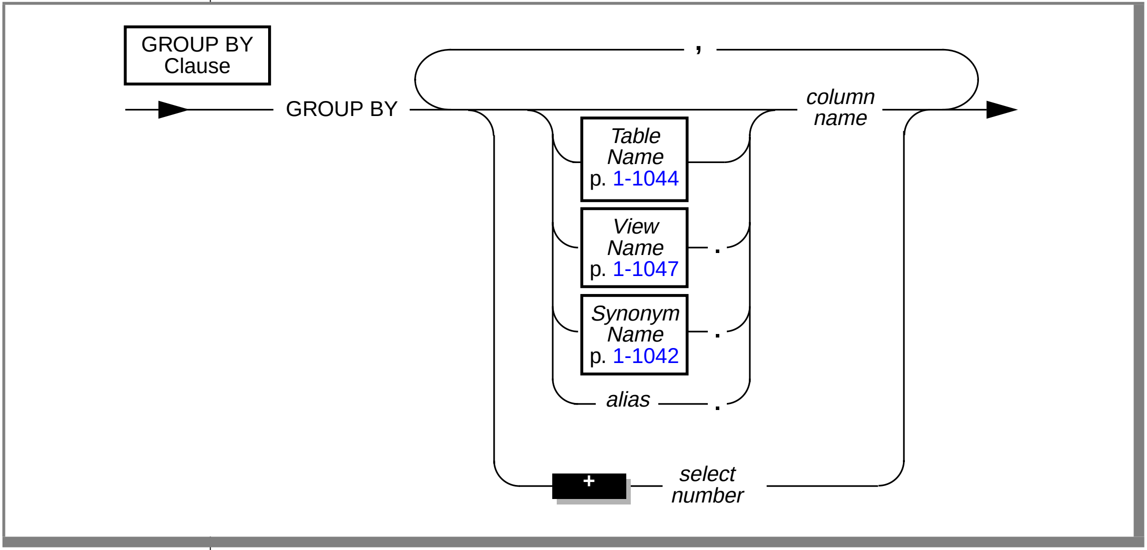
The following outer join lists the company name of the customer and all associated order numbers, if the customer has placed an order. If not, the company name is still listed, and a null value is returned for the order number.

```
SELECT company, order_num
    FROM customer c, OUTER orders o
    WHERE c.customer_num = o.customer_num
```

See Chapter 3 of the *Informix Guide to SQL: Tutorial* for more information about outer joins.

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
alias	The alias assigned to a table or view in the FROM clause. See "FROM Clause" on page 1-607 for more information on aliases for tables and views.	You cannot use an alias for a table or view in the GROUP BY clause unless you have assigned the alias to the table or view in the FROM clause.	Identifier, p. 1-962
column name	The name of a stand-alone column in the select list of the SELECT clause or the name of one of the columns joined by an arithmetic operator in the select list. The SELECT statement returns a single row of results for each group of rows that have the same value in *column name*.	See "Relationship of the GROUP BY Clause to the SELECT Clause" on page 1-628. You cannot use a column whose data type is a collection in the GROUP BY clause.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
select number	An integer that identifies a column or expression in the select list of the SELECT clause by specifying its order in the select list. The SELECT statement returns a single row of results for each group of rows that have the same value in the column or expression identified by *select number*.	See "Using Select Numbers" on page 1-629.	Literal Number, p. 1-997

(2 of 2)

Relationship of the GROUP BY Clause to the SELECT Clause

A GROUP BY clause restricts what you can enter in the SELECT clause. If you use a GROUP BY clause, each column that you select must be in the GROUP BY list. 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. Do not put constant expressions or BYTE or TEXT column expressions in the GROUP BY list.

If you are selecting a BYTE or TEXT column, you cannot use the GROUP BY clause. In addition, you cannot use ROWID in a GROUP BY clause.

If your select list includes a column with a user-defined data type, the type must either use the database server's built-in bit-hashing function or have its own hash function. Otherwise, you cannot use 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 keywords are applied to each group, not 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 a specific number.

Using Select Numbers

You can use one or more integers in the GROUP BY clause to stand for column expressions. In the following 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-number notation. In the second SELECT statement, you cannot replace the 2 with the expression **paid_date - order_date**.

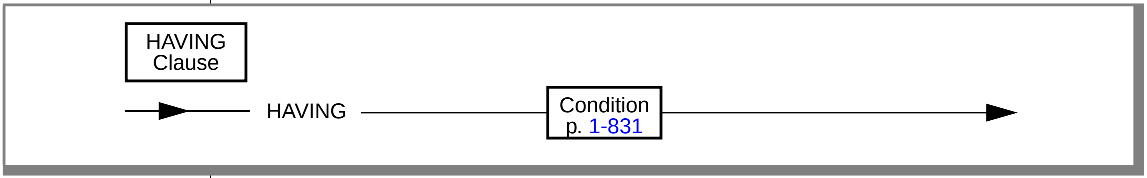
```
SELECT order_date, COUNT(*), paid_date - order_date
    FROM orders
    GROUP BY 1, 3

SELECT order_date, paid_date - order_date
    FROM orders
    GROUP BY order_date, 2
```

Nulls in the GROUP BY Clause

Each row that contains a null value in a column that is specified by a GROUP BY clause belongs to a single group (that is, all null values are grouped together).

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 if they 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. The following example returns the **customer_num**, **call_dtime** (in full year-to-fraction format), and **cust_code**, and groups them by **call_code** for all calls that have been received from customers with **customer_num** less than 120:

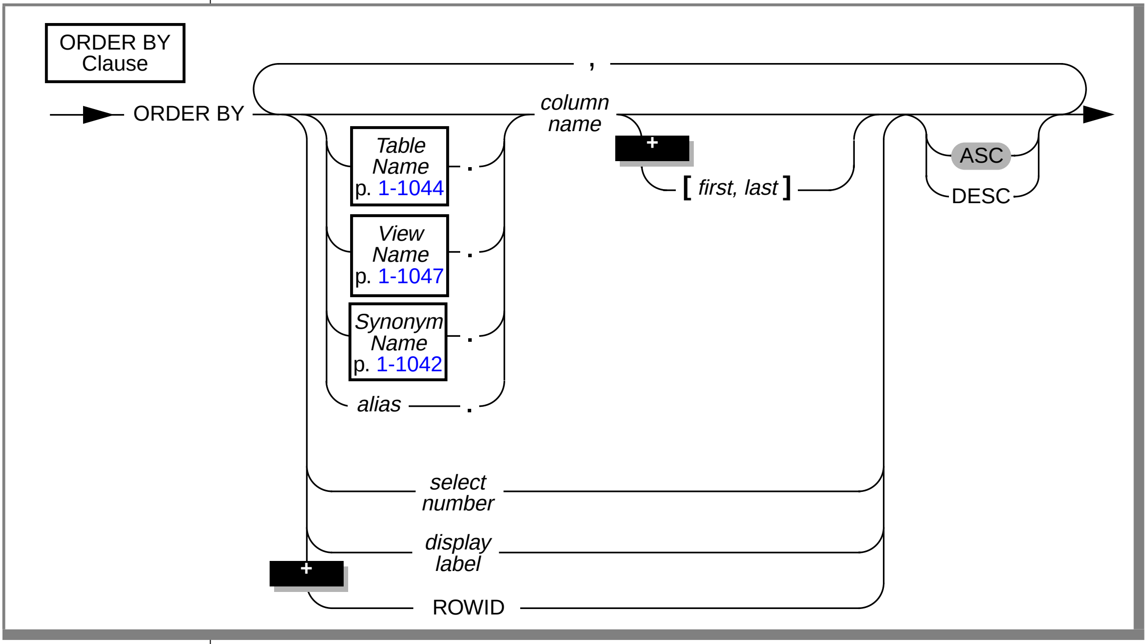
```
SELECT customer_num, EXTEND (call_dtime), call_code
    FROM cust_calls
    GROUP BY call_code, 2, 1
    HAVING customer_num < 120
```

The HAVING clause generally complements a GROUP BY clause. If you use a HAVING clause without a GROUP BY clause, the HAVING clause applies to all rows that satisfy the query. Without a GROUP BY clause, 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

Use the ORDER BY clause to sort query results by the values that are contained in one or more columns.



Element	Purpose	Restrictions	Syntax
alias	The alias assigned to a table or view in the FROM clause. See "FROM Clause" on page 1-607 for more information on aliases for tables and views.	You cannot specify an alias for a table or view in the ORDER BY clause unless you have assigned the alias to the table or view in the FROM clause.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
column name	The name of a column in the specified table or view. The query results are sorted by the values contained in this column.	A column specified in the ORDER BY clause must be listed explicitly or implicitly in the select list of the SELECT clause. If you want to order the query results by a derived column, you must supply a display label for the derived column in the select list and specify this label in the ORDER BY clause. Alternatively, 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. This cannot be a column whose data type is a collection.	Identifier, p. 1-962
display label	A temporary name that you assign to a column in the select list of the SELECT clause. You can use a display label in place of the column name in the ORDER BY clause.	You cannot specify a display label in the ORDER BY clause unless you have specified this display label for a column in the select list.	Identifier, p. 1-962
first	The position of the first character in the portion of the column that is used to sort the query results	The column must be one of the following character types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-997
last	The position of the last character in the portion of the column that is used to sort the query results	The column must be one of the following character types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-997
select number	An integer that identifies a column in the select list of the SELECT clause by specifying its order in the select list. You can use a select number in place of a column name in the ORDER BY clause.	You must specify select numbers in the ORDER BY clause when SELECT statements are joined by UNION or UNION ALL keywords and compatible columns in the same position have different names.	Literal Number, p. 1-997

(2 of 2)

You can perform an ORDER BY operation on a column or on an aggregate expression when you use SELECT * or a display label in your SELECT statement.

The following query explicitly selects the order date and shipping date from the **orders** table and then rearranges the query by the order date. By default, the query results are listed in ascending order.

```
SELECT order_date, ship_date FROM orders
    ORDER BY order_date
```

In the following query, the **order_date** column is selected implicitly by the SELECT * statement, so you can use **order_date** in the ORDER BY clause:

```
SELECT * FROM orders
    ORDER BY order_date
```

Ordering by a Column Substring

You can order by a column substring instead of ordering by the entire length of the column. The column substring is the portion of the column that the database server uses for the sort. You define the column substring by specifying column subscripts (the *first* and *last* parameters). The column subscripts represent the starting and ending character positions of the column substring.

The following example shows a SELECT statement that 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 positions of the column.

```
SELECT * from customer
    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`, not the value `Greenburg`.

You can specify column substrings only for columns that have 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.

GLS

For information on the GLS aspects of using column substrings in the ORDER BY clause, see the *Guide to GLS Functionality*. ♦

Ordering by a Derived Column

You can order by a derived column by supplying a display label in the SELECT clause, as shown in the following example:

```
SELECT paid_date - ship_date span, customer_num
    FROM orders
    ORDER BY span
```

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 standard character data types, the ASCII collating sequence is used. See page 1-1017 for a listing of the collating sequence.

Nulls in the ORDER BY Clause

Null values are ordered as less than values that are not null. Using the ASC order, the null value comes before the non-null value; using DESC order, the null value 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 one or more integers that refer to the position of items in the SELECT clause. You can use a select number to order by an expression. For instance, 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 and compatible columns in the same position have different names.

Ordering by Rowids

You can specify the **rowid** column as a column in the ORDER BY clause. The **rowid** column is 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. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the **rowid** column.

If you want to specify the **rowid** column in the ORDER BY clause, enter the keyword ROWID in lowercase or uppercase letters. You cannot specify the **rowid** column in the ORDER BY clause:

- if the table from which you are selecting is a fragmented table that does not have a rowid column.
- unless you have included the **rowid** column in the select list of the SELECT clause.

For further information on using the **rowid** column in column expressions, see "Expression" on page 1-876.

ORDER BY Clause with DECLARE

ESQL

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 further information on creating indexes that correspond to the columns of an ORDER BY clause, see "ASC and DESC Keywords" on page 1-142 under the CREATE INDEX statement.

FOR UPDATE Clause

Use the FOR UPDATE clause when you prepare a SELECT statement, and you intend to update the values returned by the 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.

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 particular columns that 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 particular value, your program might contain statements such as the sequence of statements shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char fname[ 16];
    char lname[ 16];
    EXEC SQL END DECLARE SECTION;
.
.
.
EXEC SQL connect to 'stores7';
 /* 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; --note no 'for update' clause in declare
```

```
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 interrogate 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 TEMP, 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 page 1-300.

FOR READ ONLY Clause

Use the FOR READ ONLY clause 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. A SELECT statement is a read-only operation by definition, so the FOR READ ONLY clause is usually unnecessary. However, in certain special circumstances, you must include the FOR READ ONLY clause 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-mode 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.

When the database is an ANSI-mode 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, the following 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. Therefore, you must include the FOR READ ONLY clause in all SELECT statements that access data in a read-only ANSI-mode database. The FOR READ ONLY clause causes DB-Access to declare the cursor for the SELECT statement as a read-only cursor. ♦

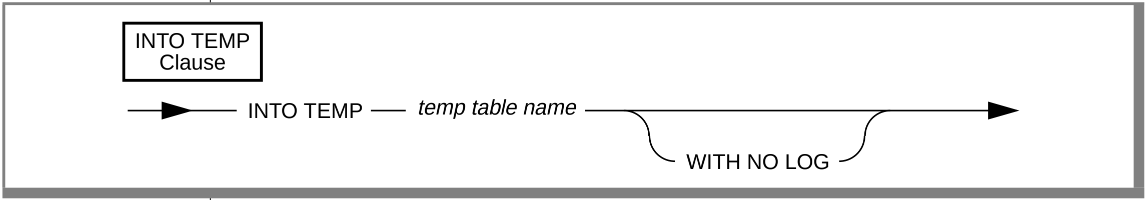
For more information on the express mode of HPL, see the *Guide to the High-Performance Loader*. For more information on level-0 backups, see the *INFORMIX-Universal Server Archive and Backup Guide*. For more information on select cursors, read-only cursors, and update cursors, see the DECLARE statement on page 1-300.

Syntax That Is Incompatible with the FOR READ ONLY Clause

Whether your database is an ANSI-mode database or a database that is not ANSI compliant, you cannot include both the FOR READ ONLY clause and the FOR UPDATE clause in the same SELECT statement. If you attempt to do so, the SELECT statement fails.

For information on how to declare a read-only cursor for a SELECT statement that does not include a FOR READ ONLY clause, see page 1-300.

INTO TEMP Clause



Element	Purpose	Restrictions	Syntax
temp table name	The simple name of a temporary table. This table contains the results of the SELECT statement. The column names of the temporary table are those that are named in the select list of the SELECT clause.	The name must be different from any existing table, view, or synonym name in the current database, but it does not have to be different from other temporary table names used by other users. You must have the Connect privilege on a database to create a temporary table in that database. If you use the INTO TEMP clause to create a temporary table, you must supply a display label for all expressions in the select list other than simple column expressions.	Identifier, p. 1-962

The INTO TEMP clause creates a temporary table that contains the query results. The initial and next extents for the temporary table are always eight pages. The temporary table must be accessible by the database server's built-in RSAM access method; you cannot specify an alternate access method.

Temporary tables created with the INTO TEMP clause are *explicit* temporary tables. Explicit temporary tables can also be created with the CREATE TEMP TABLE statement.

If the **DBSPACETEMP** environment variable is set for INFORMIX-Universal Server, temporary tables created with the INTO TEMP clause are located in the dbspaces that are specified in the **DBSPACETEMP** list. You can also specify dbspace settings with the ONCONFIG parameter **DBSPACETEMP**. If neither the environment variable nor configuration parameter is set, the default setting is the root dbspace. The settings specified for the **DBSPACETEMP** environment variable take precedence over the ONCONFIG parameter **DBSPACETEMP** and the default setting. For more information about creating temporary tables, see the CREATE TABLE statement on page 1-208. For more information about the **DBSPACETEMP** environment variable, see Chapter 3 of the *Informix Guide to SQL: Reference*. For more information about the ONCONFIG parameter **DBSPACETEMP**, see the *INFORMIX-Universal Server Administrator's Guide*.

If a temporary table is created with logging, it does not disappear automatically when your program ends; you must issue a DROP TABLE statement on the temporary table. If your database does not have logging, or if it has logging, and you created the temporary table without the WITH NO LOG keywords, the temporary table disappears when you close the current database.

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. However, the data in the temporary table is static; data is not updated as changes are made to the tables used to build the temporary table.

The column names of the temporary table are those named in the SELECT clause. You must supply a display label for all expressions other than simple column expressions. The display label for a column or expression becomes the column name in the temporary table. If you do not provide a display label for a column expression, the temporary table uses the column name from the select list. The following 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
```

You can put indexes on a temporary table.

INTO TEMP Clause and WHERE Clause

When you use the INTO TEMP 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.

INTO TEMP Clause and INTO

ESQL

Do not use the INTO option with the INTO TEMP clause. If you do, no results are returned to the program variables and the SQLCODE (**sqlca.sqlcode**) variable is set to a negative value. ♦

WITH NO LOG Option

If you use the WITH NO LOG keywords, operations on the temporary table are not included in the transaction-log operations. You can use this option to reduce the overhead of transaction logging.

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.

Restrictions on Combined SELECT

Several restrictions apply on the queries that you can connect with a UNION operator, as the following list describes:

- The number of items in the SELECT clause of each query must be the same, and the corresponding items in each SELECT clause must have compatible data types.
- If you use an ORDER BY clause, it must follow the last SELECT clause, and you must refer to the item ordered by integer, not by identifier. Ordering takes place after the set operation is complete.
- You cannot use a UNION operator inside a subquery or in the definition of a view.
- ESQL You cannot use an INTO clause in a query unless you are sure that the compound query returns exactly one row, and you are not using a cursor. In this case, the INTO clause must be in the first SELECT statement. ♦

To put the results of a UNION operator into a temporary table, use an INTO TEMP clause 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 following example uses the UNION ALL operator 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 1993 and the first quarter of 1994.

```
SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1993-1-1) YEAR TO DAY
        AND DATETIME (1993-3-31) YEAR TO DAY

UNION ALL

SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1994-1-1)YEAR TO DAY
        AND DATETIME (1994-3-31) 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 SELECT clause, as described on page 1-598.)

References

In this manual, see the descriptions of the DECLARE and FOREACH statements.

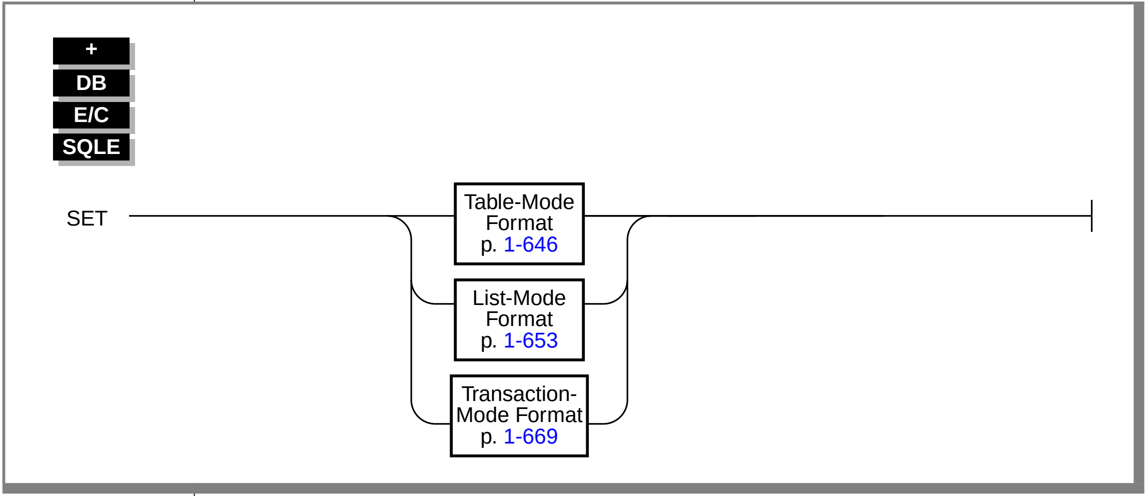
In the *Informix Guide to SQL: Tutorial*, see the discussion of the SELECT statement in Chapter 2 and Chapter 3, and Chapter 14 for the discussion of SPL routines. In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the SELECT statement.

For information on how to access row and collections with ESQL/C host variables, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*.

SET

The SET statement allows you to change the object mode of the following database objects: constraints, indexes, and triggers. You can also use the SET statement to specify the transaction mode of constraints.

Syntax


+
DB
E/C
SQLE
SET
Table-Mode Format p. 1-646
List-Mode Format p. 1-653
Transaction-Mode Format p. 1-669

Usage

The SET statement has the following purposes:

- To change the object mode of constraints, indexes, and triggers

  When you change the object mode of constraints, indexes, or triggers, the change is permanent. The setting that the SET statement produces remains in effect until you change the object mode of the object again.
- To set the transaction mode of constraints by specifying whether constraints are checked at the statement level or at the transaction level

  When you set the transaction mode of constraints, the effect of the SET statement is limited to the transaction in which it is executed. The setting that the SET statement produces is effective only during the transaction. For further information on setting the transaction mode for constraints, see "Transaction-Mode Format" on page 1-669.

Terminology for Object Modes

The SET statement operates on database objects by changing the object mode of those objects. The terms *database objects* and *objects* have a restricted meaning in the context of the SET statement. Both terms refer to the constraints, indexes, and triggers in a database.

Similarly, the term *object modes* has a restricted meaning in the context of the SET statement. The term refers to the three states that a database object can have: enabled, disabled, and filtering. The **sysobjstate** system catalog table lists all of the objects in the database and the current object mode of each object.

Do not confuse the terms *objects* and *object modes* as used in the SET statement with the term *objects* in INFORMIX-NewEra. In the context of INFORMIX-NewEra, *objects* refers to objects within an application.

Methods for Changing Object Modes

The SET statement provides the following formats for changing object modes: table mode and list mode. For an explanation of the table-mode format, see “Table-Mode Format”. For an explanation of the list-mode format, see “List-Mode Format” on page 1-653.

Privileges Required for Changing Object Modes

To change the object mode of a constraint, index, or trigger, you must have the necessary privileges. Specifically, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the table on which the object is defined and must have the Resource privilege on the database.
- You must have the Alter privilege on the table on which the object is defined and the Resource privilege on the database.

Table-Mode Format

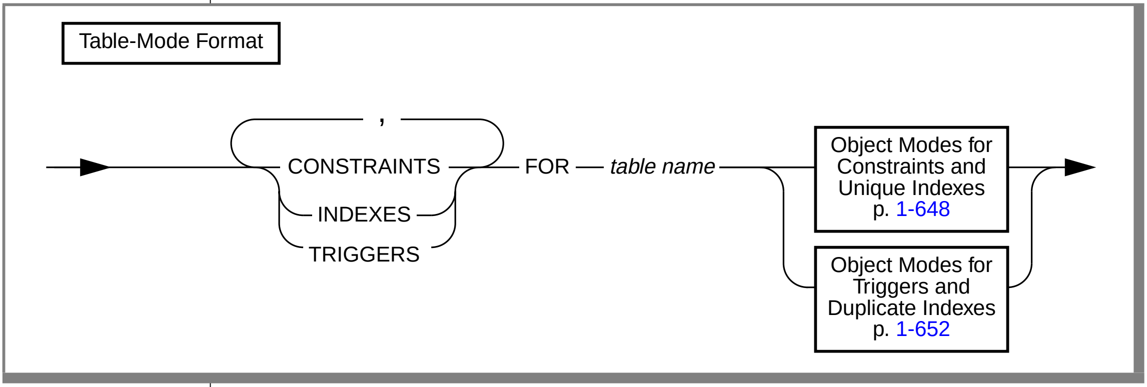


Table-Mode Format
,
CONSTRAINTS
INDEXES
TRIGGERS
FOR
table name
Object Modes for Constraints and Unique Indexes p. 1-648
Object Modes for Triggers and Duplicate Indexes p. 1-652

Element	Purpose	Restrictions	Syntax
table name	The name of the table whose objects will have their object mode changed. There is no default value.	The table must be a local table. You cannot set the object modes of objects defined on a temporary table to the disabled or filtering modes. For information on the privileges required to change the object mode of the objects defined on a table, see "Privileges Required for Changing Object Modes" on page 1-646.	Identifier, p. 1-962

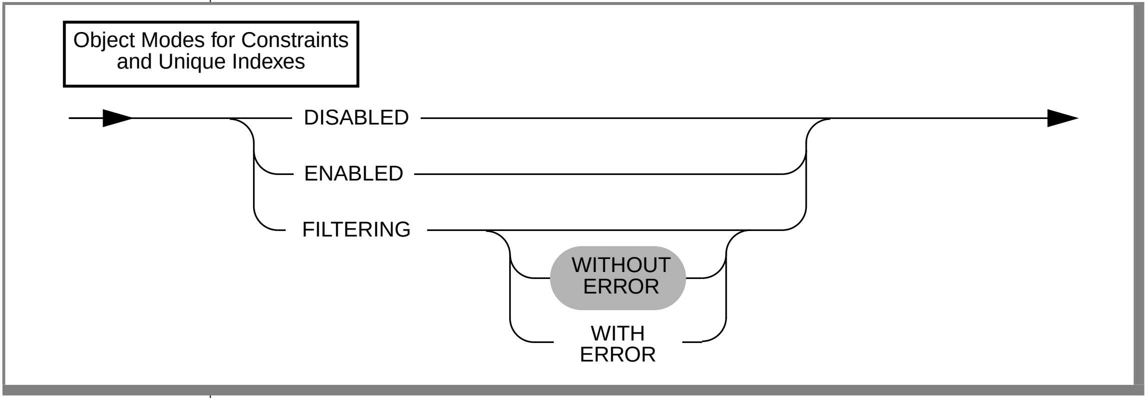
Use the table-mode format to change the object mode of all objects of a given type that have been defined on a particular table. For example, to change the object mode of all constraints that are defined on the **cust_subset** table to the disabled mode, enter the following statement:

```
SET CONSTRAINTS FOR cust_subset DISABLED
```

By using the table-mode format, you can change the object modes of more than one object type with a single SET statement. For example, to change the object mode of all constraints, indexes, and triggers that are defined on the **cust_subset** table to the enabled mode, enter the following statement:

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset
    ENABLED
```

Object Modes for Constraints and Unique Indexes



You can specify the disabled, enabled, or filtering object modes for a constraint or a unique index. You must specify one of these object modes in your SET statement. The SET statement has no default object mode.

You can also specify the object mode for a constraint when you create the constraint with the ALTER TABLE or CREATE TABLE statements. If you do not specify the object mode for a constraint in one of these statements or in a SET statement, the constraint is in the enabled object mode by default.

You can also specify the object mode for a unique index when you create the index with the CREATE INDEX statement. If you do not specify the object mode for a unique index in the CREATE INDEX statement or in a SET statement, the unique index is in the enabled object mode by default.

For definitions of the disabled, enabled, and filtering object modes see "Using Object Modes with Data Manipulation Statements" on page 1-654. For an explanation of the benefits of these object modes, see "Benefits of Object Modes" on page 1-667.

Error Options for Filtering Mode

When you change the object mode of a constraint or unique index to the filtering mode, you can specify the following error options: WITHOUT ERROR or WITH ERROR.

WITHOUT ERROR Option

The WITHOUT ERROR option signifies that when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique-index violation, no integrity-violation error message is returned to the user. The WITHOUT ERROR option is the default error option.

WITH ERROR Option

The WITH ERROR option signifies that when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique-index violation, an integrity-violation error message is returned to the user.

Scope of Error Options

The WITH ERROR and WITHOUT ERROR options apply only when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique index violation. These error options control whether the database server displays an integrity-violation error message after it executes these statements.

These error options do not apply when you attempt to change the object mode of a disabled constraint or disabled unique index to the enabled or filtering mode, and the SET statement fails because one or more rows in the target table violates the constraint or the unique-index requirement. In these cases, if a violations table has been started for the table that contains the inconsistent data, the database server returns an integrity-violation error message regardless of the error option that is specified in the SET statement.

Violations and Diagnostics Tables for Filtering Mode

When you specify the filtering mode for constraints or unique indexes in a SET statement, violations and diagnostics tables are not automatically started for the target table. When you set objects to the filtering mode, be sure to start the violations and diagnostics tables for the target table on which the filtering mode objects are defined. The violations table captures rows that fail to meet integrity requirements. The diagnostics table captures information about each row that fails to meet integrity requirements.

When to Start the Violations and Diagnostics Tables

You are not required to start the violations and diagnostics tables before you set objects to the filtering mode. If you have not started a violations and diagnostics table when you set an object to the filtering mode, the database server executes your SET statement and does not return an error. Similarly, if you issue an INSERT, DELETE, or UPDATE statement on the target table, and you have not started a violations and diagnostics table for the target table, the database server executes the statement and does not return an error as long as all of the integrity requirements on the table are satisfied.

If you have not started a violations and diagnostics table for the target table with filtering-mode objects, the database server does not return an error until an INSERT, DELETE, or UPDATE statement fails to satisfy an integrity requirement on the table. If an INSERT, DELETE, or UPDATE statement fails to satisfy the constraint or unique-index requirement for a particular row, the database server cannot filter the bad row to the violations table because no violations table is associated with the target table. The user receives an error message indicating that no violations table has been started for the target table.

To prevent such errors, start the violations and diagnostics tables for the target table at one of the following points:

- You can start the violations and diagnostics tables before you set any objects that are defined on the table to the filtering mode.
- You can start the violations and diagnostics tables after you set objects to the filtering mode but before any users issue INSERT, DELETE, or UPDATE statements that could violate any integrity requirements on the target table.

How to Start the Violations and Diagnostics Tables

To create the violations and diagnostics tables and associate them with the target table, use the START VIOLATIONS TABLE statement. In this statement, specify the name of the target table for which the violations and diagnostics tables are to be started. You can also assign names to the violations and diagnostics tables in this statement.

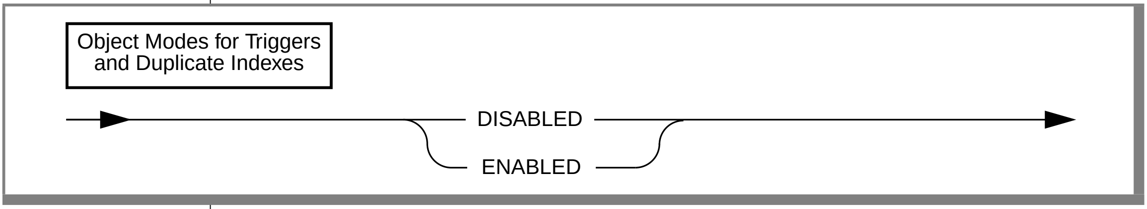
For further information on the START VIOLATIONS TABLE statement and the structure of the violations and diagnostics tables themselves, see the START VIOLATIONS TABLE statement on page 1-744.

How to Stop the Violations and Diagnostics Tables

After you turn off filtering mode for the objects that are defined on a target table, and you no longer need the violations and diagnostics tables, use the STOP VIOLATIONS TABLE statement to drop the association between the target table and the violations and diagnostics tables. In this statement, you specify the name of the target table whose association with the violations and diagnostics tables is to be dropped.

For further information on using the STOP VIOLATIONS TABLE statement, see the STOP VIOLATIONS TABLE statement on page 1-763.

Object Modes for Triggers and Duplicate Indexes



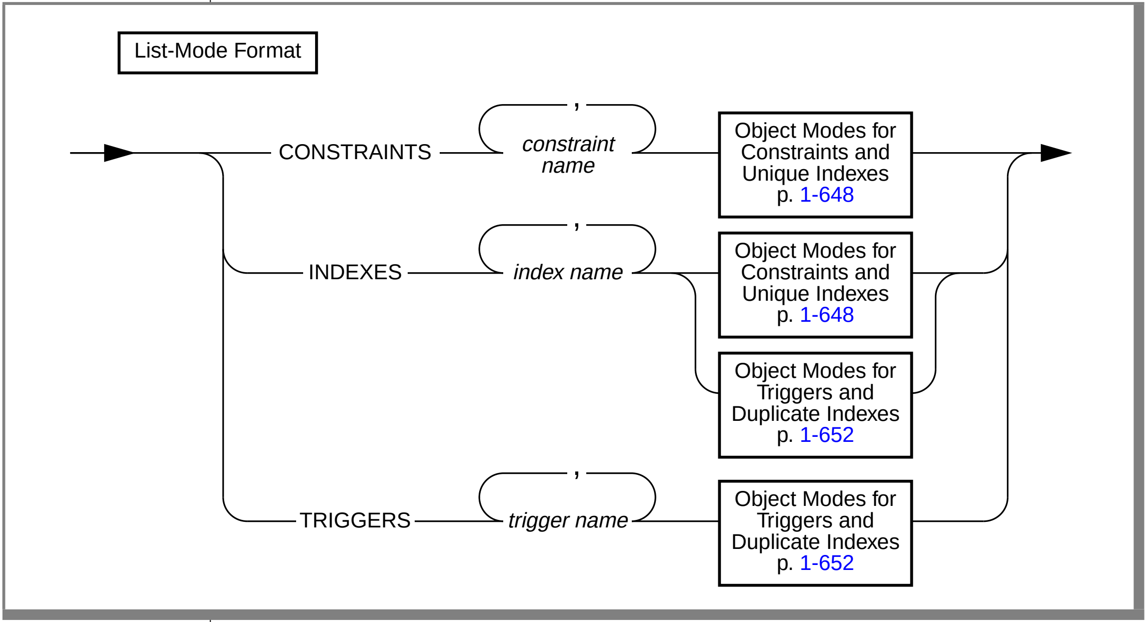
You can specify the disabled or enabled object modes for triggers or duplicate indexes. You must specify one of these object modes in your SET statement. The SET statement has no default object mode.

You can also specify the object mode for a trigger when you create the trigger with the CREATE TRIGGER statement. If you do not specify the object mode for a trigger in the CREATE TRIGGER statement or in a SET statement, the trigger is in the enabled object mode by default.

You can also specify the object mode for a duplicate index when you create the index with the CREATE INDEX statement. If you do not specify the object mode for a duplicate index in the CREATE INDEX statement or in a SET statement, the duplicate index is in the enabled object mode by default.

For definitions of the disabled and enabled object modes, see "Using Object Modes with Data Manipulation Statements" on page 1-654. For an explanation of the benefits of these two object modes, see "Benefits of Object Modes" on page 1-667.

List-Mode Format


List-Mode Format
CONSTRAINTS
,
constraint name
Object Modes for Constraints and Unique Indexes p. 1-648
INDEXES
,
index name
Object Modes for Constraints and Unique Indexes p. 1-648
Object Modes for Triggers and Duplicate Indexes p. 1-652
TRIGGERS
,
trigger name
Object Modes for Triggers and Duplicate Indexes p. 1-652

Element	Purpose	Restrictions	Syntax
constraint name	The name of the constraint whose object mode is to be set, or a list of constraint names. There is no default value.	Each constraint in the list must be a local constraint. All constraints in the list must be defined on the same table.	Identifier, p. 1-962
index name	The name of the index whose object mode is to be set, or a list of index names. There is no default value.	Each index in the list must be a local index. All indexes in the list must be defined on the same table.	Identifier, p. 1-962
trigger name	The name of the trigger whose object mode is to be set, or a list of trigger names. There is no default value.	Each trigger in the list must be a local trigger. All triggers in the list must be defined on the same table.	Identifier, p. 1-962

Use the list-mode format to change the object mode for a particular constraint, index, or trigger. For example, to change the object mode of the unique index **unq_ssn** on the **cust_subset** table to filtering mode, enter the following statement:

```
SET INDEXES unq_ssn FILTERING
```

You can also use the list-mode format to change the object 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 in the enabled mode. To change the object mode of all the triggers except **execute_trig** to the disabled mode, enter the following statement:

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED
```

Using Object Modes with Data Manipulation Statements

You can use object modes to control the effects of INSERT, DELETE, and UPDATE statements. Your choice of object modes affects the tables whose data you are manipulating, the behavior of the objects defined on those tables, and the behavior of the data manipulation statements themselves.

What do we mean by the terms *enabled*, *disabled*, and *filtering*? Definitions of these object modes follow. These definitions explain how each object mode affects tables and data manipulation statements. The definitions focus on the object modes of constraints as an illustration, but the same principles apply to indexes and triggers as well.

Definition of Enabled Mode

Constraints, indexes, and triggers are in the enabled mode by default. When an object is in the enabled mode, the database server recognizes the existence of the object and takes the object into consideration while it executes data manipulation statements. For example, when a constraint is enabled, any INSERT, UPDATE, or DELETE statement that violates the constraint fails, and the target row remains unchanged. In addition, the user receives an error message.

Definition of Disabled Mode

When an object is in the disabled mode, the database server acts as if the object did not exist and does not take it into consideration during the execution of data manipulation statements. For example, when a constraint is disabled, any INSERT, UPDATE, or DELETE statement that violates the constraint succeeds, and the target row is changed. The user does not receive an error message.

Definition of Filtering Mode

When an object is in the filtering mode, the object behaves the same as in the enabled mode in that the database server recognizes the existence of the object during INSERT, UPDATE, and DELETE statements. For example, when a constraint is in the filtering mode, and an INSERT, DELETE, or UPDATE statement is executed, any target rows that violate the constraint remain unchanged.

However, the database server handles data manipulation statements differently for objects in enabled and filtering mode, as the following paragraphs describe:

- If a constraint or unique index is in the enabled mode, the database server carries out the INSERT, UPDATE, or DELETE statement only if all the target rows affected by the statement satisfy the constraint or the unique index requirement. The database server updates all the target rows in the table.
- If a constraint or unique index is in the filtering mode, the database server carries out the INSERT, UPDATE, or DELETE statement even if one or more of the target rows fail to satisfy the constraint or the unique index requirement. The database server updates the good rows in the table (the target rows that satisfy the constraint or unique index requirement). The database server does not update the bad rows in the table (that is, the target rows that fail to satisfy the constraint or unique index requirement). Instead the database server sends each bad row to a special table called the violations table. The database server places information about the nature of the violation for each bad row in another special table called the diagnostics table.

Example of Object Modes with Data Manipulation Statements

An example with the INSERT statement can illustrate the differences between the enabled, disabled, and filtering modes. Consider an INSERT statement in which a user tries to add a row that does not satisfy an integrity constraint on a table. For example, assume that a user **joe** has created a table named **cust_subset**, and this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives). The **ssn** column has the INT data type. The other three columns have the CHAR data type.

Assume that user **joe** has defined the **lname** column as not null but has not assigned a name to the not null constraint, so the database server has implicitly assigned the name **n104_7** to this constraint. Finally, assume that user **joe** has created a unique index named **unq_ssn** on the **ssn** column.

Now a user **linda** who has the Insert privilege on the **cust_subset** table enters the following INSERT statement on this table:

```
INSERT INTO cust_subset (ssn, fname, city)
    VALUES (973824499, "jane", "los altos")
```

User **linda** has entered values for all the columns of the new row except for the **lname** column, even though the **lname** column has been defined as a not null column. The database server behaves in the following ways, depending on the object mode of the constraint:

- If the constraint is disabled, the row is inserted in the target table, and no error is returned to the user.
- If the constraint is enabled, the row is not inserted in the target table. A constraint-violation error is returned to the user, and the effects of the statement are rolled back (if the database is a Universal Server database with logging).
- If the constraint is filtering, the row is not inserted in the target table. Instead the row is inserted in the violations table. Information about the integrity violation caused by the row is placed in the diagnostics table. The effects of the INSERT statement are not rolled back. You receive an error message if you specified the WITH ERROR option for the filtering-mode constraint. By analyzing the contents of the violations and the diagnostics tables, you can identify the reason for the failure and either take corrective action or roll back the operation.

We can better grasp the distinctions among disabled, enabled, and filtering modes by viewing the actual results of the INSERT statement shown in the preceding example.

Results of the Insert Operation When the Constraint Is Disabled

If the not-null constraint on the **cust_subset** table is disabled, the INSERT statement that user **linda** issues successfully inserts the new row in this table. The new row of the **cust_subset** table has the following column values.

ssn	fname	lname	city
973824499	jane	NULL	los altos

Results of the Insert Operation When the Constraint Is Enabled

If the not-null constraint on the **cust_subset** table is enabled, the INSERT statement fails to insert the new row in this table. Instead user **linda** receives the following error message when she enters the INSERT statement:

```
-292 An implied insert column lname does not accept NULLs.
```

Results of the Insert When Constraint Is in Filtering Mode

If the not-null constraint on the **cust_subset** table is set to the filtering mode, the INSERT statement that user **linda** issues fails to insert the new row in this table. Instead the new row is inserted into the violations table, and a diagnostic row that describes the integrity violation is added to the diagnostics table.

Assume that user **joe** has started a violations and diagnostics table for the **cust_subset** table. The violations table is named **cust_subset_vio**, and the diagnostics table is named **cust_subset_dia**. The new row added to the **cust_subset_vio** violations table when user **linda** issues the INSERT statement on the **cust_subset** target table has the following column values.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_recowner
973824499	jane	NULL	los altos	1	I	linda

This new row in the **cust_subset_vio** violations table has the following characteristics:

- The first four columns of the violations table exactly match the columns of the target table. These four columns have the same names and the same data types as the corresponding columns of the target table, and they have the column values that were supplied by the INSERT statement that user **linda** entered.
- The value `1` in the **informix_tupleid** column is a unique serial identifier that is assigned to the nonconforming row.
- The value `I` in the **informix_optype** column is a code that identifies the type of operation that has caused this nonconforming row to be created. Specifically, `I` stands for an insert operation.
- The value `linda` in the **informix_recowner** column identifies the user who issued the statement that caused this nonconforming row to be created.

The INSERT statement that user **linda** issued on the **cust_subset** target table also causes a diagnostic row to be added to the **cust_subset_dia** diagnostics table. The new diagnostic row added to the diagnostics table has the following column values.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7

This new diagnostic row in the **cust_subset_dia** diagnostics table has the following characteristics:

- This row of the diagnostics table is linked to the corresponding row of the violations table by means of the **informix_tupleid** column that appears in both tables. The value 1 appears in this column in both tables.
- The value C in the **objtype** column identifies the type of integrity violation that the corresponding row in the violations table caused. Specifically, the value C stands for a constraint violation.
- The value joe in the **objowner** column identifies the owner of the constraint for which an integrity violation was detected.
- The value n104_7 in the **objname** column gives the name of the constraint for which an integrity violation was detected.

By joining the violations and diagnostics tables, user **joe** (who owns the **cust_subset** target table and its associated special tables) or the DBA can find out that the row in the violations table whose **informix_tupleid** value is 1 was created after an INSERT statement and that this row is violating a constraint. The table owner or DBA can query the **sysconstraints** system catalog table to determine that this constraint is a not null constraint. Now that the reason for the failure of the INSERT statement is known, user **joe** or the DBA can take corrective action.

Multiple Diagnostic Rows for One Violations Row

In the preceding example, only one row in the diagnostics table corresponds to the new row in the violations table. However, more than one diagnostic row can be added to the diagnostics table when a single new row is added to the violations table. For example, if the **ssn** value (973824499) that user **linda** entered in the INSERT statement had been the same as an existing value in the **ssn** column of the **cust_subset** target table, only one new row would appear in the violations table, but the following two diagnostic rows would be present in the **cust_subset_dia** diagnostics table.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7
1	I	joe	unq_ssn

Both rows in the diagnostics table correspond to the same row of the violations table because both of these rows have the value 1 in the **informix_tupleid** column. However, the first diagnostic row identifies the constraint violation caused by the INSERT statement that user **linda** issued, while the second diagnostic row identifies the unique-index violation caused by the same INSERT statement. In this second diagnostic row, the value I in the **objtype** column stands for a unique-index violation, and the value unq_ssn in the **objname** column gives the name of the index for which the integrity violation was detected.

For information on when and how to start violations and diagnostics tables for a target table, see "Violations and Diagnostics Tables for Filtering Mode" on page 1-650. For further information on the structure of the violations and diagnostics tables, see the START VIOLATIONS TABLE statement on page 1-744.

Using Object Modes to Achieve Data Integrity

In addition to using object modes with data manipulation statements, you can also use object modes when you add a new constraint or new unique index to a target table. By selecting the correct object mode, you can add the constraint or index to the target table easily even if existing rows in the target table violate the new integrity specification.

You can add a new constraint or index easily by taking the following steps. If you follow this procedure, you do not have to examine the entire target table to identify rows that fail to satisfy the constraint or unique-index requirement:

- Add the constraint or index in the enabled mode. If all existing rows in the table satisfy the constraint or unique-index requirement, your ALTER TABLE or CREATE INDEX statement executes successfully, and you do not need to take any further steps. However, if any existing rows in the table fail to satisfy the constraint or unique-index requirement, your ALTER TABLE or CREATE INDEX statement returns an error message, and you need to take the following steps.
- Add the constraint or index in the disabled mode. Issue the ALTER TABLE statement again, and specify the DISABLED keyword in the ADD CONSTRAINT or MODIFY clause; or issue the CREATE INDEX statement again, and specify the DISABLED keyword.
- Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
- Issue a SET statement to switch the object mode of the constraint or index to the enabled mode. When you issue this statement, the statement fails, and existing rows in the target table that violate the constraint or the unique-index requirement are duplicated in the violations table. The constraint or index remains disabled, and you receive an integrity-violation error message.
- Issue a SELECT statement on the violations table to retrieve the nonconforming rows that were duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
- Take corrective action on the rows in the target table that violate the constraint.
- After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled constraint or index to the enabled mode. This time the constraint or index is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint or unique-index requirement.

Example of Using Object Modes to Achieve Data Integrity

The following example shows how to use object modes to add a constraint and unique index to a target table easily. Assume that a user **joe** has created a table named **cust_subset**, and this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

Also assume that no constraints or unique indexes are defined on the **cust_subset** table and that the **fname** column is the primary key. In addition, assume that no violations and diagnostics tables currently exist for this target table. Finally, assume that this table currently contains four rows with the following column values.

ssn	fname	lname	city
111763227	mark	jackson	sunnyvale
222781244	rhonda	NULL	palo alto
111763227	steve	NULL	san mateo
333992276	tammy	jones	san jose

Adding the Objects in the Enabled Mode

User **joe**, the owner of the **cust_subset** table, enters the following statements to add a unique index on the **ssn** column and a not null constraint on the **lname** column:

```
CREATE UNIQUE INDEX unq_ssn ON cust_subset (ssn) ENABLED;
ALTER TABLE cust_subset MODIFY (lname CHAR(15)
    NOT NULL CONSTRAINT lname_notblank ENABLED);
```

Both of these statements fail because existing rows in the **cust_subset** table violate the integrity specifications. The row whose **fname** value is `rhonda` violates the not null constraint on the **lname** column. The row whose **fname** value is `steve` violates both the not null constraint on the **lname** column and the unique-index requirement on the **ssn** column.

Adding the Objects in the Disabled Mode

To recover from the preceding errors, user **joe** reenters the CREATE INDEX and ALTER TABLE statements and specifies the disabled mode in both statements, as follows:

```
CREATE UNIQUE INDEX unq_ssn ON cust_subset (ssn) DISABLED;
ALTER TABLE cust_subset MODIFY (lname CHAR(15)
    NOT NULL CONSTRAINT lname_notblank DISABLED);
```

Both of these statements execute successfully because the database server does not enforce unique-index requirements or constraint specifications when these objects are disabled.

Starting a Violations and Diagnostics Table

Now that the new constraint and index are added for the **cust_subset** table, user **joe** takes steps to find out which existing rows in the **cust_subset** table violate the constraint and the index.

First, user **joe** enters the following statement to start a violations and diagnostics table for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
```

Because user **joe** has not assigned names to the violations and diagnostics tables in this statement, the tables are named **cust_subset_vio** and **cust_subset_dia** by default.

Using the SET Statement to Capture Violations

Now that violations and diagnostics tables exist for the target table, user **joe** issues the following SET statement to switch the mode of the new index and constraint from the disabled mode to the enabled mode:

```
SET CONSTRAINTS, INDEXES FOR cust_subset ENABLED
```

The result of this SET statement is that the existing rows in the **cust_subset** table that violate the constraint and the unique-index requirement are copied to the **cust_subset_vio** violations table, and diagnostic information about the nonconforming rows is added to the **cust_subset_dia** diagnostics table. The SET statement fails, and the constraint and index remain disabled.

The following table shows the contents of the **cust_subset_vio** violations table after user **joe** issues the SET statement.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_recowner
222781244	rhonda	NULL	palo alto	1	S	joe
111763227	steve	NULL	san mateo	2	S	joe

These two rows in the **cust_subset_vio** violations table have the following characteristics:

- The row in the **cust_subset** target table whose **fname** value is `rhonda` is duplicated to the **cust_subset_vio** violations table because this row violates the not null constraint on the **lname** column.
- The row in the **cust_subset** target table whose **fname** value is `steve` is duplicated to the **cust_subset_vio** violations table because this row violates the not null constraint on the **lname** column and the unique-index requirement on the **ssn** column.
- The value `1` in the **informix_tupleid** column for the first row and the value `2` in the **informix_tupleid** column for the second row are unique serial identifiers assigned to the nonconforming rows.
- The value `S` in the **informix_optype** column for each row is a code that identifies the type of operation that has caused this nonconforming row to be placed in the violations table. Specifically, the `S` stands for a SET statement.
- The value `joe` in the **informix_recowner** column for each row identifies the user who issued the statement that caused this nonconforming row to be placed in the violations table.

The following table shows contents of the **cust_subset_dia** diagnostics table after user **joe** issues the SET statement.

informix_tupleid	objtype	objowner	objname
1	C	joe	lname_notblank
2	C	joe	lname_notblank
2	I	joe	unq_ssn

These three rows in the **cust_subset_dia** diagnostics table have the following characteristics:

- Each row in the diagnostics table and the corresponding row in the violations table are joined by the **informix_tupleid** column that appears in both tables.
- The first row in the diagnostics table has an **informix_tupleid** value of `1`. It is joined to the row in the violations table whose **informix_tupleid** value is `1`. The value `C` in the **objtype** column for this diagnostic row identifies the type of integrity violation that was caused by the corresponding row in the violations table. Specifically, the value `C` stands for a constraint violation. The value `lname_notblank` in the **objname** column for this diagnostic row gives the name of the constraint for which an integrity violation was detected.
- The second row in the diagnostics table has an **informix_tupleid** value of `2`. It is joined to the row in the violations table whose **informix_tupleid** value is `2`. The value `C` in the **objtype** column for this second diagnostic row indicates that a constraint violation was caused by the corresponding row in the violations table. The value `lname_notblank` in the **objname** column for this diagnostic row gives the name of the constraint for which an integrity violation was detected.

- The third row in the diagnostics table has an **informix_tupleid** value of 2. It is also joined to the row in the violations table whose **informix_tupleid** value is 2. The value I in the **objtype** column for this third diagnostic row indicates that a unique-index violation was caused by the corresponding row in the violations table. The value unq_ssn in the **objname** column for this diagnostic row gives the name of the index for which an integrity violation was detected.
- The value joe in the **objowner** column of all three diagnostic rows identifies the owner of the object for which an integrity violation was detected. The name of user **joe** appears in all three rows because he created the constraint and index on the **cust_subset** table.

Identifying Nonconforming Rows to Obtain Information

To determine the contents of the violations table, user **joe** enters a SELECT statement to retrieve all rows from the table. Then, to obtain complete diagnostic information about the nonconforming rows, user **joe** joins the violations and diagnostics tables by means of another SELECT statement. User **joe** can perform these operations either interactively or through a program.

Taking Corrective Action on the Nonconforming Rows

After the user **joe** identifies the nonconforming rows in the **cust_subset** table, he can correct them. For example, he can enter UPDATE statements on the **cust_subset** table either interactively or through a program.

Enabling the Disabled Objects

Once all the nonconforming rows in the **cust_subset** table are corrected, user **joe** issues the following SET statement to set the new constraint and index to the enabled mode:

```
SET CONSTRAINTS, INDEXES FOR cust_subset ENABLED
```

This time the SET statement executes successfully. The new constraint and new unique index are enabled, and no error message is returned to user **joe** because all rows in the **cust_subset** table now satisfy the new constraint specification and unique-index requirement.

Benefits of Object Modes

The preceding examples show how object modes work when users execute data manipulation statements on target tables or add new constraints and indexes to target tables. The preceding examples suggest some of the benefits of the different object modes. The following sections state these benefits explicitly.

Benefits of Disabled Mode

The benefits of the disabled mode are as follows:

- You can use the disabled mode to insert many rows quickly into a target table. Especially during load operations, updates of the existing indexes and enforcement of referential constraints make up a big part of the total cost of the operation. By disabling the indexes and referential constraints during the load operation, you improve the performance and efficiency of the load.
- To add a new constraint or new unique index to an existing table, you can add the object even if some rows in the table do not satisfy the new integrity specification. If the constraint or index is added to the table in disabled mode, your ALTER TABLE or CREATE INDEX statement does not fail no matter how many existing rows violate the new integrity requirement.

  If a violations table has been started, a SET statement that switches the disabled objects to the enabled or filtering mode fails, but it causes the nonconforming rows in the target table to be duplicated in the violations table so that you can identify the rows and take corrective action. After you fix the nonconforming rows in the target table, you can reissue the SET statement to switch the disabled objects to the enabled or filtering mode.

Benefits of Enabled Mode

The enabled mode is the default object mode for all database objects. We can summarize the benefits of this mode for each type of database object as follows:

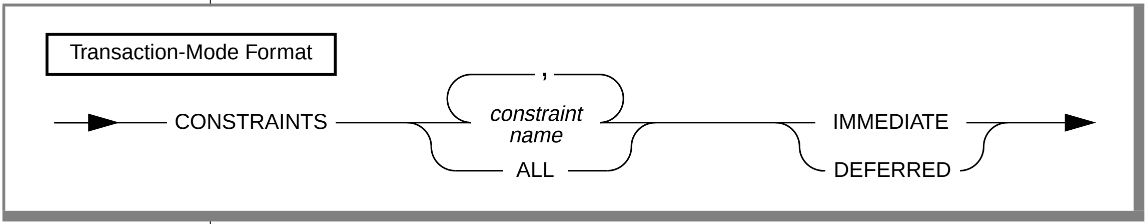
- The benefit of enabled mode for constraints is that the database server enforces the constraint and thus ensures the consistency of the data in the database.
- The benefit of enabled mode for indexes is that the database server updates the index after insert, delete, and update operations. Thus the index is up to date and is used by the optimizer during database queries.
- The benefit of enabled mode for triggers is that the trigger event always sets the triggered action in motion. Thus the purpose of the trigger is always realized during actual data-manipulation operations.

Benefits of Filtering Mode

The benefits of setting a constraint or unique index to the filtering mode are as follows:

- During load operations, inserts that violate a filtering mode constraint or unique index do not cause the load operation to fail. Instead, the database server filters the bad rows to the violations table and continues the load operation.
- When an INSERT, DELETE, or UPDATE statement that affects multiple rows causes a filtering mode constraint or unique index to be violated for a particular row or rows, the statement does not fail. Instead, the database server filters the bad row or rows to the violations table and continues to execute the statement.
- When any INSERT, DELETE, or UPDATE statement violates a filtering mode constraint or unique index, the user can identify the failed row or rows and take corrective action. The violations and diagnostics tables capture the necessary information, and users can take corrective action after they analyze this information.

Transaction-Mode Format



Element	Purpose	Restrictions	Syntax
constraint name	The name of the constraint whose transaction mode is to be changed, or a list of constraint names. There is no default value.	The specified constraint must exist in a database with logging. You cannot change the transaction mode of a constraint to deferred mode unless the constraint is currently in the enabled mode. All constraints in a list of constraints must exist in the same database.	Identifier, p. 1-962

You can use the transaction-mode format of the SET statement to set the transaction mode of constraints.

You use the IMMEDIATE keyword to set the transaction mode of constraints to statement-level checking. You use the DEFERRED keyword to set the transaction mode to transaction-level checking.

You can set the transaction mode of constraints only in a database with logging.

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. Immediate is the default transaction mode of constraints.

Transaction-Level Checking

When you set the transaction mode of constraints to deferred, statement-level checking is turned off, and all 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, the checking of 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 statement specifies is the transaction in which the SET statement is executed. You cannot execute this form of the SET statement outside a transaction. Once a COMMIT WORK or ROLLBACK WORK statement is successfully completed, the transaction mode of all constraints reverts to IMMEDIATE.

Switching Transaction Modes

To switch from transaction-level checking to statement-level checking, you can use the SET statement to set the transaction mode to immediate, or you can use a COMMIT WORK or ROLLBACK WORK statement in your transaction.

Specifying All Constraints or a List of Constraints

You can specify all constraints in the database in your SET statement, or you can specify a single constraint or list of constraints.

Specifying All Constraints

If you specify the ALL keyword, the SET 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 statement.

Specifying a List of Constraints

If you specify a single constraint name or a list of constraints, the SET 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 statement, that constraint is checked at the statement level regardless of the setting that you specified in the SET 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 transaction-mode form of the SET 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 CONSTRAINTS 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
```

References

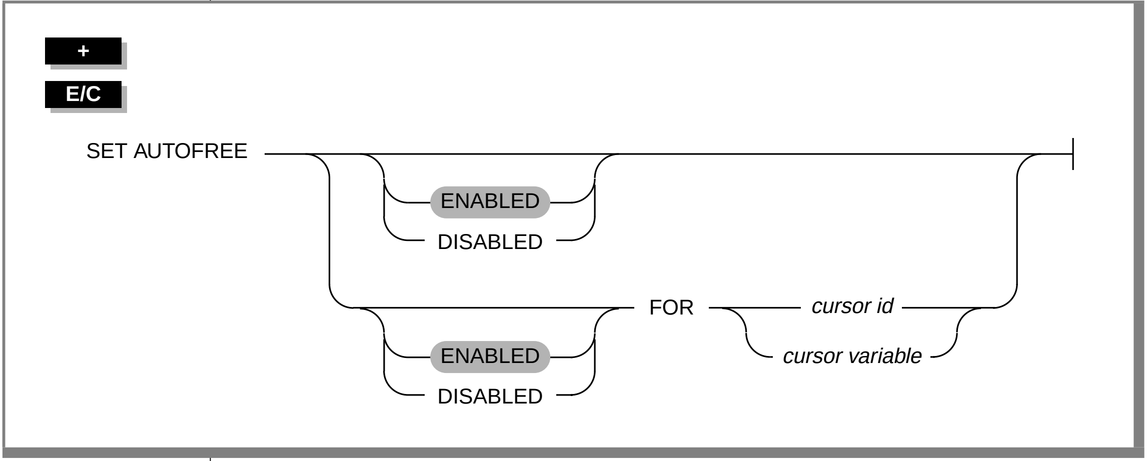
See the START VIOLATIONS TABLE and STOP VIOLATIONS TABLE statements in this manual.

For information on the system catalog tables associated with the SET statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

SET AUTOFREE

The SET AUTOFREE statement enables the AUTOFREE feature for cursors in an INFORMIX-ESQL/C application.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	The name of a cursor for which the AUTOFREE feature is enabled or disabled	The cursor must be declared within the program.	Identifier, p. 1-962
cursor variable	A host variable that holds the value of *cursor id*	The host variable must store the name of a cursor that is declared within the program.	Variable name must conform to language-specific rules for variable names.

Usage

The *Auto*matic-*FREE* feature (AUTOFREE) is one of the ESQL/C optimization features that can minimize network traffic when an ESQL/C application fetches rows from a database server. When the AUTOFREE feature is enabled, ESQL/C saves a round trip of message requests because it does not need to send the FREE statement to the database server for execution. Instead, the database server automatically frees a cursor when it closes this cursor. If this cursor has an associated prepared statement, the database server also frees the prepared statement.

The SET AUTOFREE statement allows an ESQL/C application to:

- enable the AUTOFREE feature.

  Use the ENABLED option of the SET AUTOFREE statement.
- disable the AUTOFREE feature.

  Use the DISABLED option of the SET AUTOFREE statement.

If you do not specify either option, the default is ENABLED. The following SET AUTOFREE statement enables the AUTOFREE feature for all cursors in the application:

```
EXEC SQL set autofree;
```

ENABLED Option

The ENABLED option of the SET AUTOFREE statement enables the AUTOFREE feature within the ESQL/C application. You can use the SET AUTOFREE statement in two modes:

- Global-AUTOFREE mode

  This mode affects *all* cursors that are declared or opened after this SET AUTOFREE statement executes. The upper part of the syntax diagram represents the global-AUTOFREE mode.
- Cursor-AUTOFREE mode

  This mode affects a particular cursor that is prepared or opened after this SET AUTOFREE statement executes. The lower part of the syntax diagram represents the cursor-AUTOFREE mode.

When you execute the SET AUTOFREE statement in either of these modes, the AUTOFREE feature only takes affect on a cursor if that cursor is declared or opened *after* this SET AUTOFREE statement executes.

Important: *You can also set the **IFX_AUTOFREE** environment variable to one (1) to enable the AUTOFREE feature. For more information on the **IFX_AUTOFREE** environment variable, see the "Using the IFX_AUTOFREE Environment Variable" on page 1-680.*

Once you enable the AUTOFREE feature on a cursor, you cannot open the cursor a second time; the database server automatically frees the cursor when it closes it the first time. For more information, see "Implicit Closing of Cursors" on page 1-681.

Using Global-AUTOFREE Mode

In global-AUTOFREE mode, the ENABLED option of SET AUTOFREE statement enables the AUTOFREE feature for *all* cursors that are subsequently declared or opened in the program. After the database server closes a cursor, it automatically frees this cursor only if the cursor has been declared or opened *after* this SET AUTOFREE statement executes.

The following SET AUTOFREE statement enables the AUTOFREE feature for all cursors that are subsequently declared or opened in the application:

```
EXEC SQL set autofree enabled;
```

If you omit the ENABLED or DISABLED option in the SET AUTOFREE statement, the AUTOFREE feature is enabled for all subsequent cursors by default. The following SET AUTOFREE statement also enables the AUTOFREE feature for all subsequently declared or opened cursors:

```
EXEC SQL set autofree;
```

The following code fragment shows how the ENABLED option of the SET AUTOFREE statement automatic frees memory for all subsequent cursors:

```
/* Declare curs1 cursor for stmt1 prepared statement */
EXEC SQL prepare stmt1 from 'select a from tab_x';
EXEC SQL declare curs1 cursor for stmt1;

/* Open curs1 cursor and fetch the contents. */
EXEC SQL open curs1;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs1 into :a;
    printf("a=%d\n", a);
    }

/* Declare curs2 cursor for stmt2 prepared statement */
EXEC SQL prepare stmt2 from 'select * from tab_x';
EXEC SQL declare curs2 cursor for stmt2;

/* Enable autofree feature for all subsequent cursors. */
EXEC SQL set autofree enabled;

/* Open curs2 cursor and fetch the contents. */
EXEC SQL open curs2;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs2 into :a, :b, :c, :d;
    printf("a=%d b=%d c=%d d=%d\n", a, b, c, d);
    }

/* Declare curs3 cursor for stmt3 prepared statement */
EXEC SQL prepare stmt3 from 'select a, b from tab_x';
EXEC SQL declare curs3 cursor for stmt3;

/* Open curs3 cursor and fetch the contents. */
EXEC SQL open curs3;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs3 into :a, :b;
    printf("a=%d b=%d\n", a, b);
    }
```

In the preceding code fragment, the SET AUTOFREE statement enables the AUTOFREE feature for the following cursors:

- For the **curs2** cursor because **curs2** is *opened* after SET AUTOFREE executes
- For the **curs3** cursor because **curs3** is *declared* and opened after SET AUTOFREE executes

However, this SET AUTOFREE statement does *not* enable the AUTOFREE feature for the **curs1** cursor because this cursor is neither declared nor opened after the SET AUTOFREE statement executes.

Using Cursor-AUTOFREE Mode

In cursor-AUTOFREE mode, the ENABLED option of the SET AUTOFREE statement enables the AUTOFREE feature for *only* the cursor that you specify after the FOR keyword. After the database server closes the specified cursor, it automatically frees this cursor only if the cursor has been declared or opened *after* this SET AUTOFREE statement executes. You can specify the cursor by its cursor identifier or by a host variable that contains the cursor identifier.

The following SET AUTOFREE statement enables the AUTOFREE feature for the x1 cursor if it is subsequently declared or opened in the application:

```
EXEC SQL set autofree enabled for x1;
```

If you omit the ENABLED or DISABLED option in the SET AUTOFREE statement, the AUTOFREE feature is enabled for the specified cursor by default. The following SET AUTOFREE statement also enables the AUTOFREE feature for the **x1** cursor:

```
EXEC SQL set autofree for x1;
```

In the following code fragment, the SET AUTOFREE statement enables the AUTOFREE feature for the cursor named **curs3**:

```
/* Declare curs1 cursor for stmt1 prepared statement */
EXEC SQL prepare stmt1 from 'select a from tab_x';
EXEC SQL declare curs1 cursor for stmt1;

/* Open curs1 cursor and fetch the contents. */
EXEC SQL open curs1;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs1 into :a;
    printf("a=%d\n", a);
    }

/* Declare curs2 cursor for stmt2 prepared statement */
EXEC SQL prepare stmt2 from 'select * from tab_x';
EXEC SQL declare curs2 cursor for stmt2;

/* Enable autofree feature for the curs3 cursor. */
EXEC SQL set autofree enabled for curs3;

/* Open curs2 cursor and fetch the contents. */
EXEC SQL open curs2;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs2 into :a, :b, :c, :d;
    printf("a=%d b=%d c=%d d=%d\n", a, b, c, d);
    }

/* Declare curs3 cursor for stmt3 prepared statement */
EXEC SQL prepare stmt3 from 'select a, b from tab_x';
EXEC SQL declare curs3 cursor for stmt3;

/* Open curs3 cursor and fetch the contents. */
EXEC SQL open curs3;
while (sqlca.sqlcode == 0)
    {
    EXEC SQL fetch curs3 into :a, :b;
    printf("a=%d b=%d\n", a, b);
    }
```

In the preceding code fragment, the SET AUTOFREE statement enables the AUTOFREE feature *only* for the **curs3** cursor. Even though the **curs2** cursor is opened after this SET AUTOFREE executes, this cursor is not AUTOFREE-enabled because the SET AUTOFREE statement has specified only the **curs3** cursor.

DISABLED Option

The DISABLED option of the SET AUTOFREE statement disables the AUTOFREE feature within the ESQL/C application. This option works with both modes of the SET AUTOFREE statement:

- Global-AUTOFREE mode

  This mode affects *all* cursors that are declared or opened after this SET AUTOFREE statement executes. The upper part of the syntax diagram represents the global-AUTOFREE mode.

- Cursor-AUTOFREE mode

  This mode affects a particular cursor that is prepared or opened after this SET AUTOFREE statement executes. The lower part of the syntax diagram represents the cursor-AUTOFREE mode.

Important: *You can also set the* ***IFX_AUTOFREE*** *environment variable to zero (*0*) to disable the AUTOFREE feature. For more information on the* ***IFX_AUTOFREE*** *environment variable, see the "Using the IFX_AUTOFREE Environment Variable" on page 1-680.*

Using Global-AUTOFREE Mode

In global-AUTOFREE mode, the DISABLED option of SET AUTOFREE statement disables the AUTOFREE feature for *all* cursors that are subsequently declared or opened in the program. However, the SET AUTOFREE DISABLED statement does not disable the AUTOFREE feature for any cursor that has already been opened.

The following example shows how to use the DISABLED option to disable automatic freeing of memory for *all* subsequent cursors:

```
EXEC SQL set autofree disabled;
```

Using Cursor-AUTOFREE Mode

In cursor-AUTOFREE mode, the DISABLED option of the SET AUTOFREE statement disables the AUTOFREE feature for *only* the cursor that you specify after the FOR keyword. You can specify the cursor by its cursor identifier or by a host variable that contains the cursor identifier.

When you specify the DISABLED option for a specific cursor, the database server automatically frees that cursor only. The following SET AUTOFREE statement disables the AUTOFREE feature for a cursor named **x1**:

```
EXEC SQL set autofree disabled for x1;
```

One advantage of cursor-AUTOFREE mode is that you can use it to override a global setting for all cursors. For example, if you issue a SET AUTOFREE ENABLED statement to enable the AUTOFREE feature for all cursors in a program, you can issue a subsequent SET AUTOFREE DISABLED FOR statement to disable the AUTOFREE feature for a particular 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 particular cursor named **x1**:

```
EXEC SQL set autofree enabled;
EXEC SQL set autofree disabled for x1;
```

Using the IFX_AUTOFREE Environment Variable

You can also enable or disable the AUTOFREE feature with the **IFX_AUTOFREE** environment variable, as follows:

- Set this environment variable to 1 to *enable* the AUTOFREE feature for every cursor in every thread of the application.
- Set this environment variable to 0 to *disable* the AUTOFREE feature for every cursor in every thread of the application.

If you do not set the **IFX_AUTOFREE** environment variable, the AUTOFREE feature is disabled. However, in each thread, a SET AUTOFREE statement overrides the value of the **IFX_AUTOFREE** environment variable.

The **IFX_AUTOFREE** environment variable works only with client applications such as those written in INFORMIX-ESQL/C. This environment variable has no effect on Informix database utilities such as DB-Access, **dbload**, **dbimport**, **dbexport**, and **dbschema**.

For more information on the **IFX_AUTOFREE** environment variable, see the *Informix Guide to SQL: Reference*.

Implicit Closing of Cursors

If you do not close the cursor explicitly, and then you open it again, the database server implicitly closes the cursor before it can reopen it. If the cursor has the AUTOFREE feature enabled, this implicit close of the cursor triggers the AUTOFREE feature. The second open of the cursor generates an error message (cursor not found) because the database server has already freed the cursor.

References

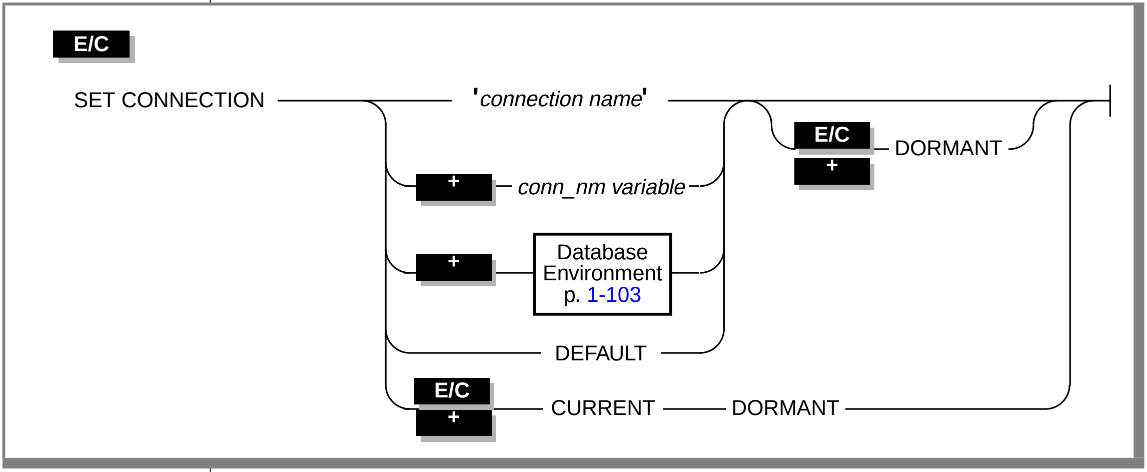
See the DECLARE, OPEN, FETCH, CLOSE, FREE, and PREPARE statements in this manual. For another ESQL/C optimization, see the SET DEFERRED_PREPARE statement.

In the *INFORMIX-ESQL/C Programmer's Manual*, see the chapter on how to use dynamic SQL for information on the AUTOFREE feature.

SET CONNECTION

The SET CONNECTION statement reestablishes a connection between an application and a database environment and makes the connection current. You can also use the SET CONNECTION statement with the DORMANT option to put the current connection in a dormant state.

Syntax



Element	Purpose	Restrictions	Syntax
connection name	Quoted string that identifies the *connection name* that you assigned to a specific connection. It is the *connection name* assigned by the CONNECT statement when the initial connection was made.	The database must already exist. If you use the SET CONNECTION statement with the DORMANT option, *connection name* must represent the current connection. If you use the SET CONNECTION statement without the DORMANT option, *connection name* must represent a dormant connection.	Quoted String, p. 1-1010
conn_nm variable	Host variable that contains the value of *connection name*	Variable must be the character data type.	Variable name must conform to language-specific rules for variable names.

Usage

You can use the SET CONNECTION statement to change the state of a connection in the following ways:

- Make a dormant connection current

  For information on using SET CONNECTION to make a dormant connection current, see “Making a Dormant Connection the Current Connection”.

- Make the current connection dormant

  For information on using SET CONNECTION to make the current connection dormant, see “Making a Current Connection Dormant” on page 1-684.

Making a Dormant Connection the Current Connection

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. A dormant connection is a connection that has been established but is not current.

The SET CONNECTION statement in the following example makes connection `con1` the current connection and makes `con2` a dormant connection:

```
CONNECT TO 'stores7' AS 'con1'
...
CONNECT TO 'demo7' 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 page 1-100.) Reestablishing a connection is comparable to establishing the initial connection, except that it typically avoids authenticating the user’s permissions 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 Dormant

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
```

However, the following SET CONNECTION statement executes successfully:

```
SET CONNECTION 'con1' DORMANT
```

Dormant Connections in a Single-Threaded Environment

In a single-threaded application (an ESQL/C application that does not use threads), the DORMANT option makes the current connection dormant. The availability of the DORMANT option in single-threaded applications makes single-threaded ESQL/C applications upwardly compatible with thread-safe ESQL/C applications.

Dormant Connections in a Thread-Safe ESQL/C Environment

E/C

As in a single-threaded application, a thread-safe ESQL/C application (an ESQL/C application that uses threads) can establish many connections to one or more databases. However, in the single-threaded environment, only one connection can be active while the program executes. In the thread-safe environment, there can be 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.

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.

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 is initiated with the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, an ongoing transaction can used by any thread that subsequently connects to that database environment. 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.

For a detailed discussion of thread-safe ESQL/C applications and the use of the SET CONNECTION statement in these applications, see the *INFORMIX-ESQL/C Programmer's Manual*. ♦

Identifying the Connection

If the application did not use *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 *connection name*. For information about quoted strings that contain a database environment, see "Database Environment" on page 1-103.

```
CONNECT TO 'stores7'
...
CONNECT TO 'demo7'
...
SET CONNECTION 'stores7'
```

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

Use the DEFAULT option to identify 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 made using the DATABASE, CREATE DATABASE, or START DATABASE statements)

You can use SET CONNECTION without a DORMANT option to reestablish the default connection or with the DORMANT option to make the default connection dormant. See "DEFAULT Option" on page 1-100 and "Implicit Connection with DATABASE Statements" on page 1-101 for more information.

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

When you issue a SET CONNECTION statement without the DORMANT option, the SET CONNECTION statement implicitly puts the current connection in the dormant state. When you issue a SET CONNECTION statement (with the DORMANT option), the SET CONNECTION statement explicitly puts the current connection in the dormant state. In either case, 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 with the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the SET CONNECTION statement succeeds and puts the connection in a dormant state.
- If the connection was established without the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the SET CONNECTION statement 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.

When Current Connection Is to INFORMIX-OnLine Dynamic Server Prior to Version 6.0

If the current connection is to a version of the OnLine database server prior to 6.0, the following conditions apply when a SET CONNECTION statement with or without the DORMANT option executes:

- If the connection to be made dormant was established using the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the application *can* switch to a different connection.
- If the connection to be made dormant was established without the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the application *cannot* switch to a different connection; the SET CONNECTION statement returns an error. The application must use the CLOSE DATABASE statement to close the database and drop the connection.

References

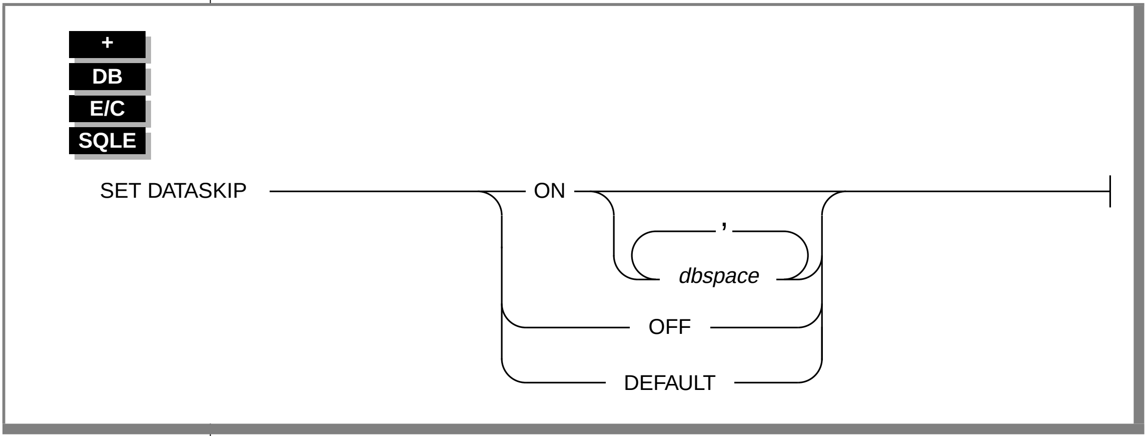
See the CONNECT, DISCONNECT, and DATABASE statements in this manual.

In the *INFORMIX-ESQL/C Programmer's Manual*, see the discussions of the SET CONNECTION statement and thread-safe applications.

SET DATASKIP

The SET DATASKIP statement allows you to control whether Universal Server skips a dbspace that is unavailable (for example, due to a media failure) in the course of processing a transaction.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the skipped dbspace	The dbspace must exist at the time the statement is executed.	Identifier, p. 1-962

Usage

Use the SET DATASKIP statement to instruct the database server to skip a dbspace that is unavailable during the course of processing a transaction.

ESQL

You receive a warning if a dbspace is skipped. The warning flag **sqlca.sqlwarn.sqlwarn6** is set to `W` if a dbspace is skipped. For more information about this topic, see the *INFORMIX-ESQL/C Programmer's Manual*. ♦

When you SET DATASKIP ON without specifying a dbspace, you are telling 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 if a dbspace is down.

When you SET DATASKIP ON *dbspace*, you are telling the database server to skip the specified dbspace if it is unavailable.

Use the SET DATASKIP OFF statement to turn off the dataskip feature.

When the setting is DEFAULT, the database server uses the setting for the dataskip feature from the ONCONFIG file. The Universal Server administrator can change the setting of the dataskip feature at runtime. See the *INFORMIX-Universal Server Administrator's Guide* for more information.

Under What Circumstances Is a Dbspace Skipped?

The database server skips a dbspace when SET DATASKIP is set to ON and the dbspace is unavailable. 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. The child rows must exist in an available fragment.

  When you want to insert a new child table, the parent table 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 in 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 happens during insert statements.

References

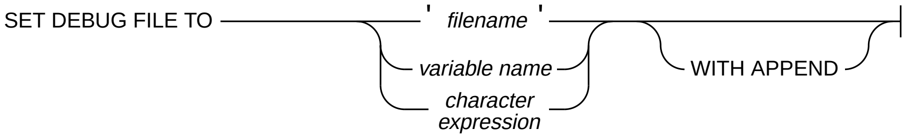
For additional information about how to set the dataskip feature in the ONCONFIG file and how to use the **onspaces** utility, see the *INFORMIX-Universal Server Administrator's Guide*.

SET DEBUG FILE TO

Use the SET DEBUG FILE TO statement to name the file that is to hold the runtime trace output of a stored procedure.

Syntax

+
DB
E/C
SQLE



Element	Purpose	Restrictions	Syntax
character expression	An expression that evaluates to a filename	The filename that is derived from the expression must be usable. The same restrictions apply to the derived filename as to the *filename* parameter.	Expression, p. 1-876
filename	A quoted string that identifies the pathname and filename of the file that contains the output of the TRACE statement. See "Location of the Output File" on page 1-694 for information on the default actions that are taken if you omit the pathname.	You can specify a new or existing file. If you specify an existing file, you must include the WITH APPEND keywords if you want to preserve the current contents of the file intact. See "Using the WITH APPEND Option" on page 1-693 for further information.	Quoted String, p. 1-1010. The pathname and filename must conform to the conventions of your operating system.
variable name	A host variable that holds the value of *filename*	The host variable must be a character data type.	The name of the host variable must conform to language-specific rules for variable names.

Usage

This statement indicates that the output of the TRACE statement in the stored procedure 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 TO file statement can be a new file or existing file.

If you specify an existing file, the current contents of the file are purged when you issue the SET DEBUG TO FILE statement. The first execution of a TRACE command sends trace output to the beginning of the file.

However, if you include the WITH APPEND option, the current contents of the file are preserved when you issue the SET DEBUG TO FILE 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 TO FILE 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 a procedure to direct the trace output of the procedure to a file. You also can use this statement inside a procedure 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.

Example of the SET DEBUG FILE TO Statement

The following example sends the output of the SET DEBUG FILE TO statement to a file called **debugging.out**:

```
SET DEBUG FILE TO 'debugging' || '.out'
```

References

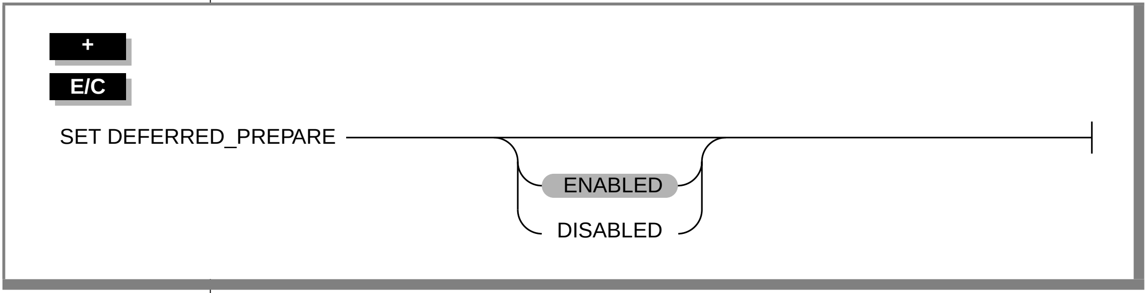
See the TRACE statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of stored procedures in Chapter 14.

SET DEFERRED_PREPARE

The SET DEFERRED_PREPARE statement enables or disables the Deferred-PREPARE feature for cursors in an INFORMIX-ESQL/C application program.

Syntax



Usage

The Deferred-PREPARE feature is one of the ESQL/C optimization features that can minimize network traffic when an ESQL/C application receives rows from a database server. When the Deferred-PREPARE feature is enabled, ESQL/C saves a round trip of message requests because it does not need to send the PREPARE statement separately to the database server for execution. Instead, ESQL/C sends the PREPARE and OPEN statements to the database server at the same time.

The SET DEFERRED_PREPARE statement allows an ESQL/C application to:

- enable the Deferred-PREPARE feature.

  Use the ENABLED option of the SET DEFERRED_PREPARE statement.
- disable the Deferred-PREPARE feature.

  Use the DISABLED option of the SET DEFERRED_PREPARE statement.

If you do not specify either option, the default is ENABLED. The following SET DEFERRED_PREPARE statement enables the Deferred-PREPARE feature for all prepared statements in the application:

```
EXEC SQL set deferred_prepare;
```

ENABLED Option

The ENABLED option enables the Deferred-PREPARE feature within the ESQL/C application. The following SET DEFERRED_PREPARE statement enables the Deferred-PREPARE feature:

```
EXEC SQL set deferred_prepare enabled;
```

ESQL/C automatically defers execution of any prepared statement that is prepared *after* this SET DEFERRED_PREPARE statement executes.

Important: *You can also use the **IFX_DEFERRED_PREPARE** environment variable to enable the Deferred-PREPARE feature. For more information on the **IFX_DEFERRED_PREPARE** environment variable, see the "Using the IFX_DEFERRED_PREPARE Environment Variable" on page 1-697.*

When you enable the Deferred-PREPARE feature, the application then exhibits the following behavior:

- The sequence PREPARE/EXECUTE returns an error on the EXECUTE statement.

  The Deferred-PREPARE feature does not defer the execution of a PREPARE statement until an EXECUTE statement. This feature is meant to work primarily with PREPARE, DECLARE, OPEN sequences that operate with the FETCH or PUT statements. If you enable the Deferred-PREPARE feature before such a PREPARE/EXECUTE sequence executes, the EXECUTE statement generates an error.

- The sequence PREPARE/DESCRIBE/OPEN returns an error on the DESCRIBE statement.

  A DESCRIBE statement must execute on a prepared statement after the associated cursor has been opened with an OPEN statement. The following sequence of statements is valid: PREPARE, DECLARE, OPEN, DESCRIBE. If you enable the Deferred-PREPARE feature before such a PREPARE/DESCRIBE/OPEN sequence executes, the DESCRIBE statement generates an error.

- The sequence PREPARE, DECLARE, OPEN sends the PREPARE statement to the database server with the OPEN statement.

  If a prepared statement contains syntax errors, the database server does not return error messages to the application until the application has declared a cursor for the prepared statement and opened the cursor.

DISABLED Option

The DISABLED option disables the Deferred-PREPARE feature within the ESQL/C application. The following SET DEFERRED_PREPARE statement disables the Deferred-PREPARE feature:

```
EXEC SQL set deferred_prepare disabled;
```

ESQL/C atomically resumes execution of any prepared statement when the PREPARE statement *after* this SET DEFERRED_PREPARE statement executes. None of the application restrictions listed in "ENABLED Option" on page 1-674 applies when the Deferred-PREPARE feature is disabled.

Using the IFX_DEFERRED_PREPARE Environment Variable

You can also enable or disable the Deferred-PREPARE feature with the **IFX_DEFERRED_PREPARE** environment variable, as follows:

- Set this environment variable to 1 to *enable* the Deferred-PREPARE feature for every prepared statement in every thread of the application.
- Set this environment variable to 0 to *disable* the Deferred-PREPARE feature for every prepared statement in every thread of the application.

If you do not set the **IFX_DEFERRED_PREPARE** environment variable, the Deferred-PREPARE feature is disabled. However, in each thread, a SET DEFERRED_PREPARE statement overrides the value of the **IFX_DEFERRED_PREPARE** environment variable.

The **IFX_DEFERRED_PREPARE** environment variable works only with client applications such as those written in INFORMIX-ESQL/C. This environment variable has no effect on Informix database utilities such as DB-Access, **dbexport**, **dbimport**, **dbload**, and **dbschema**.

For further information on the **IFX_DEFERRED_PREPARE** environment variable, see the *Informix Guide to SQL: Reference*.

References

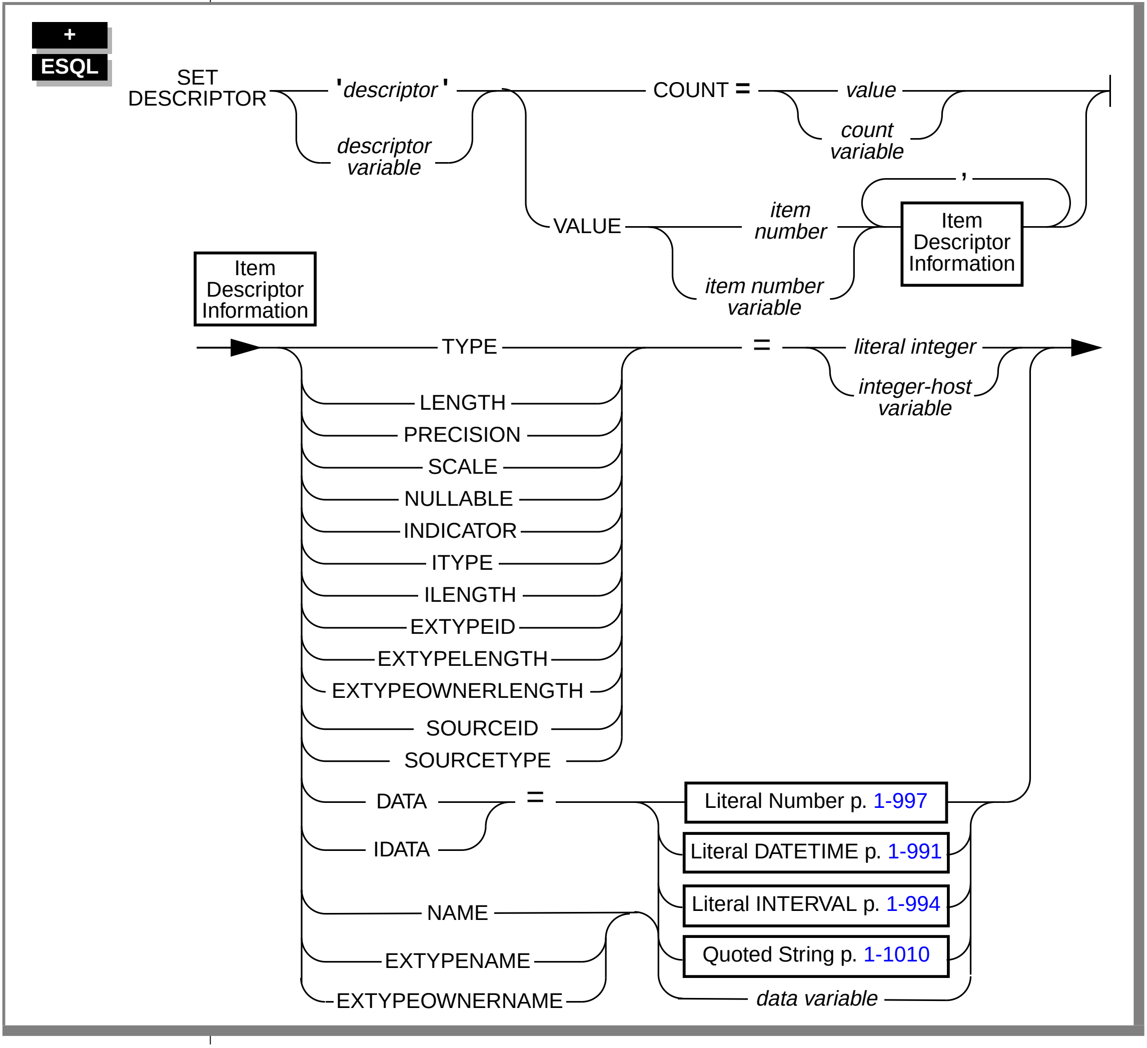
See the DECLARE, DESCRIBE, EXECUTE, OPEN, and PREPARE statements in this manual. For another ESQL/C optimization, see the SET AUTOFREE statement.

In the *INFORMIX-ESQL/C Programmer's Manual*, see the chapter on how to use dynamic SQL for information on the Deferred-PREPARE feature.

SET DESCRIPTOR

Use the SET DESCRIPTOR statement to assign values to a system-descriptor area.

Syntax



Element	Purpose	Restrictions	Syntax
count variable	A host variable that holds a literal integer. This integer specifies how many items are actually described in the system-descriptor area.	See restriction for *value* in this table.	The name of the host variable must conform to language-specific rules for variable names.
data variable	A host variable that contains the information for the specified field (DATA, IDATA, or NAME) in the specified item descriptor	The information that is contained in *data variable* must be appropriate for the specified field.	The name of the host variable must conform to language-specific rules for variable names.
descriptor	A quoted string that identifies the system-descriptor area to which values will be assigned	The system-descriptor area must have been previously allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-1010
descriptor variable	A host variable that holds the value of *descriptor*	The same restrictions apply to *descriptor variable* as apply to *descriptor.*	The name of the host variable must conform to language-specific rules for variable names.
integer host variable	The name of a host variable that contains the value of *literal integer*	The same restrictions apply to *integer host variable* as apply to *literal integer.*	The name of the host variable must conform to language-specific rules for variable names.
item number	An unsigned integer that specifies one of the occurrences (item descriptors) in the system-descriptor area	The value of *item number* must be greater than 0 and less than (or equal to) the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-997

(1 of 2)

Element	Purpose	Restrictions	Syntax
item number variable	The name of an integer host variable that holds the value of *item number*	The same restrictions apply to *item number variable* as apply to *item number.*	The name of the host variable must conform to language-specific rules for variable names.
literal integer	A positive, nonzero integer that assigns a value to the specified field in the specified item descriptor. The specified field must be one of the following keywords: TYPE, LENGTH, PRECISION, SCALE, NULLABLE, INDICATOR, ITYPE, or ILENGTH.	The restrictions that apply to *literal integer* vary with the field type you specify in the VALUE option (TYPE, LENGTH, and so on). For information on the codes that are allowed for the TYPE field and their meaning, see "Setting the TYPE Field" on page 1-704. For the restrictions that apply to other field types, see the individual headings for field types under "VALUE Clause" on page 1-703.	Literal Number, p. 1-997
value	A literal integer that specifies how many items are actually described in the system-descriptor area	The integer that *value* specifies must be greater than 0 and less than (or equal to) the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-997

(2 of 2)

Usage

The SET DESCRIPTOR statement can be used after you have described SELECT, EXECUTE FUNCTION, and INSERT statements with the DESCRIBE...USING SQL DESCRIPTOR statement. The SET DESCRIPTOR statement can assign values to a system-descriptor area in the following instances:

- 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 field in a system-descriptor area, the contents of all identified fields are set to 0 or null, depending on the variable type.

Using the COUNT Keyword

Use the COUNT keyword to set the number of items that are to be used in the system-descriptor area (typically the items are in a WHERE clause). 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 the sequence of statements in INFORMIX-ESQL/C that can be used in a 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;
```

VALUE Clause

Use the VALUE clause to assign values from host variables into fields for a particular item in 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 the DESCRIBE statement to fill the fields for a SELECT or an INSERT statement.

The *item number* must be greater than zero and less than the number of occurrences that were specified when you allocated the system-descriptor area with the ALLOCATE DESCRIPTOR statement.

Setting the TYPE Field

Use the following codes to set the value of TYPE for each item.

SQL Data Type	Integer Value
CHAR	0
SMALLINT	1
INTEGER	2
FLOAT	3
SMALLFLOAT	4
DECIMAL	5
SERIAL	6
DATE	7
MONEY	8
DATETIME	10
BYTE	11
TEXT	12
VARCHAR	13
INTERVAL	14
NCHAR	15
NVARCHAR	16
INT8	17
SERIAL8	18
SET	19
MULTISET	20
LIST	21
ROW	22
COLLECTION	23
Varying-length opaque type	40
Fixed-length opaque type	41
LVARCHAR (client-side only)	43
BOOLEAN	45

These TYPE constants are the same values that the **coltype** column in the **syscolumns** system catalog table.

For code that is easier to maintain, use the predefined constants for these SQL data types instead of their actual integer value. These constants are defined in the **sqltypes.h** header file. However, you cannot 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.

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;
}
```

Compiling without the -xopen option

If you do not compile using the **-xopen** option, the regular 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 particular type is not defined under X/Open mode but is defined under normal mode, executing a SET DESCRIPTOR statement can result in an error.

Setting the TYPE field in X/Open programs

X/O

In X/Open mode, you must use the X/Open set of integer codes for the data type in the TYPE field. The following table shows the X/Open codes for data types.

SQL Data Type	Integer Value
CHAR	1
SMALLINT	4
INTEGER	5
FLOAT	6
DECIMAL	3

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 **sqlxtype.h** header file. However, you cannot 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.♦

Setting the DATA Field

When you set the DATA field, you must provide the appropriate type of data (character string for CHAR or VARCHAR, integer for INTEGER, and so on).

When any value other than DATA is set, the value of DATA is undefined. You cannot set the DATA 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.

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 sets automatically 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 the length of a string in the DATA or IDATA field dynamically.

If a DESCRIBE statement precedes a SET DESCRIPTOR statement, DESCRIBE automatically sets LENGTH 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.

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, you can set the DATA field as a literal DATETIME or INTERVAL or as a character string. If you use a character string, you must set the LENGTH field to 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 Indicator Fields

If you want to put a null value in the system-descriptor area, set the following item-descriptor fields:

- 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.
- The ITYPE field expects an integer constant that indicates the data type of your indicator variable.

  Use the same set of constants as for the TYPE field. The constants are listed on page 1-704.

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 Chapter 1 of the *Informix Guide to SQL: Reference*.

Setting Distinct-Type Fields

The following item-descriptor fields provide information about a column that has an 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 1-699). 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 Chapter 1 of the *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 a DESCRIBE statement on 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.

References

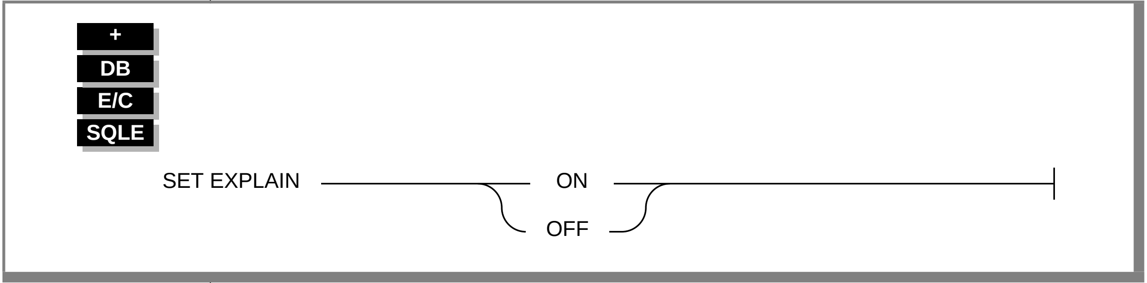
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, and PUT statements in this manual for further information about using dynamic SQL statements.

For further information about the system-descriptor area, see the *INFORMIX-ESQL/C Programmer's Manual*.

SET EXPLAIN

Use the SET EXPLAIN statement to obtain a measure of the work involved in performing a query.

Syntax



Usage

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, it occurs when you open the cursor.

When you issue a SET EXPLAIN ON statement, the path that the optimizer chooses for each subsequent query is written to the **sqexplain.out** file. The SET EXPLAIN ON statement remains in effect until you issue a SET EXPLAIN OFF statement or until the program ends. The owner name (for example, *owner.customer)* qualifies table names in the **sqexplain.out** file.

If the file already exists, 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.

When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. If you do not have a home directory on the remote host, the program stores **sqexplain.out** in the directory from which the database server was started.

SET EXPLAIN Output

The SET EXPLAIN output file contains a copy of the query, a plan of execution that the database-server optimizer selects, and an estimate of the amount of work. The optimizer selects a plan to provide the most efficient way to perform the query, based on such things as the presence and type of indexes and the number of rows in each table.

The optimizer uses an estimate to compare the cost of one path with another. The estimated cost does not translate directly into time. However, when data distributions are used, a query with a higher estimate generally takes longer to run than one with a smaller estimate.

The estimated cost of the query is included in the SET EXPLAIN output. 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.

In addition to the estimated cost, the output file contains the following information:

- An estimate of the number of rows to be returned
- The order in which tables are accessed during execution
- The table column or columns that serve as a filter, if any, and whether the filtering occurs through an index
- The method (access path) by which the executor reads each table. The following list shows the possible methods.

Method	Effect
SEQUENTIAL SCAN	Reads rows in sequence
INDEX PATH	Scans one or more indexes
AUTOINDEX PATH	Creates a temporary index
SORT SCAN	Sorts the result of the preceding join or table scan
MERGE JOIN	Uses a sort / merge join instead of nested-loop join
REMOTE PATH	Accesses another distributed database
HASH JOIN	Uses a hash join

The optimizer chooses the best path of execution to produce the fastest possible table join using a nested-loop join or sort-merge join wherever appropriate.

The SORT SCAN section indicates that sorting the result of the preceding join or table scan is necessary for a sort-merge join. It includes a list of the columns that form the sort key. The order of the columns is the order of the sort. As with indexes, the default order is *ascending*. Where possible, this ordering is arranged to support any requested ORDER BY or GROUP BY clause. If the ordering can be generated from a previous sort or an index lookup, the SORT SCAN section does not appear.

The MERGE JOIN section indicates that a sort-merge join, instead of the nested-loop join, is to be used on the preceding join / table pair. It includes a list of the filters that control the sort-merge join and, where applicable, a list of any other join filters. For example, a join of tables A and B with the filters `A.c1 = B.c1` and `A.c2 < B.c2` lists the first join under "Merge Filters" and the second join under "Other Join Filters."

The DYNAMIC HASH JOIN section indicates that a hash join is to be used on the preceding join / table pair. It includes a list of the filters used to join the tables together.

A dynamic hash join uses one of the tables to construct a *hash* index and adds the index for the other table into the hash index. This is referred to as the *build phase*. If DYNAMIC HASH JOIN is followed by the (Build Outer) in the output, then the build phase is occurring on the first table; otherwise it occurs on the second table, preceding the DYNAMIC HASH JOIN. In the following example, the build phase occurs on table **username.a**:

```
SELECT a.adatetime FROM manytypes a, alltypes b
    WHERE a.adatetime = b.adate and a.along + 7 = b.along/3

Estimated Cost: 10
Estimated # of Rows Returned: 2

1) username.a: SEQUENTIAL SCAN
2) username.b: SEQUENTIAL SCAN

DYNAMIC HASH JOIN
    Dynamic Hash Filters: username.a.adatetime =
    username.b.adate and a.along + 7 = b.along/3
```

The following output examples represent what you might see when a SET EXPLAIN ON statement is issued using INFORMIX-Universal Server.

The first two examples contain two entries for a multiple-table query and show the SORT SCAN and MERGE JOIN lines. Note that in both cases, if SORT MERGE was not chosen, the second table would have been scanned using an *autoindex path.* An autoindex path is an index constructed automatically at execution time by the database server. It is removed when the query completes.

```
QUERY:
-----------
select i.stock_num from items i, stock s, manufact m
    where i.stock_num = s.stock_num
    and i.manu_code = s.manu_code
    and s.manu_code = m.manu_code

Estimated Cost: 52
Estimated # of Rows Returned: 130

1) rdtest.m: SEQUENTIAL SCAN

SORT SCAN: rdtest.m.manu_code

2) rdtest.s: SEQUENTIAL SCAN

SORT SCAN: rdtest.s.manu_code

MERGE JOIN:
   Merge Filters: rdtest.m.manu_code = rdtest.s.manu_code

3) rdtest.i: INDEX PATH

(1) Index Keys: stock_num manu_code
    Lower Index Filter: (rdtest.i.stock_num = rdtest.s.stock_num AND
    rdtest.i.manu_code = rdtest.s.manu_code)

QUERY:
-----------
select stock.description from stock, stock2
    where stock.description = stock2.description
    and stock.unit_price < stock2.unit_price

Estimated Cost: 15
Estimated # of Rows Returned: 370

1) rdtest.stock: SEQUENTIAL SCAN

SORT SCAN: rdtest.stock.description

2) rdtest.stock2: SEQUENTIAL SCAN

SORT SCAN: rdtest.stock2.description

MERGE JOIN
   Merge Filters: rdtest.stock2.description = rdtest.stock.description
   Other Join Filters: rdtest.stock.unit_price < rdtest.stock2.unit_price
```

The following example shows the SET EXPLAIN output for a simple query and a complex query from the **customer** table:

```
QUERY:
-----------
SELECT fname, lname, company FROM customer

Estimated Cost: 3
Estimated # of Rows Returned: 28

1) joe.customer: SEQUENTIAL SCAN


QUERY:
 ------
SELECT fname, lname, company FROM customer
    WHERE company MATCHES 'Sport*' AND customer_num BETWEEN 110 AND 115
    ORDER BY lname;

Estimated Cost: 4
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

1) joe.customer: INDEX PATH

Filters: joe.customer.company MATCHES 'Sport*'

 (1) Index Keys: customer_num
  Lower Index Filter: joe.customer.customer_num >= 110
  Upper Index Filter: joe.customer.customer_num <= 115
```

The following example shows the SET EXPLAIN output for a multiple-table query:

```
QUERY:
-----------
SELECT * FROM customer, orders, items
    WHERE customer.customer_num = orders.customer_num
    AND orders.order_num = items.order_num

Estimated Cost: 20
Estimated # of Rows Returned: 69

1) joe.orders: SEQUENTIAL SCAN

2) joe.customer: INDEX PATH

 (1) Index Keys: customer_num
 Lower Index Filter: joe.customer.customer_num = joe.orders.customer_num

3) joe.items: INDEX PATH

  (1) Index Keys: order_num
  Lower Index Filter: joe.items.order_num  = joe.orders.order_num
```

SET EXPLAIN Output with Fragmentation and PDQ

When the table is fragmented, the output shows which table or index is scanned. Fragments are identified with a fragment number. The fragment numbers are the same as those contained in the dbspace column in the **sysfragments** system catalog table. If the optimizer must scan all fragments (that is, if it is unable to eliminate any fragment from consideration), the optimizer indicates this with ALL. In addition, if the optimizer eliminates all the fragments from consideration, that is, none of the fragments contain the queried information, the optimizer indicates this with NONE. For information on how Universal Server eliminates a fragment from consideration, see the INFORMIX-Universal Server.

When PDQ is turned on, the output shows whether the optimizer used parallel scans. If the optimizer used parallel scans, the output shows PARALLEL; if PDQ is turned off, the output shows SERIAL. If PDQ is turned on, the optimizer indicates the maximum number of threads that are required to answer the query. The output shows `# of Secondary Threads`. This field indicates the number of threads that are required in addition to your user session thread. The total number of threads necessary is the number of secondary threads plus 1.

The output indicates when a hash join is used. The query is marked with DYNAMIC HASH JOIN, and the table on which the hash is built is marked with Build Outer.

The following example shows the SET EXPLAIN output for a table with fragmentation and PDQ priority set to low:

```
select * from t1 where c1 > 20

Estimated Cost: 2
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Parallel, fragments: 2)

 Filters: informix.t1.c1 > 20

# of Secondary Threads = 1
```

The following example of SET EXPLAIN output shows a table with fragmentation but without PDQ:

```
select * from t1 where c1 > 12

Estimated Cost: 3
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Serial, fragments: 1, 2)

    Filters: informix.t1.c1 > 12
```

The following example of SET EXPLAIN output shows a table with hash join (fragmentation, and PDQ priority set to ON). The hash join is created when you create an equality join between two tables that are not indexed.

```
QUERY:
------
select h1.c1, h2.c1 from h1, h2 where h1.c1=h2.c1

Estimated Cost: 2
Estimated # of Rows Returned: 5

1) informix.h1: SEQUENTIAL SCAN (Parallel, fragments: ALL)

2) informix.h2: SEQUENTIAL SCAN (Parallel, fragments: ALL)


DYNAMIC HASH JOIN (Build Outer)
 Dynamic Hash Filters: informix.h1.c1 = informix.h2.c1

# of Secondary Threads = 6
```

The following example of SET EXPLAIN output shows a table with fragmentation, with PDQ priority set to LOW, and an index that was selected as the search method:

```
QUERY:
------
select * from t1 where c1 < 13

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.t1: INDEX PATH

 (1) Index Keys: c1 (Parallel, fragments: ALL)
 Upper Index Filter: informix.t1.c1 < 13


# of Secondary Threads = 3
```

Using SET EXPLAIN With SET OPTIMIZATION

If you SET OPTIMIZATION to low, the output of SET EXPLAIN displays the following uppercase string:

```
QUERY:{LOW}
```

If you SET OPTIMIZATION to high, the output of SET EXPLAIN displays the following uppercase string:

```
QUERY:
```

SET EXPLAIN Output With Table Inheritance

The SET EXPLAIN statement returns information about the table inheritance that a query uses. Suppose you a super table, **super_tab**, and two subtables, **sub_tab11** and **sub_tab21**. In addition, you have a subtable **sub_tab22** that is derived from **sub_tab21**. The following SQL statements create this table inheritance:

```
CREATE ROW TYPE super_tab (id INTEGER, a SMALLINT);
CREATE ROW TYPE sub_tab21 (b INTEGER) UNDER super_tab;
CREATE ROW TYPE sub_tab22 (C DECIMAL(5,2)) UNDER sub_tab21;
CREATE ROW TYPE sub_tab11 (d CHAR(5)) UNDER super_tab;
```

Suppose further that you now run the following query (with SET EXPLAIN set to ON):

```
SELECT DISTINCT TYPE id FROM super_tab WHERE a IN (15, -23, 42, 17);
```

The following example shows the SET EXPLAIN output for this query:

```
select distinct type id from super_tab where a in (15, -23, 42, 17);

Estimated Cost: 12
Estimated # of Rows Returned: 1

1) USERNAME.super_tab: SEQUENTIAL SCAN (Serial)
USERNAME.sub_tab21: SEQUENTIAL SCAN (Serial)
USERNAME.sub_tab11: SEQUENTIAL SCAN (Serial)
USERNAME.sub_tab22: SEQUENTIAL SCAN (Serial)

Filters: USERNAME.super_tab.a IN (15, -23, 43, 17)
```

Reference

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of SET EXPLAIN and the optimizer discussion.

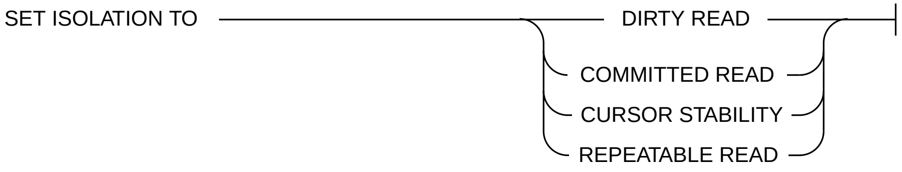
SET ISOLATION

Use the SET ISOLATION statement with INFORMIX-Universal Server to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

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. See the SET TRANSACTION statement on page 1-738 for a comparison of these two statements.

Syntax

+
DB
E/C
SQLE



Usage

The database isolation level affects read concurrency when rows are retrieved from the database. Universal Server uses shared locks to support four 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 will be denied access to those rows.

ESQL

Cursors that are currently open when you execute the SET ISOLATION statement might or might not use the new isolation level when rows are later retrieved. The isolation level in effect could be any level that was set from the time the cursor was opened until the time the application actually fetches a row. 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 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.

Isolation Level	Characteristics
Dirty Read	Provides zero isolation. Dirty Read is appropriate for static tables that are used for queries. With a Dirty Read isolation level, a query 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. Dirty Read is the only isolation level available to databases that do not have transactions.
Committed Read	Guarantees that every retrieved row is committed in the table at the time that the row is retrieved. Even so, no locks are acquired. After one process retrieves a row because no lock is held on the row, another process can acquire an exclusive lock on the same row and modify or delete data in the row. Committed Read is the default level of isolation in a database with logging that is not ANSI compliant.

(1 of 2)

Isolation Level	Characteristics
Cursor Stability	Acquires a shared lock on the selected row. Another process can also acquire a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row. When you fetch another row or close the cursor, Universal Server releases the shared lock. 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. Locks are acquired when the isolation level is set to Cursor Stability outside a transaction, but they are released immediately at the end of the statement that reads the row.
Repeatable Read	Acquires a shared lock on every row that is selected during the transaction. Another process can also acquire a shared lock on a selected row, but no other process can modify any selected row 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.

(2 of 2)

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 of isolation in a database without logging
Committed Read	Default level of isolation in a database with logging that is not ANSI compliant
Repeatable Read	Default level of isolation 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 isolation level that your last SET ISOLATION statement specified.
- The program ends.

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 has been opened.

The data obtained during blob retrieval can vary, depending on the database isolation level. Under Dirty Read or Committed Read levels of isolation, a process is permitted to read a blob 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 blob when certain conditions exist. See the *INFORMIX-Universal Server Administrator's Guide* for information about these conditions.

DB

When you use DB-Access, you see more lock conflicts with higher levels of isolation. For example, if you use Cursor Stability, you see more lock conflicts than if you use Committed Read. ♦

ESQL

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 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. ♦

References

See the CREATE DATABASE, SET LOCK MODE, and SET TRANSACTION statements in this manual.

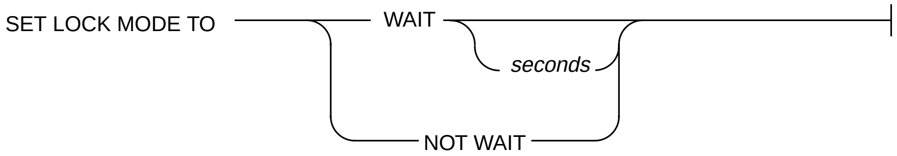
In the *Informix Guide to SQL: Tutorial*, see the discussion of isolation levels in Chapter 7.

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

+
DB
E/C
SQLE



Element	Purpose	Restrictions	Syntax
seconds	The maximum number of seconds that a process waits for a lock to be released. If the lock is still held at the end of the waiting period, the database server ends the operation and returns an error code to the process.	In a networked environment, the DBA establishes a default value for the waiting period by using the ONCONFIG parameter DEADLOCK_TIMEOUT. See "WAIT Keyword" on page 1-725 for an explanation of when the *seconds* parameter overrides the DEADLOCK_TIMEOUT parameter.	Literal Number, p. 1-997

Usage

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	Ends the operation immediately and returns an error code. This condition is the default.
WAIT	Suspends the process until the lock releases
WAIT *seconds*	Suspends the process until the lock releases or until the end of a waiting period, which is specified in seconds. If the lock remains after the waiting period, it ends the operation and returns an error code.

The SET LOCK MODE statement is available on computers that use kernel locking. To determine whether your computer uses kernel locking, check the directory that holds the database files. If the directory contains files with the extension **.lok**, your system does not use kernel locking, and the SET LOCK MODE statement is unavailable.

WAIT Keyword

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.

References

See the LOCK TABLE, UNLOCK TABLE, SET ISOLATION, and SET TRANSACTION statements in this manual.

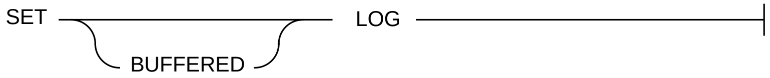
In the *Informix Guide to SQL: Tutorial*, see the discussion of SET LOCK MODE in Chapter 7.

SET LOG

Use the SET LOG statement to change your database server logging mode from buffered transaction logging to unbuffered transaction logging or vice versa.

Syntax

+
DB
E/C
SQLE



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.

The default condition for transaction logs is unbuffered logging. As soon as a transaction ends, the database server writes the transaction to the disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions.

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 the completed transactions that were buffered in memory.

The SET LOG statement changes the transaction-logging mode to unbuffered logging; the SET BUFFERED LOG statement changes the mode to buffered logging.

The SET LOG statement redefines the mode for the current session only. The default mode, which the Universal Server administrator sets using ON-Monitor, 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 logs. ♦

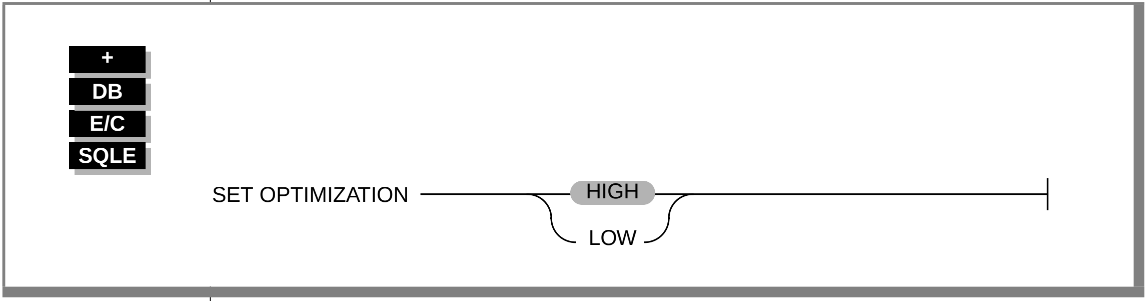
References

See the CREATE DATABASE and START DATABASE statements in this manual.

SET OPTIMIZATION

Use the SET OPTIMIZATION statement to specify a high or low level of database server optimization.

Syntax



Usage

You can execute a SET OPTIMIZATION statement at any time. The optimization level carries across databases but applies only within the current database server.

After a SET OPTIMIZATION statement executes, the new optimization level remains in effect until you enter another SET OPTIMIZATION statement or until the program ends.

The default database server optimization level, HIGH, remains in effect until you issue another SET OPTIMIZATION statement. The LOW option invokes a less sophisticated, but faster, optimization algorithm.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is a sophisticated, cost-based strategy that examines all reasonable choices and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

The following example shows optimization across a network. The **central** database (on computer 1) is to have LOW optimization; the **western** database (on computer 2) is to have HIGH optimization. If the **western** database were on the same computer as **central**, it would have LOW optimization.

```
CONNECT TO 'central';
SET OPTIMIZATION low;
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
CLOSE DATABASE;
CONNECT TO 'western@rockie';
SET OPTIMIZATION low;
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

Optimizing SPL Routines

In earlier Informix products, the term *stored procedure* was used for both SPL procedures and SPL functions. In Universal Server, the term *SPL routine* is used for both SPL procedures and SPL functions.

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 routine. This optimization level stores the best query plans for the routine. Then SET OPTIMIZATION to LOW before you execute the routine. The routine then uses the optimal query plans and runs at the more cost-effective rate.

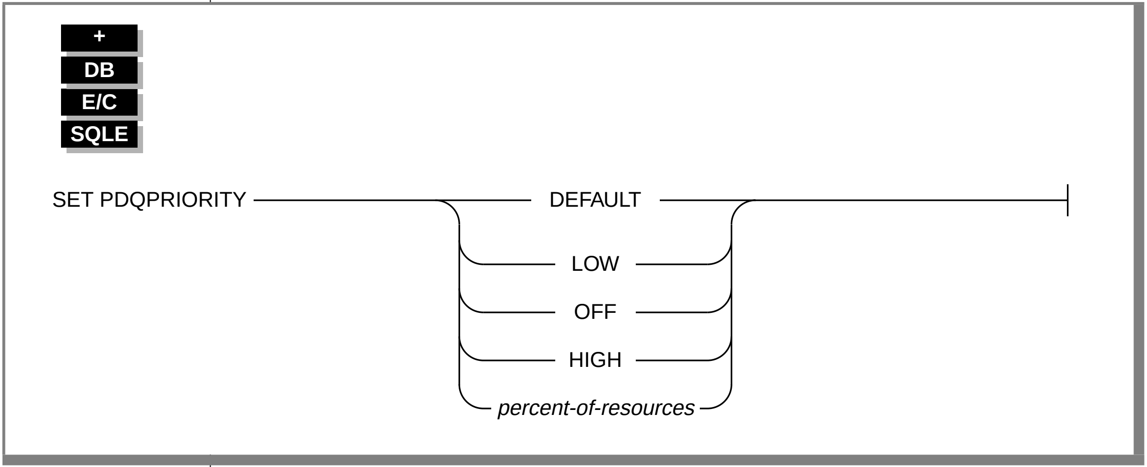
References

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of optimizing queries.

SET PDQPRIORITY

The SET PDQPRIORITY statement allows an application to set the query priority level dynamically within an application.

Syntax



Element	Purpose	Restrictions	Syntax
percent-of-resources	An integer value that specifies the query priority level and the amount of resources the database server uses in order to process the query	You must specify a value in the following range: -1, 0, 1 to 100. The values -1, 0, and 1 have special meanings. See "Meaning of SET PDQPRIORITY Parameters" on page 1-732 for an explanation of these values.	Literal Number, p. 1-997

Usage

Priority set with the SET PDQPRIORITY statement overrides the environment variable **PDQPRIORITY**. However, no matter what priority value you set with the SET PDQPRIORITY statement, the ONCONFIG configuration parameter MAX_PDQPRIORITY determines the actual priority value that the INFORMIX-Universal Server uses for your queries.

For example, assume that the DBA has set the MAX_PDQPRIORITY parameter to 50. A user enters the following SET PDQPRIORITY statement to set the query priority level to 80.

```
SET PDQPRIORITY 80
```

When it processes the user's query, Universal Server uses the value of the MAX_PDQPRIORITY parameter to factor the query priority level set by the user. Universal Server silently processes the query with a priority level of 40. This priority level represents 50 percent of the 80 percent of resources specified by the user.

Meaning of SET PDQPRIORITY Parameters

The parameters that the SET PDQPRIORITY statement can use are shown in the following table.

Parameter	Meaning
DEFAULT	Uses the value that is specified in the PDQPRIORITY environment variable, if any. DEFAULT is the symbolic equivalent of -1.
LOW	Signifies that data is fetched from fragmented tables in parallel. Universal Server uses no other forms of parallelism. LOW is the symbolic equivalent of 1.
OFF	Indicates that PDQ is turned off. Universal Server uses no parallelism. OFF is the symbolic equivalent of 0. OFF is the default setting if you do not specify the PDQPRIORITY environment variable or the SET PDQPRIORITY statement.

(1 of 2)

Parameter	Meaning
HIGH	Signifies that the database server determines an appropriate value to use for PDQPRIORITY. This decision is based on several things, including the number of available processors, the fragmentation of the tables being queried, the complexity of the query, and so on. Informix reserves the right to change the performance behavior of queries when HIGH is specified in future releases.
percent-of-resources	Indicates a query priority level and indicates the percent of resources a database server uses in order to answer the query. Resources include the amount of memory and the number of processors. The higher the number, the more resources the database server uses. Although usually the more resources a database server uses indicates better performance for a given query, using too many resources can cause contention among the resources and remove resources from other queries, which results in degraded performance. Range = -1, 0, 1 to 100.

(2 of 2)

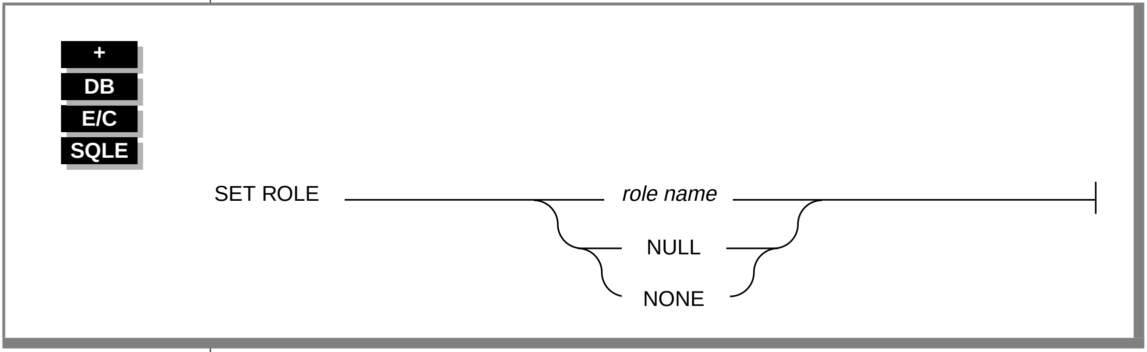
References

For information about the **PDQPRIORITY** environment variable, see the *Informix Guide to SQL: Reference*. See the *INFORMIX-Universal Server Administrator's Guide* for information about the ONCONFIG parameter MAX_PDQPRIORITY.

SET ROLE

Use the SET ROLE statement to enable the privileges of a role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name of the role that you want to enable	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-962

Usage

Any user who is granted a role can enable the role using the SET ROLE statement. A user can enable only one role at a time. If a user executes 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 (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 a user sets a role, the user gains the privileges of the role, in addition to the privileges of PUBLIC and the user's own privileges. If a role is granted to another role, the user gains the privileges of both roles, in addition to those of PUBLIC and the user's own privileges. After a SET ROLE statement executes successfully, the role remains effective until the current database is closed or the user executes another SET ROLE statement. Additionally, the user, not the role, retains ownership of all the objects, such as tables, that were created during a session.

A user 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 stored procedure, and the owner of the trigger or stored procedure was granted the role with the WITH GRANT OPTION, the role is enabled even if the user is not granted the role.

The following example sets the role **engineer:**

```
SET ROLE engineer
```

The following example sets a role and then relinquishes the role after it performs a SELECT operation:

```
EXEC SQL set role engineer;
EXEC SQL select fname, lname, project
        into :efname, :elname, :eproject
        where project_num > 100 and lname = 'Larkin';
printf ("%s is working on %s\n", efname, eproject);
EXEC SQL set role null;
```

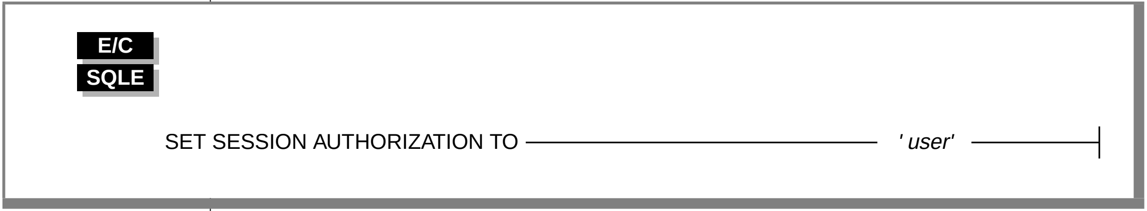
References

See the CREATE ROLE, DROP ROLE, GRANT, and REVOKE statements in this manual.

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement lets you change the user name under which database operations are performed in the current Universal Server session. This statement is enabled by 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.

Syntax



Element	Purpose	Restrictions	Syntax
' *user* '	The user name under which database operations are to be performed in the current session	You must specify a valid user name. You must put quotation marks around the user name.	Identifier, p. 1-962

Usage

The SET SESSION AUTHORIZATION 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.

When you use this statement, the user name to which the authorization is set 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. In other words, 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.

Using SET SESSION AUTHORIZATION to Obtain Privileges

You can use the SET SESSION AUTHORIZATION 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 spec TO mary;
SET SESSION AUTHORIZATION TO 'mary';
UPDATE case
    SET col1 = SELECT state FROM zip
                WHERE zip_code = 94433;
```

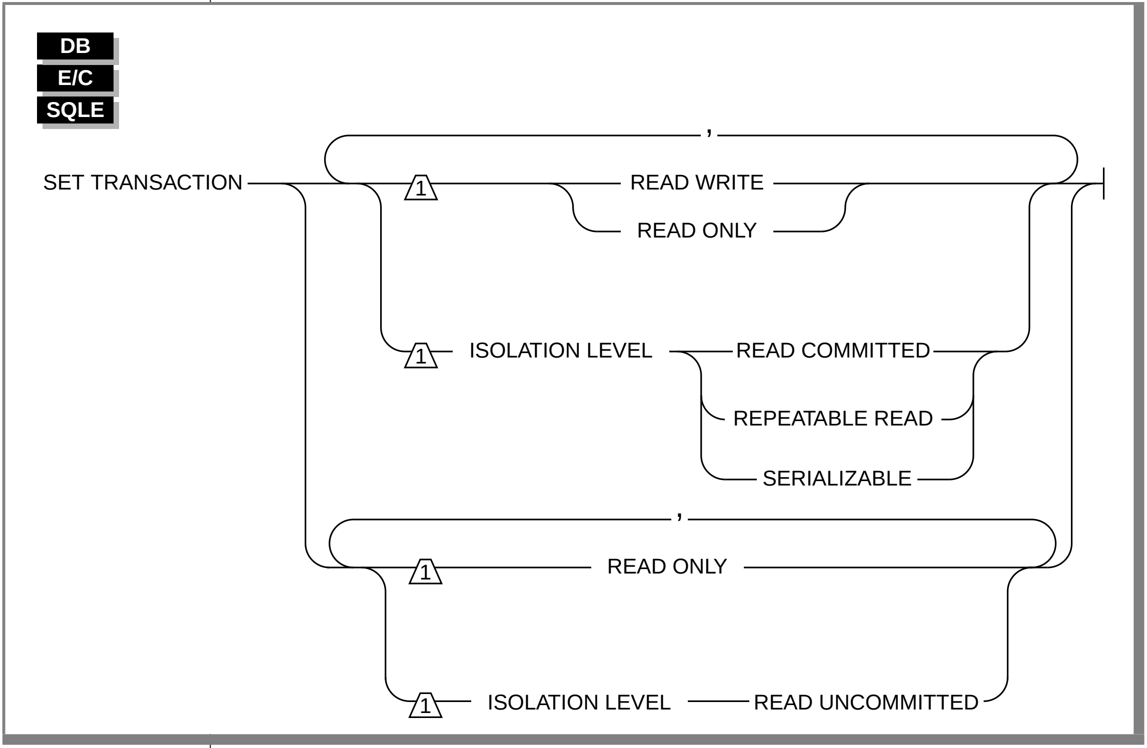
References

See the CONNECT, DATABASE, GRANT, and SET ROLE statements in this manual.

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

You can use SET TRANSACTION only in databases with logging.

You can issue a SET TRANSACTION statement from a client computer only after a database has been opened.

The database isolation level affects concurrency among processes that attempt to access the same rows simultaneously from the database. INFORMIX-Universal Server uses shared locks to support four 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 rows that you are updating or deleting; however, 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	Correlates to	SET ISOLATION
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 the SET TRANSACTION and SET ISOLATION statements is the behavior of the isolation levels within transactions. The SET TRANSACTION statement can be issued only once for a transaction. Any cursors that are opened during that transaction are guaranteed to get that isolation level (or access mode if you are defining an access mode). With the SET ISOLATION statement, after a transaction is started, you can change the isolation level more than once within the transaction. The following examples show the SET ISOLATION and SET TRANSACTION statements, respectively:

SET ISOLATION

```
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
```

SET TRANSACTION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Error 876: Cannot issue SET TRANSACTION in an active
transaction.
```

Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Isolation Level	Characteristics
Read Uncommitted	Provides zero isolation. Read Uncommitted is appropriate for static tables that are used for queries. With a Read Uncommitted isolation level, a query might return a *phantom* row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. Read Uncommitted is the only isolation level that is available to databases that do not have transactions.
Read Committed	Guarantees that every retrieved row is committed in the table at the time that the row is retrieved. Even so, no locks are acquired. After one process retrieves a row because no lock is held on the row, another process can acquire an exclusive lock on the same row and modify or delete data in the row. Read Committed is the default isolation level in a database with logging that is not ANSI compliant.
(ANSI) Repeatable Read	The Informix implementation of ANSI Repeatable Read. Informix uses the same approach to implement Repeatable Read that it uses for Serializable. Thus Repeatable Read meets the SQL-92 requirements.
Serializable	Acquires a shared lock on every row that is selected during the transaction. Another process can also acquire a shared lock on a selected row, but no other process can modify any selected row 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. Serializable is the default isolation level in an ANSI-compliant database.

Default Isolation Levels

The default isolation level for a particular database is established according to database type when you create the database. The default isolation level for each database type is described in the following table.

Informix	ANSI	Description
Dirty Read	Read Uncommitted	Default level of isolation in a database without logging
Committed Read	Read Committed	Default level of isolation in a database with logging that is not ANSI compliant
Repeatable Read	Serializable	Default level of isolation in an ANSI-compliant database

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.

You can specify that a transaction is read-only or read-write through the SET TRANSACTION statement. By default, transactions are read-write. When you specify that a transaction is read-only, certain limitations apply. Read-only transactions cannot perform the following actions:

- Insert, delete, or update table rows
- Create, alter, or drop any database object such as schemas, tables, temporary tables, indexes, or stored procedures
- Grant or revoke privileges
- Update statistics
- Rename columns or tables

You can execute stored procedures in a read-only transaction as long as the procedure 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 blob retrieval can vary, depending on the database isolation levels. Under Read Uncommitted or Read Committed isolation levels, a process is permitted to read a blob 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 blob when certain conditions exist. See the *INFORMIX-Universal Server Administrator's Guide* for information about these conditions.

ESQL

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 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 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. ♦

References

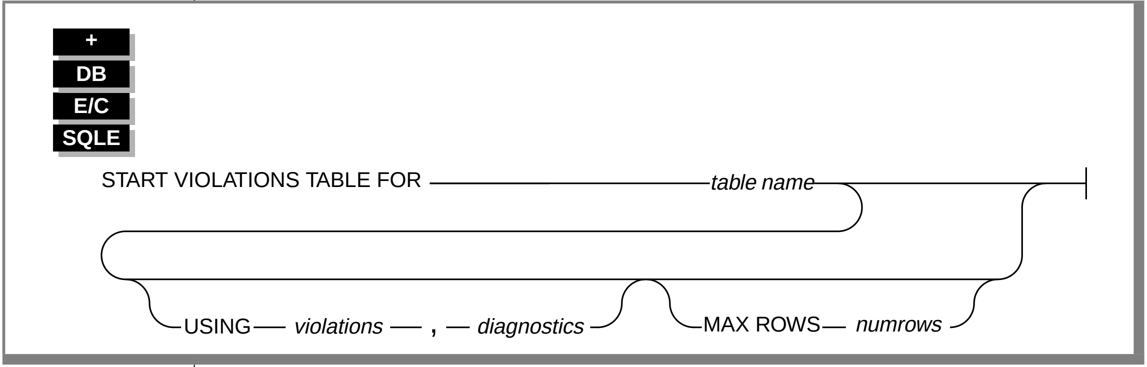
See the CREATE DATABASE, SET ISOLATION, and SET LOCK MODE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of isolation levels and concurrency issues in Chapter 7.

START VIOLATIONS TABLE

The START VIOLATIONS TABLE statement creates a violations table and a diagnostics table for a specified target table. The database server associates the violations and diagnostics tables with the target table by recording the relationship among the three tables in the **sysviolations** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
diagnostics	The name of the diagnostics table to be associated with the target table. The default name is the name of the target table followed by the characters **_dia**. For further information on the diagnostics table, see "Structure of the Diagnostics Table" on page 1-756.	Whether you specify the name of the diagnostics table explicitly, or the database server generates the name implicitly, the name cannot match the name of any existing table in the database.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
numrows	The maximum number of rows that can be inserted into the diagnostics table when a single statement (for example, INSERT or SET) is executed on the target table. There is no default value for *numrows*. If you do not specify a value for *numrows*, there is no upper limit on the number of rows that can be inserted into the diagnostics table when a single statement is executed on the target table.	You must specify an integer value in the range 1 to the maximum value of the INTEGER data type.	Literal Number, p. 1-997
table name	The name of the target table for which a violations table and diagnostics table are to be created. There is no default value.	If you do not include the USING clause in the statement, the name of the target table must be less than 15 characters. The target table cannot have a violations and diagnostics table associated with it before you execute the statement. The target table cannot be a system catalog table. The target table must be a local table.	Identifier, p. 1-962
violations	The name of the violations table to be associated with the target table. The default name is the name of the target table followed by the characters **_vio**. For further information on the violations table, see "Structure of the Violations Table" on page 1-748.	Whether you specify the name of the violations table explicitly, or the database server generates the name implicitly, the name cannot match the name of any existing table in the database.	Identifier, p. 1-962

(2 of 2)

Usage

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 caused by each row in the violations table.

Relationship of START VIOLATIONS TABLE and SET Statements

The START VIOLATIONS TABLE statement is closely related to the SET statement. If you use the SET statement to set the constraints or unique indexes defined on a table to the filtering object mode, but you do not use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for this target table, any rows that violate a constraint or unique-index requirement during an insert, update, or delete operation are not filtered out to a violations table. Instead you receive an error message indicating that you must start a violations table for the target table.

Similarly, if you use the SET statement to set a disabled constraint or disabled unique index to the enabled or filtering object mode, but you do not use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the table on which the objects are defined, any existing rows in the table that do not satisfy the constraint or unique-index requirement are not filtered out to a violations table. If, in these cases, you want the ability to identify existing rows that do not satisfy the constraint or unique-index requirement, you must issue the START VIOLATIONS TABLE statement to start the violations and diagnostics tables before you issue the SET statement to set the objects to the enabled or filtering object mode.

Starting and Stopping the Violations and Diagnostics Tables

After you use a START VIOLATIONS TABLE statement 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 further information see the STOP VIOLATIONS TABLE statement on page 1-763.

Examples of START VIOLATIONS TABLE Statements

The following examples show different ways to execute the START VIOLATIONS TABLE statement.

Starting Violations and Diagnostics Tables Without Specifying Their 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
```

Starting Violations and Diagnostics Tables and Specifying Their 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
```

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 or SET statement, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX ROWS 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 a START VIOLATIONS TABLE statement 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.

The following table shows the structure of the violations table.

Column Name	Type	Explanation
All columns of the target table, in the same order that they appear in the target table	These columns of the violations table match the data type of the corresponding columns in the target table, except that SERIAL columns in the target table are converted to INTEGER data types in the violations table.	The table definition of the target table is reproduced in the violations table so that rows that violate constraints or unique-index requirements during insert, update, and delete operations can be filtered to 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.
informix_tupleid	SERIAL	This column contains the unique serial identifier that is assigned to the nonconforming row.

informix_optype	CHAR(1)	This column indicates the type of operation that caused this bad row. This column can have the following values:
		I = Insert
		D = Delete
		O = Update (with this row containing the original values)
		N = Update (with this row containing the new values)
		S = SET statement

informix_recowner	CHAR(8)	This column identifies the user who issued the statement that created this bad row.

Relationship Between the Violations and Diagnostics Tables

Users can take advantage of the relationships among the target table, violations table, and diagnostics table to obtain complete diagnostic information about rows that have caused 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. The row in the violations table contains a copy of the row in the target table for which a data-integrity violation was detected. The row in the diagnostics table contains information about the nature of the data-integrity violation caused by the bad row in the violations table. The row in the violations table has a unique serial identifier in the **informix_tupleid** column. The 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 complete diagnostic information about any or all bad rows in the violations table. You can use SELECT statements to 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. However, the database server follows different rules when it grants each type of privilege.

The following table shows the initial set of privileges on the violations table. The **Privilege** column lists the privilege. The **Condition** column explains the conditions under which the database server grants the privilege to a user.

Privilege	Condition
Insert	The user has the Insert privilege on the violations 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	The user has the Delete privilege on the violations 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	The 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. The 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	The 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. The user has the Update privilege on any other column of the violations table if the user has the Update privilege on the same column in the target table.

(1 of 2)

Privilege	Condition
Index	The user has the Index privilege on the violations table if the user has the Index privilege on the target table.
Alter	The Alter privilege is not granted on the violations table. (Users cannot alter violations tables.)
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. However, the database server 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 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 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 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 the user **henry** has been granted the Insert privilege on the target table by both the user **jill** and the user **albert**, the Insert privilege on the violations table is granted to user **henry** both by user **jill** and by user **albert**.
- Once a violations table has been started for a target table, 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.

For example, assume that we have created a table named **cust_subset** and that this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's 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 **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, as follows:

```
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 violations table. She also has the Select privilege on the following columns of the violations table: the **ssn** column, the **lname** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **carrie** has the Insert and Delete privileges on the violations table. She has the Update privilege on the following columns of the violations table: the **city** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column. She has the Select privilege on the following 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 attempt fails, the database server returns an error, and the violations table is not 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 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 existence of the violations and diagnostics tables causes the DROP TABLE statement to fail.
- 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 violations table before or after it uses the violations table during an Insert, Update, Delete, or Set 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 object mode of the constraint or unique index to the enabled or disabled object mode.
  - ❑ You can issue a STOP VIOLATIONS TABLE statement 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.
- 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 blob columns, blobs in the violations table are created in the same blob space as the blobs in the target table.

Example of a Violations Table

To start a violations and diagnostics table for the target table named **customer** in the **stores7** demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer
```

Because your START VIOLATIONS statement does not include a USING clause, the violations table is named **customer_vio** by default. The **customer_vio** table includes the following 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.

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	Explanation
informix_tupleid	INTEGER	This column in the diagnostics table implicitly refers to the values in the **informix_tupleid** column in the violations table. However, this relationship is not declared as a foreign-key to primary-key relationship.

objtype	CHAR(1)	This column identifies the type of the violation. This column can have the following values.
		C = Constraint violation
		I = Unique-index violation

(1 of 2)

Column Name	Type	Explanation
objowner	CHAR(8)	This column identifies the owner of the constraint or index for which an integrity violation was detected.
objname	CHAR(18)	This column contains the name of the constraint or index for which an integrity violation was detected.

(2 of 2)

Initial Privileges on the Diagnostics Table

When the START VIOLATIONS TABLE statement creates the diagnostics table, the database server uses the set of privileges granted on the target table as a basis for granting privileges on the diagnostics table. However, the database server follows different rules when it grants each type of privilege.

The following table shows the initial set of privileges on the diagnostics table. The **Privilege** column lists the privilege. The **Condition** column explains the conditions under which the database server grants the privilege to a user.

Privilege	Condition
Insert	The 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	The 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	The user has the Select privilege on the diagnostics table if the user has the Select privilege on any column in the target table.
Update	The user has the Update privilege on the diagnostics table if the user has the Update privilege on any column in the target table.

(1 of 2)

Privilege	Condition
Index	The user has the Index privilege on the diagnostics table if the user has the Index privilege on the target table.
Alter	The Alter privilege is not granted on the diagnostics table. (Users cannot alter diagnostics tables.)
References	The References privilege is not granted on the diagnostics table. (Users cannot add referential constraints to diagnostics tables.)

(2 of 2)

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. However, the database server 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.
- 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 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 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 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** has been 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 has been 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.

Example of Privileges on the Diagnostics Table

The following example illustrates how the initial set of privileges on a diagnostics table is derived from the current set of privileges on the target table.

For example, assume that there is a table called **cust_subset** and that this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's 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 **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, as follows:

```
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 "Relationship Between the Violations and Diagnostics Tables" on page 1-749.

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 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 object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in the 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 existence of the violations and diagnostics tables causes the DROP TABLE statement to fail.
- 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.

Example of a Diagnostics Table

To start a violations and diagnostics table for the target table named **stock** in the **stores7** 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.

References

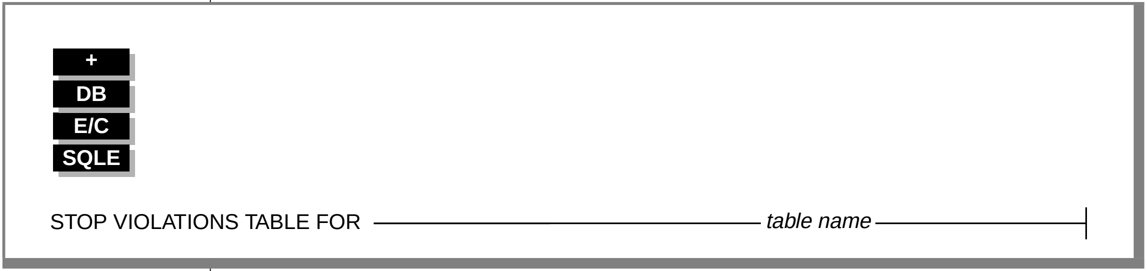
See the STOP VIOLATIONS TABLE and SET statements in this manual.

For information on the system catalog tables that are associated with the START VIOLATIONS TABLE statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

STOP VIOLATIONS TABLE

The STOP VIOLATIONS TABLE statement drops the association between a target table and the special violations and diagnostics tables.

Syntax



Element	Purpose	Restrictions	Syntax
table name	The name of the target table whose association with the violations and diagnostics table is to be dropped. There is no default value.	The target table must have a violations and diagnostics table associated with it before you can execute the statement. The target table must be a local table.	Identifier, p. 1-962

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, and 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.

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.

References

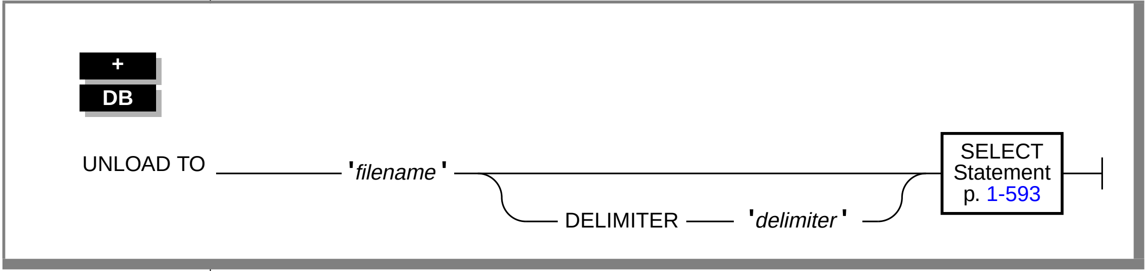
See the SET and START VIOLATIONS TABLE statements in this manual.

For information on the system catalog tables associated with the STOP VIOLATIONS TABLE statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

UNLOAD

Use the UNLOAD statement to write the rows retrieved in a SELECT statement to an operating-system file.

Syntax



Element	Purpose	Restrictions	Syntax
delimiter	A quoted string that identifies the character to use as the delimiter in the output file. The delimiter is a character that separates the data values in each line of the output file. If you do not specify a delimiter character, the database server uses the setting in the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the vertical bar (|).	You cannot use the following items as the delimiter character: backslash (\ ), new-line character (=CTRL-J), hexadecimal numbers (0 to 9, a to f, A to F).	Quoted String, p. 1-1010
filename	A quoted string that specifies the pathname and filename of an ASCII operating-system file. This output file receives the selected rows from the table during the unload operation. The default pathname for the output file is the current directory.	You can unload table data containing VARCHAR or BLOB data types to the output file, but you should be aware of the consequences. See "The UNLOAD TO File" on page 1-766 for further information.	Quoted String, p. 1-1010. The pathname and filename specified in the quoted string must conform to the conventions of your operating system.

Usage

To use the UNLOAD statement, you must have the Select privilege on all columns selected in the SELECT statement. For information on database-level and table-level privileges, see the GRANT statement on page 1-458.

The SELECT statement can consist of a literal SELECT statement or the name of a character variable that contains a SELECT statement. (See the SELECT statement on page 1-593.)

The UNLOAD TO File

The UNLOAD TO file contains the selected rows retrieved from the table. You can use the UNLOAD TO file as the LOAD FROM file in a LOAD statement.

The following table shows types of data and their output format for an UNLOAD statement in DB-Access (when DB-Access uses the default locale, U.S. English).

Data Type	Output Format
boolean	BOOLEAN data is represented as a `'t'` for a TRUE value and an `'f'` for a FALSE value.
character	If a character field contains the delimiter character, 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.) Trailing blanks are automatically clipped.
collections	A collection is unloaded with its values surrounded by braces ({}) and a field delimiter separating each element. For more information, see "Unloading Complex Types" on page 1-771.
date	DATE values are represented as *mm*/*dd*/*yyyy*, 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. See the *Guide to GLS Functionality* for more information about these environment variables.

(1 of 3)

Data Type	Output Format
MONEY	MONEY values are unloaded with no leading currency symbol. They use the comma (,) as the thousands separator and the period as the decimal separator. If you have set the **DBMONEY** environment variable, the UNLOAD statement uses the specified currency format for MONEY values. See the *Guide to GLS Functionality* for more information about this environment variable.
NULL	NULL columns are unloaded by placing no characters between the delimiters.
number	Number data types are displayed with no leading blanks. INT8, INTEGER or SMALLINT zero are represented as 0, and FLOAT, SMALLFLOAT, DECIMAL, or MONEY zero are represented as 0.00.
row types (named and unnamed)	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 1-771.
simple large objects (TEXT, BYTE)	TEXT and BYTE columns are unloaded directly into the UNLOAD TO file. For more information, see "Unloading Simple Large Objects" on page 1-769.
smart large objects (CLOB, BLOB)	CLOB and BLOB columns are unloaded into a separate operating-system file on the client computer. The field for the CLOB or BLOB column in the UNLOAD TO file contains the name of this separate file. For more information, see "Unloading Smart Large Objects" on page 1-769.

(2 of 3)

Data Type	Output Format
time	DATETIME and INTERVAL values are represented in character form, showing only their field digits and delimiters. No type specification or qualifiers are included in the output. The following pattern is used: *yyyy-mm-dd hh:mi:ss.fff*, omitting fields that are not part of the data. If you have set the **GL_DATETIME** or **DBTIME** environment variable, the UNLOAD statement uses the specified format for DATETIME values. See the *Guide to GLS Functionality* for more information on these environment variables.
user-defined data formats (opaque types)	The associated opaque type must have an export support function defined if special processing is required to copy the data in the internal format of the opaque type to the format in the UNLOAD TO file. An exportbinary support function might also be required if the data is in binary format. The data in the UNLOAD TO file would correspond to the format that the export or exportbinary support function returns.

(3 of 3)

GLS

If you are using a nondefault locale, the formats of DATE, DATETIME, MONEY, and numeric column values in the UNLOAD TO file are determined by the formats that the locale supports for these data types. For more information, see the *Guide to GLS Functionality*. ♦

The following statement unloads rows from the **customer** table where the value of **customer_num** is greater than or equal to 138, and puts them in a file named **cust_file**:

```
UNLOAD TO 'cust_file' DELIMITER '!'
    SELECT * FROM customer WHERE customer_num> = 138
```

The output file, **cust_file**, appears as shown in the following example:

```
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
```

If you are unloading columns of the VARCHAR data type, the database server does not retain trailing blanks.

Unloading Simple Large Objects

The database server unloads simple large objects (BYTE and TEXT columns) directly into the UNLOAD TO file. BYTE data are written in hexadecimal dump format with no added spaces or new lines. Consequently, the logical length of an unloaded file that contains BYTE items can be very long and very difficult to print or edit.

GLS

For TEXT columns, the database server handles any required code-set conversions for the data. For more information, see the *Guide to GLS Functionality*. ♦

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 *Informix Guide to SQL: Reference*.

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. It creates this file in the same directory as the UNLOAD TO file. The filename of this file has one of the following formats:

- For a BLOB value:

  ```
  blob#########
  ```

- For a CLOB value:

  ```
  clob#########
  ```

In the preceding formats, the # 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; however must smart large objects would have an identifier with significantly fewer digits.

When the database server unloads the first smart large object, it creates the appropriate BLOB or CLOB file with the hexadecimal identifier of the smart large object. It appends any additional BLOB or CLOB values to the appropriate file until the file size reaches a limit of 2 gigabytes. If there are still additional smart large-object values, the database server creates another BLOB or CLOB file whose filename contains the hexadecimal identifier of the next smart large object to unload.

Each BLOB or CLOB value is appended to the appropriate file. The database server might create several files if the values are extremely large or there any many values.

In an UNLOAD TO file, a BLOB or CLOB column value appears as follows:

```
start_off, end_off, client_path
```

In this format, *start_off* is the starting offset of the smart large-object value within the file, *end_off* is the length of the BLOB or CLOB value, and *client_path* is the pathname for the client file.

For example, suppose the database server unloads the following CLOB values.

CLOB Value	Size
1	2048
2	4096
3	1024
4	1024
5	2048

If the first CLOB value has a hexadecimal identifier of **203b2**, the database server creates the **clob203b2** file to hold this first value. It then unloads the next four CLOB values to the **clob203b2** file. The corresponding fields in the UNLOAD TO file appears as follows:

```
|0, 2048, /usr/apps/clob203b2|
|2049, 6145, /usr/apps/clob203b2|
|6146, 7170, /usr/apps/clob203b2|
|7171, 8195, /usr/apps/clob203b2|
|8196, 10244, /usr/apps/clob203b2|
```

If the database server unloaded additional CLOB values, it would store these values in the **clob203b2** file until this file reached a size of 2 gigabytes. It would then create a new file of the form **clob**########, with the # symbols replaced by the hexadecimal identifier of the first CLOB value in the file.

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) have their fields enclosed with parentheses and separated with the field separator, as follows:

  ```
  ( val1 | val2 | ... )
  ```

  For example, to unload the ROW values (1, 'abc'), the corresponding field of the UNLOAD TO file appears as follows:

  ```
  |(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 vertical bar (|). See Chapter 3 of the *Informix Guide to SQL: Reference* for information about setting the **DBDELIMITER** environment variable.

You can specify the TAB (= CTRL-I) or <blank> (= ASCII 32) as the delimiter symbol. You cannot use the following as the delimiter symbol:

- Backslash (\)
- New-line character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

Do not use the backslash (\) as a field separator or UNLOAD delimiter. It serves as an escape character to inform the UNLOAD statement that the next character is to be interpreted as part of the data.

The following statement specifies the semicolon (;) as the delimiter character:

```
UNLOAD TO 'cust.out' DELIMITER ';'
    SELECT fname, lname, company, city
        FROM customer
```

References

See the LOAD and SELECT statements in this manual.

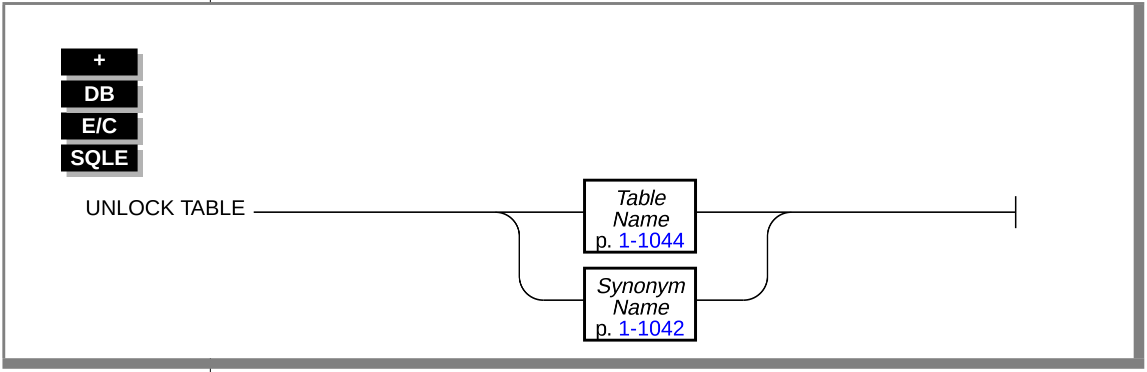
In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the UNLOAD statement.

In the *Informix Migration Guide*, see the task-oriented discussion of the UNLOAD statement and other utilities for moving data.

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.

Syntax



Usage

You can lock a table if you own the table or if you have the Select privileges on the table, either from a direct grant 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.

The *table name* either is the name of the table you are unlocking or a synonym for the table. Do not specify a view or a synonym of 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 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

You should not issue an UNLOCK TABLE statement within an ANSI-compliant database. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database. ♦

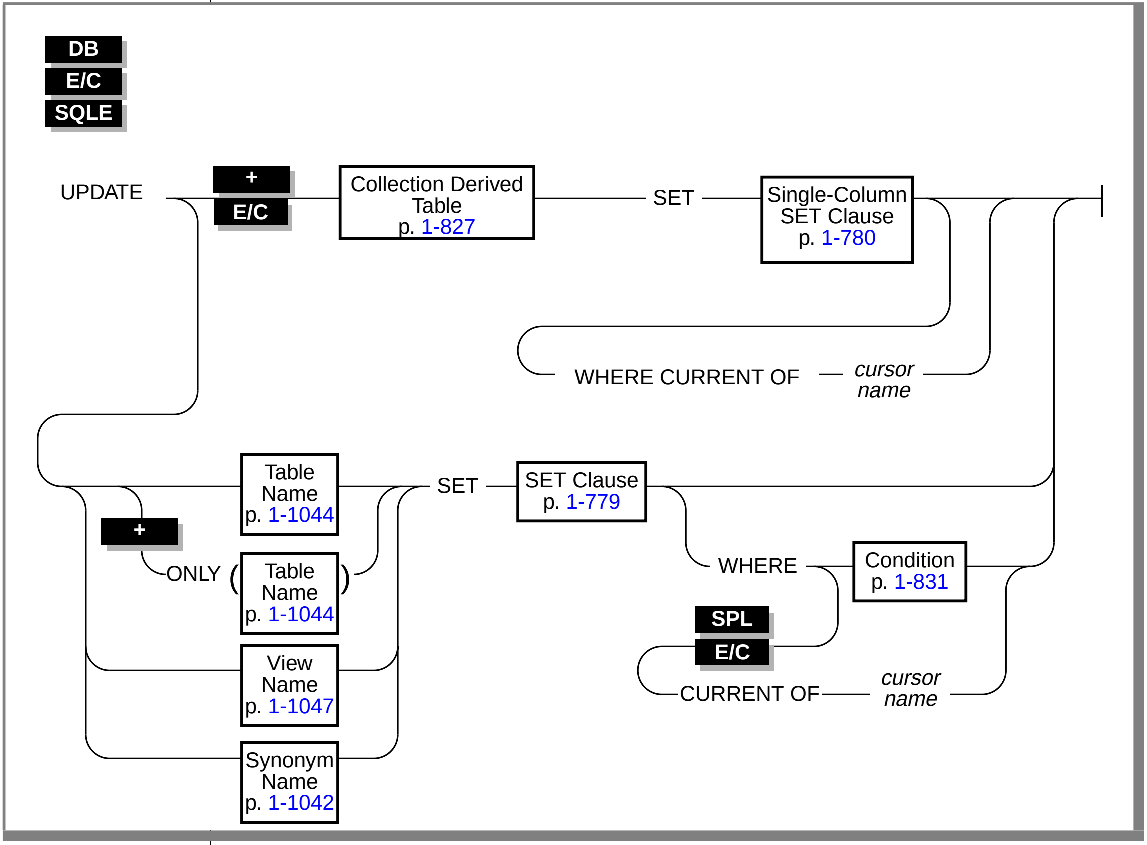
References

See the COMMIT WORK, ROLLBACK WORK, and LOCK TABLE statements in this manual.

UPDATE

Use the UPDATE statement to change the values in one or more columns or one or more elements in an SPL or INFORMIX-ESQL/C collection variable.

Syntax



Element	Purpose	Restrictions	Syntax
derived column	An alias for the column name used in a SET clause to update a collection.	You can only specify a derived column if the collection being updated is a collection of opaque, distinct, built-in, or collection data types. You cannot specify a derived column for collections of named or unnamed row types.	Identifier, p. 1-962
cursor name	The name of the cursor to be used by the UPDATE statement. The current row of the active set for this cursor is updated when the UPDATE statement is executed. See "WHERE CURRENT OF Clause" on page 1-789 for more information on this parameter.	You cannot update a row with a cursor if that row includes aggregates. The specified cursor (as defined in the SELECT...FOR UPDATE portion of a DECLARE statement) can contain only column names. If the cursor was created without specifying particular columns for updating, you can update any column in a subsequent UPDATE...WHERE CURRENT OF statement. But if the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you can update only those columns in a subsequent UPDATE...WHERE CURRENT OF statement.	Identifier, p. 1-962

Usage

Use the UPDATE statement to update any of the following types of objects:

E/C

SPL

- A row in a table: a single row, a group of rows, or all rows in a table
- An element in a collection variable ♦
- A ESQL/C **row** variable: a field or all fields ♦

For information on how to update elements of a collection variable, see "Updating a Collection Variable" on page 1-793. The other sections of this UPDATE statement describe how to update a row in a table.

To update data in a table, you must either own the table or have the Update privilege for the table (see the GRANT statement on page 1-458). 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".

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.

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. However, if the statement is in a command file, and you are running from the command line, the statement executes immediately. ♦

Updating Rows Through a View

You can update data through a *single-table* view if you have the Update privilege on the view (see the GRANT statement on page 1-458). To do this, the defining SELECT statement can select from only one table, and it cannot contain any of the following elements:

- DISTINCT keyword
- GROUP BY clause
- Derived value (also called a virtual column)
- Aggregate value

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 further information, refer to the WITH CHECK OPTION discussion in the CREATE VIEW statement on page 1-286.

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: *You cannot update rows to a remote table through views with check options.*

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.

ANSI

If you are updating rows 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 the ROLLBACK WORK statement to undo the update.

When you use INFORMIX-Universal Server, you are within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

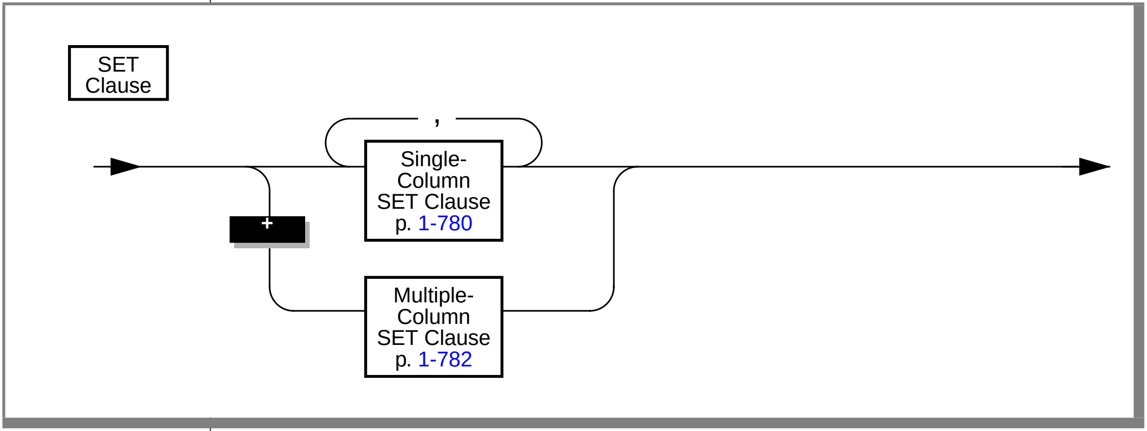
Locking Considerations

When Universal Server selects a row 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 do not let those processes 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.

Universal Server allows only one update lock at a time on a row or a page (the type of lock depends on the lock mode that is selected in the CREATE TABLE or ALTER TABLE statements). An update process can acquire an update lock on a row or 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 affected by a single update is very 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.

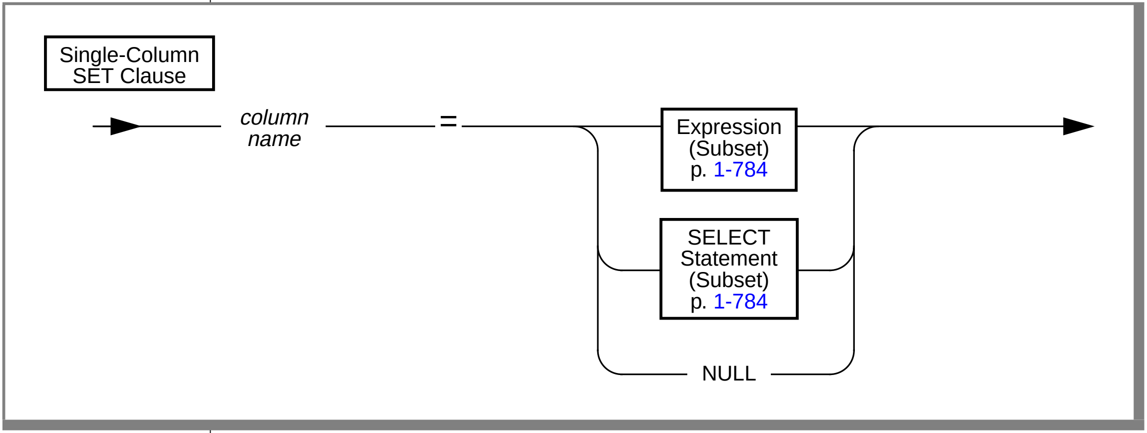
SET Clause


SET Clause
,
Single-Column SET Clause p. 1-780
+
Multiple-Column SET Clause p. 1-782

The SET clause identifies the columns to be updated and assigns values to each column. The clause supports the following formats:

- A single-column SET clause, which pairs a single column to a single expression
- A multiple-column SET clause, which lists multiple columns and sets them equal to corresponding expressions.

Single-Column SET Clause


Single-Column SET Clause
column name
=
Expression (Subset) p. 1-784
SELECT Statement (Subset) p. 1-784
NULL

Element	Purpose	Restrictions	Syntax
column name	The name of the column that you want to update	You cannot update SERIAL or SERIAL8 columns. 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 as long as the number of columns named in the column list equals the number of values that the expressions in the expression list produce.	Identifier, p. 1-962

You can use a SET clause to set a single column to a single expression. A single column in a SET clause can be a named row type column or an unnamed row type column. When you use UPDATE to update a database column, you can include any number of single-column to single-expressions in the UPDATE statement. The following examples illustrate the single-column to single-expression form of the SET clause:

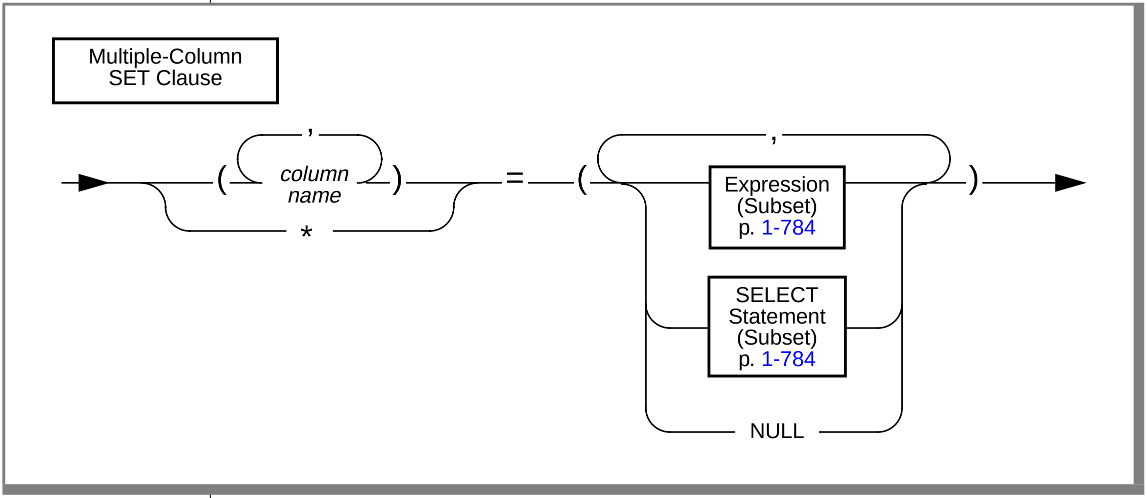
```
UPDATE customer
    SET address1 = '1111 Alder Court',
        city = 'Palo Alto',
        zipcode = '94301'
    WHERE customer_num = 103

UPDATE orders
    SET ship_charge =
        (SELECT SUM(total_price) * .07
            FROM items
            WHERE orders.order_num = items.order_num)
        WHERE orders.order_num = 1001

UPDATE stock
    SET unit_price = unit_price * 1.07

UPDATE  empinfo
    SET name = ROW('dennis', 'banks')
        WHERE emp_id = 322
```

For more information on the column values that are valid in a SET clause, see "SET-Clause Values" on page 1-784. For more information on how to specify values of a row column in a SET clause, see "Updating Row-Type Columns" on page 1-785.

E/C

If you use UPDATE to update a **collection** variable (an UPDATE with a Collection Derived Table segment), you can include only one pair of *column name* and *column value*. Furthermore, you cannot include complex expressions as the column values. Column values are restricted to literal values or collection variables. For more information, see "Updating a Collection Variable" on page 1-793. ♦

Multiple-Column SET Clause


Multiple-Column SET Clause
(
,
column name
)
*
=
(
,
Expression (Subset) p. 1-784
SELECT Statement (Subset) p. 1-784
NULL
)

Element	Purpose	Restrictions	Syntax
*	Indicator that all columns in the specified table or view are to be updated	The restrictions that are discussed under *column name* also apply to the asterisk (*).	The asterisk (*) is a literal value with a special meaning in this statement.
column name	The name of the column that you want to update	You cannot update SERIAL or SERIAL8 columns. You cannot use this syntax to update a row column. For more information, see page 1-785. An expression list can include an SQL subquery that returns a single row of multiple values as long as the number of columns named in the column list equals the number of values that the expressions in the expression list produce.	Identifier, p. 1-962

You can use a second format of the SET clause to set multiple columns to multiple expression. The SET clause offers the following options for listing a series of columns you intend to update:

- Explicitly list each column, placing commas between columns and enclosing the set of columns in parentheses.
- Implicitly list all columns in *table name* using the asterisk notation (*).

To complete the multiple-column SET clause, you must list each expression explicitly, placing commas between expressions and enclosing the set of expressions in parentheses. The number of columns in the column list must be equal to the number of expressions in the expression list, unless the expression list includes an SQL subquery. The following examples illustrate this form of the multiple-column SET clause:

```
UPDATE customer
    SET (fname, lname) = ('John', 'Doe')
    WHERE customer_num = 101


UPDATE manufact
    SET * = ('HNT', 'Hunter')
    WHERE manu_code = 'ANZ'
```

An expression list can include an SQL subquery that returns a single row of multiple values as long as the number of columns named, 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 SQL subqueries in a multiple-column SET clause:

```
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),
            '07/01/1993')
    WHERE col4 = 1001
```

For more information on the column values that are valid in a SET clause, see "SET-Clause Values" on page 1-784.

SET-Clause Values

You can express a value in a single-column or multiple-column SET clause in any of following ways:

- As an expression
- As a SELECT statement
- As a NULL value

Subset of Expressions Allowed in the SET Clause

When you update a table or view, you cannot use an expression comprised of aggregate functions in the SET clause. For a complete description of syntax and usage, see the Expression segment on page 1-876.

Subset of SELECT Statements Allowed in the SET Clause

A SELECT statement used in a SET clause can return more than *one column* of information in a row. However, the SELECT statement cannot return more than *one row* of information in a table. For a complete description of syntax and usage, refer to the SELECT statement on page 1-593.

Updating a Column to NULL

You can use the NULL keyword to modify a column value when you use the UPDATE statement. 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',
    address2 = null,
    city = 'Palo Alto',
    zipcode = '94303'
    WHERE customer_num = 134
```

Updating Row-Type Columns

You use the SET clause to update a named row type 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 1-895 in the Expression segment. See also the Literal Row segment on page 1-999.

ESQL

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 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 1-798. ♦

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 1-798. ♦

To update only some of the fields in a row, you can perform one of the following operations:

- 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.

ESQL

- Select the row into a row variable and update the desired fields.

  For more information, see "Updating a Row Variable" on page 1-798. ♦

Updating Collection Columns

You can use the SET clause to insert literal values into a collection column, which can be a LIST, MULTISET, or SET. 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 the **tab1** table with literal values:

```
EXEC SQL update tab1
    set list1 = "LIST{ROW(4, 'opqrs'),
                ROW(5, 'tuvwx'),
                ROW(6, 'yxabc')}"
    where int1 = 5
```

The collection column, **list1**, in the **tab1** row has three elements, and each element is an unnamed row type with an INTEGER field and a CHAR(5) field. For more information on the syntax for literal collection values, see "Literal DATETIME" on page 1-991.

E/C

SPL

You can use an ESQL/C **collection** variable or an SPL collection variable to update:

- an entire collection into a column.

  Use a collection variable as a *variable name* in the SET clause to insert an entire collection.

  For example, the following ESQL/C code fragment updates with the elements of the **a_set** host variable the **set_col** column of the row in the **tab_a** table whose **int_col** value is 6:

  ```
  EXEC SQL BEGIN DECLARE SECTION;
     client collection set(smallint not null) a_set;
  EXEC SQL END DECLARE SECTION;
  ...
  EXEC SQL update tab_a set set_col = :a_set
     where int_col = 6;
  ```

- individual elements in a collection.

  To update a collection column with non-literal values, you must first update the elements in a collection variable and then specify the **collection** variable in the SET clause of an UPDATE statement. For information on how to update elements of a **collection** variable, see "Updating a Collection Variable" on page 1-793. ♦

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 very large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign()**. When you execute the UPDATE statement 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 the *Extending INFORMIX-Universal Server: Data Types* manual.

WHERE Clause

The WHERE clause lets you limit the rows that you want to update. If you omit the WHERE clause, every row in the table is updated.

The WHERE clause consists of a standard search condition. (For more information, see the SELECT statement on page 1-593). The following example illustrates a WHERE condition within an UPDATE statement. In this example, the statement updates 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)
```

Updating and the WHERE Clause

ANSI

When you update a table in an ANSI-compliant database using an UPDATE statement with the WHERE clause, and the database server finds no matching rows, 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. ♦

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). However, if the UPDATE ... WHERE ... is a part of a multistatement prepare, and no rows are returned, the database server does issue a warning. It sets SQLSTATE to '02000' and **SQLCODE** value to 100.

For additional information about the **SQLSTATE** code, see the GET DIAGNOSTICS statement in this manual. For information about the SQLCODE code, see the description of the **sqlca** structure in the *Informix Guide to SQL: Tutorial*.

WHERE CURRENT OF Clause

ESQL

SPL

You can use the WHERE CURRENT OF clause to update either of the following objects:

- The current row of the active set of a cursor
- The current element of a collection cursor (INFORMIX-ESQL/C only)

You access both of these objects with an update cursor. An update cursor is a sequential cursor that is associated with a SELECT statement but can modify and delete the contents of the cursor. For more information on the update cursor, see page 1-307. ♦

ESQL

To use the WHERE CURRENT OF clause, you must have previously used the DECLARE statement with the FOR UPDATE clause to define the *cursor name* for the update cursor. (See the DECLARE statement on page 1-300.) ♦

SPL

Before you can use the WHERE CURRENT OF clause, you must declare a cursor with the FOREACH statement. (See the FOREACH statement on page 2-27.) ♦

ANSI

All select cursors are potentially update cursors in ANSI-compliant databases. You can use the WHERE CURRENT OF clause with any select cursor. ♦

Tip: *You can use an update cursor to perform updates that are not possible with the UPDATE statement.*

Updating the Current Row

ESQL

SPL

When you specify a table or view name in the FROM clause of the SELECT, the DECLARE statement defines a cursor that populates an active set with the rows of the specified tables or views. The UPDATE...WHERE CURRENT OF statement updates values in the current row of the active set. However, you cannot update a row with a cursor if that row includes aggregates. The cursor named in the WHERE CURRENT OF clause can only contain column names. The UPDATE statement does not advance the cursor to the next row, so the current row position remains unchanged.

To use the WHERE CURRENT OF clause, you must have previously used the DECLARE statement with the FOR UPDATE clause to define the *cursor name* for the update cursor. (See the DECLARE statement on page 1-300.) If you created the cursor without specifying any columns for updating, you can update any column in a subsequent UPDATE...WHERE CURRENT OF statement.

You can restrict the effect of the WHERE CURRENT OF clause if the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause. In this case, 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. Universal Server can usually perform updates more quickly if columns are specified in the DECLARE statement.

The following INFORMIX-ESQL/C example illustrates the WHERE CURRENT OF clause of the UPDATE statement. 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. (For more information, see the PREPARE statement on page 1-538.)

```
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 'stores7';
    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;
}
```

♦

Updating A Collection Element

ESQL

SPL

You declare a collection cursor when you associate a cursor with SELECT statement that includes a Collection Derived Table clause. You use one of the following statements to declare a collection cursor:

- In an ESQL/C program, use the DECLARE statement.

  For more information, see "Associating a Cursor With a Collection Variable" on page 1-317 in the DECLARE statement.
- In an SPL routine, use the FOREACH statement.

  For more information, see the FOREACH statement on page 2-27.

A collection cursor is an update cursor by default. However, you can optionally specify the FOR UPDATE clause with the SELECT statement. With an update cursor, you can use the UPDATE...WHERE CURRENT OF statement to update the current element of a collection cursor. For more information, see "Updating a Collection Variable" on page 1-793.

Important: *You can only declare a select cursor on a collection variable. Neither INFORMIX-ESQL/C nor SPL supports cursors on row variables. For more information, see "Updating a Row Variable" on page 1-798.*

♦

Updating a Collection Variable

The UPDATE statement with the Collection Derived Table segment allows you to update elements in a collection variable. The Collection Derived Table segment identifies the collection variable in which to update the elements. For more information on the Collection Derived Table segment, see page 1-827.

E/C

In an INFORMIX-ESQL/C program, declare a host variable of type **collection** for a collection variable. This **collection** variable can be typed or untyped. ♦

SPL

In an SPL routine, declare a variable of type COLLECTION, LIST, MULTISET, or SET for a collection variable. This collection variable can be typed or untyped. ♦

To update elements, follow these steps:

1. Create a collection variable in your SPL routine or ESQL/C program.
2. Optionally, select a collection column into the collection variable with the SELECT statement (without the Collection Derived Table segment).
3. Update elements of the collection variable with the UPDATE statement and the Collection Derived Table segment.
4. Once the collection variable contains the correct elements, you then use the UPDATE or INSERT statement on a table or view name to save the collection variable in the collection column (SET, MULTISET, or LIST).

The UPDATE statement and the Collection Derived Table segment allow you to perform the following operations on a collection variable:

- Update a *particular* element in the collection

  You must declare an update cursor for the collection variable and use UPDATE with the WHERE CURRENT OF clause. For more information on how to use an update cursor with ESQL/C, see the DECLARE statement. For more information on how to use an update cursor with SPL, see "FOREACH" on page 2-27.

  The application or SPL routine must position the update cursor on the element to be updated and then use UPDATE...WHERE CURRENT OF to update this value. For more information on the WHERE CURRENT OF clause of UPDATE, see page 1-789.

- Update *all* elements in the collection to the same value

  Use the UPDATE statement (without the WHERE CURRENT OF clause) and specify a derived column name in the SET clause.

E/C

For example, the following UPDATE changes all elements in the **a_list** ESQL/C **collection** variable to a value of 16:

```
EXEC SQL update table(:a_list) (list_elmt)
      set list_elmt = 16;
```

♦

SPL

The following UPDATE changes all elements in **a_list** to the value 16 in an SPL routine:

```
UPDATE TABLE (a_list) (list_elmt)
      SET list_elmt = 16;
```

♦

In these examples, the derived column **list_elmt** provides an alias to identify an element of the collection. No update cursor is required to update all elements of a collection.

An UPDATE of an element or elements in a collection variable cannot include a WHERE clause. When you use UPDATE to update a collection variable, you can only include one pair of column-name and column-value. Furthermore, you cannot include complex expressions as the column values. Column values are restricted to literal values or collection variables.

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 collection column with one of the following SQL statements:

- To update the collection column in the table with the collection variable, use an UPDATE statement on a table or view name and specify the collection variable in the SET clause.

  For more information, see "Updating Collection Columns" on page 1-786.

- To insert a collection in a column, use the INSERT statement on a table or view name and specify the collection variable in the VALUES clause.

  For more information, see "Inserting Into a Collection Variable" on page 1-506 in the INSERT statement.

E/C

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 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 *INFORMIX-ESQL/C Programmer's Manual*. ♦

SPL

The following SPL routine performs the same task as preceding ESQL/C program.

```
CREATE PROCEDURE test5()

    DEFINE a SMALLINT;
    DEFINE b SET(SMALLINT NOT NULL);

    SELECT set_col INTO b FROM table1
        WHERE id = 6;
        -- select the collection from the database
        -- into a collection variable

    FOREACH cursor1 FOR
        SELECT * INTO a FROM TABLE(b);
            -- select one element at a time
            -- from the collection derived table b
        IF a = 4 THEN
            UPDATE TABLE(b)(x)
                SET x = 10
                WHERE CURRENT OF cursor1;
            EXIT FOREACH;
            -- if the element has the value 4
            -- update it to have the value 10 and exit
        END IF;
    END FOREACH;

    UPDATE table1 SET set_col = b;
        WHERE id = 6;
        -- update the base table with the new collection

END PROCEDURE;
```

After you execute this SPL routine, the SET stored in **table1** will contain the values {1,8,10,5,2}. This SPL routine defines two collection variables, **a** and **b**, and selects a SET from **table1** into **b**. For more information on how to use SPL collection variables, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. ♦

You can also use a collection variable as a *variable name* in the SET clause to update all elements of a collection. For more information, see "Updating Collection Columns" on page 1-786.

Updating a Row Variable

E/C

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 on the Collection Derived Table segment, see page 1-827.

To update fields, follow these steps:

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. Once 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. You specify the new field value(s) in the SET clause. For example, the following UPDATE changes the **x** and **y** fields in **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, **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. However, you can use the UPDATE statement and the Collection Derived Table segment to insert new field values into a **row** host variable, as long as you specify a value for every field in the row. For example, the following code fragment inserts new field values into the **myrect** row variable 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 a field or fields in a row variable cannot include a WHERE clause.

The row variable stores the fields of the row. However, 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 1-785 in the UPDATE statement.

- To insert a row in 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 1-502.

For more information on how to use SPL row variables, see Chapter 14 in the *Informix Guide to SQL: Tutorial*. For more information on how to use ESQL/C **row** variables, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*. ♦

References

See the DECLARE, INSERT, OPEN, and SELECT statements in Chapter 1 of this manual. See also the FOREACH statement in Chapter 2 of this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the UPDATE statement in Chapter 6 and Chapter 12. In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the UPDATE statement.

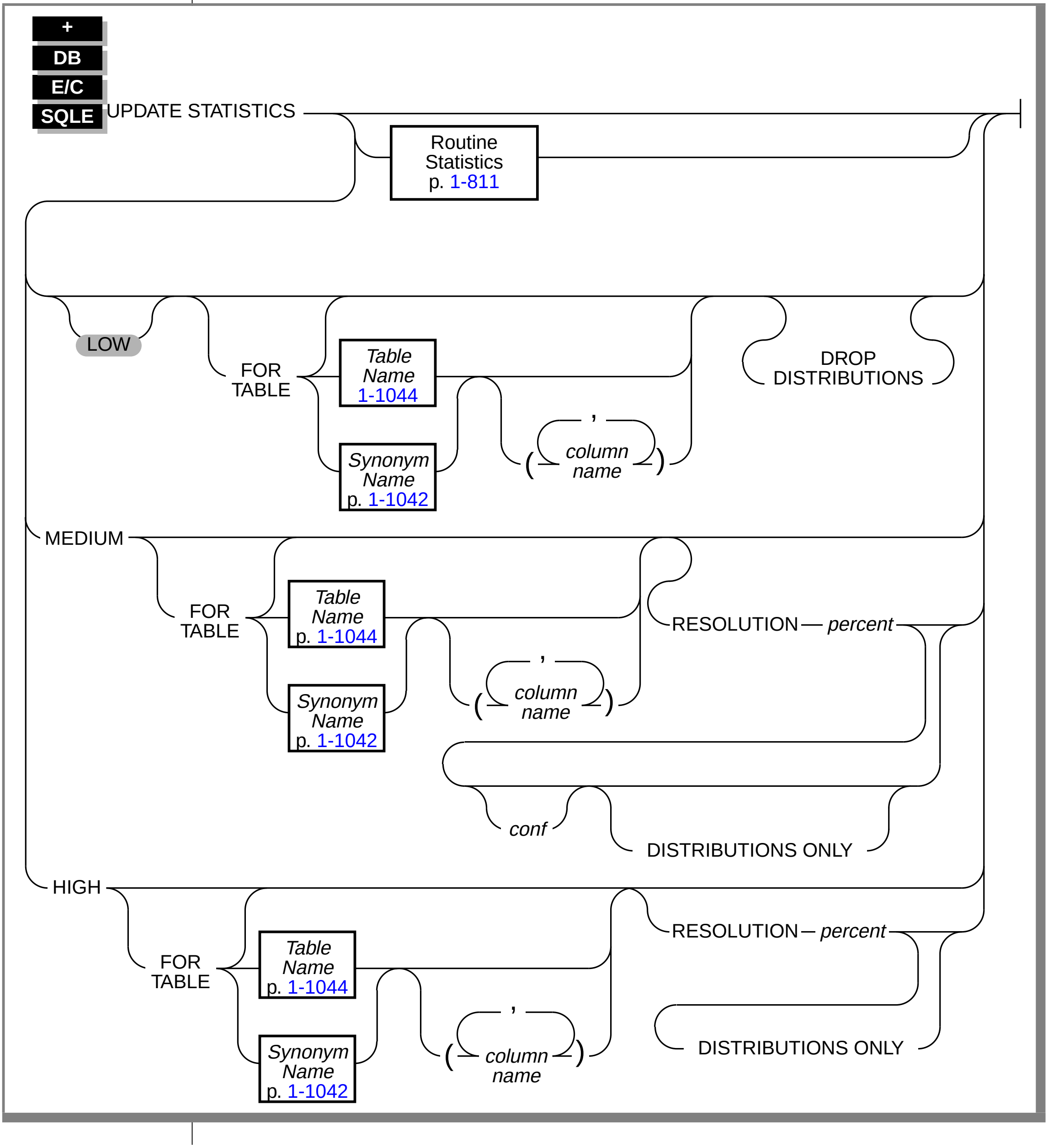
For information on how to access row and collections with ESQL/C host variables, see the discussion of complex data types in the *INFORMIX-ESQL/C Programmer's Manual*.

UPDATE STATISTICS

Use the UPDATE STATISTICS statement to:

- determine the distribution of column values.
- update the system catalog tables that the server uses to optimize queries.
- force reoptimization of stored procedures.
- convert existing table indexes when you upgrade the database server.

Syntax


+
DB
E/C
SQLE
UPDATE STATISTICS
Routine Statistics p. 1-811
LOW
FOR TABLE
Table Name 1-1044
Synonym Name p. 1-1042
(
,
column name
)
DROP DISTRIBUTIONS
MEDIUM
FOR TABLE
Table Name p. 1-1044
Synonym Name p. 1-1042
(
,
column name
)
RESOLUTION
percent
conf
DISTRIBUTIONS ONLY
HIGH
FOR TABLE
Table Name p. 1-1044
Synonym Name p. 1-1042
(
,
column name
)
RESOLUTION
percent
DISTRIBUTIONS ONLY

Element	Purpose	Restrictions	Syntax
column name	The name of a column in the specified table	The column must exist. You cannot use the name of a BYTE or TEXT column with the MEDIUM or HIGH keywords.	Identifier, p. 1-962
conf	A measure of confidence in the accuracy of medium distribution data. Confidence level is expressed as the proportion of values obtained with the MEDIUM keyword that you project you would also obtain with the HIGH keyword. The default confidence level is `0.95`.	Valid values range from the minimum confidence level of `0.80` to the maximum value of `0.99`.	Literal Number, p. 1-997
percent	The percentage of samples in each bin of a distribution, or bin resolution. With the MEDIUM keyword, the default value of *percent* is `2.5`. With the HIGH keyword, the default value of *percent* is `0.5`. For further information on this parameter, see "Using the MEDIUM or HIGH Keyword" on page 1-808.	The minimum resolution possible for a table is 1 / *nrows*, where *nrows* is the number of rows in the table.	Literal Number, p. 1-997

Usage

Use the UPDATE STATISTICS statement to distribute data values in table columns and to optimize execution plans for routines. The information produced by the UPDATE STATISTICS statement is restricted to objects in the database from which you execute the statement.

Tables and Columns

The database server evaluates statistics to determine the optimal execution plan for queries

The UPDATE STATISTICS statement stores statistics in the **systables**, **syscolumns**, **sysindexes**, and **sysdistrib** system catalog tables. These stored statistics describe the distribution of data values in tables, columns, and indexes.

The server does not automatically update **systables**, **syscolumns**, **sysindexes**, and **sysdistrib** if a change to the database obsoletes the corresponding statistics in these system catalog tables. Issue an UPDATE STATISTICS statement to ensure that the stored distribution information reflects the state of the database.

Routines

The **sysprocplan** system catalog table stores execution plans for routines. Two actions update the **sysprocplan** system catalog table:

- Execution of a routine that uses a modified table
- The UPDATE STATISTICS statement

If you change an object, such as a table, you can run UPDATE STATISTICS to reoptimize on demand, rather than waiting until the routine next executes.

When to Update Statistics

Update the statistics in the following situations:

- You perform extensive modifications to a table.
- You change a table referenced by a routine, if you want the server to immediately reoptimize rather than wait for execution time.
- An application changes the distribution of column values. UPDATE STATISTICS 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 *Informix Migration Guide*.

Updating Statistics for Tables

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. To prepare for the optimizer, the UPDATE STATISTICS statement distributes cell values from a one column into ranges, each of which contains an equal portion of the column data.

A *distribution* is a mapping of the data in a column into a set of column values. The contents of the column are divided into bins, each of which represents a percentage of data. For example, if one bin holds 2 percent of the data, 50 of these 2-percent bins hold all the data. A bin contains the particular range of data values that reflects the appropriate percentage of entries in the column.

Using the FOR TABLE Keywords

Without a FOR TABLE clause, UPDATE STATISTICS updates data for every table in the current database, including the system tables.

Use FOR TABLE to exclude statistics on system tables.

The FOR TABLE clause without a table name updates the statistics for all tables, including temporary tables, in the current database.

Specify a table name or synonym name to update statistics for only that table. You can explicitly update the statistics for a temporary table or build distributions for a temporary table by specifying the name of the table.

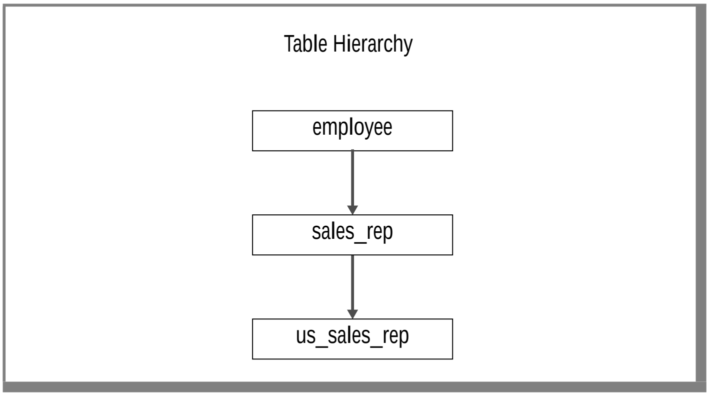
To narrow the scope of UPDATE STATISTICS further, specify column names.

Important: *You cannot create distributions for TEXT or BYTE columns. If you include a TEXT or BYTE column in UPDATE STATISTICS (MEDIUM) or UPDATE STATISTICS (HIGH), the statement does not return an error or create distributions for those columns, but it does construct distributions for other columns in the list.*

Using the FOR TABLE ONLY Keywords

If the table specified in the UPDATE STATISTICS ON TABLE statement 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 1-2, 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 1-2



To update statistics on *both* tables **sales_rep** and **us_sales_rep**, you would use the following statement:

```
UPDATE STATISTICS ON TABLE sales_rep
```

Use the ONLY keyword to collect data for one table in a hierarchy of typed tables. The following example creates a distribution of data for each column in table **sales_rep** but does not act on tables **employee** or **us_sales_rep**:

```
UPDATE STATISTICS ON TABLE ONLY sales_rep
```

Tip: *The more specific you make the list of objects that UPDATE STATISTICS examines, the faster it completes execution. For help in deciding what is needed, see "When to Update Statistics" on page 1-805.*

Use the FOR TABLE and ONLY keywords to specify that you want statistics updated for only some of the tables and columns in the database. Use the LOW, MEDIUM, or HIGH keyword to indicate the precision of every distribution that a single UPDATE STATISTICS statement creates. Limiting the number of columns distributed speeds the update. Similarly, precision effects the speed of the update. If all other keywords are the same, LOW works fastest, but HIGH examines the most data.

Using the LOW Keyword

To create a low distribution, use the LOW keyword or issue the UPDATE STATISTICS statement without a distribution level keyword. A low distribution update does the following:

- Updates data in the **systables**, **syscolumns**, and **sysindexes** tables
- Adds no new information to the **sysdistrib** system catalog table

  Normally a low distribution update does not remove data from the **sysdistrib** system catalog table. You can optionally remove it with the DROP DISTRIBUTIONS option. "Using LOW with DROP DISTRIBUTIONS" on page 1-808 explains how this option works.
- Updates table, row, and page counts, as well as index and column statistics for specified columns

Tip: *If you want the UPDATE STATISTICS statement to do minimal work, specify a column that is not part of an index.*

The following example updates statistics on the **customer_num** column of the **customer** table. All distributions associated with the **customer** table remain intact, even those that already exist on the **customer_num** column.

```
UPDATE STATISTICS LOW FOR TABLE customer (customer_num)
```

Using LOW with DROP DISTRIBUTIONS

The DROP DISTRIBUTIONS keywords force the removal of distribution information from the **sysdistrib** system catalog table to accompany the construction of a low distribution.

When you issue the statement with a table name but no column names, all the distributions for the table name are dropped. When you specify column names in the UPDATE STATISTICS, only the distribution data for those columns is dropped from **sysdistrib**.

You must have the DBA privilege or be the owner of the table in order to drop distributions.

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
```

Using the MEDIUM or HIGH Keyword

A medium or high distribution update does the following:

- Updates data in the **systables**, **syscolumns**, and **sysindexes** tables
- Updates distribution data in the **sysdistrib** system catalog table
- Updates table, row, and page counts, as well as index and column statistics for specified columns

You must have the DBA privilege or be the owner of the table in order to create high or medium distributions.

Both the MEDIUM or HIGH keywords provide a RESOLUTION *percent* clause with which you can set the percentage of data distributed to every bin. Unless you change the *percent* value with a RESOLUTION clause:

- The resolution for HIGH keyword defaults to `0.5` percent, which means that each bin contains 0.5 percent of one column's data.
- The resolution for HIGH keyword defaults to `2.5` percent.

The HIGH keyword collects data from every row to construct an exact distribution for each column. UPDATE STATISTICS (MEDIUM) samples a percentage of data rows to construct statistically significant, but not exact, distribution data. The medium distribution usually contains significantly less data and takes less time to construct than a high distribution on the same table.

You can specify a confidence ratio with the MEDIUM keyword. If you do not specify a value for *conf*, the default confidence is `0.95`, which means that for approximately 95 percent of samples, the estimate is equivalent to using high distributions.

The HIGH keyword 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. The MEDIUM keyword scans tables at least once and takes longer to execute on a given table than the LOW keyword. To minimize processing time, specify a table name and column names within that table.

Tip: *The amount of space that the **DBUPSPACE** environment variable designates determines the number of times that the database server scans a table. For information about **DBUPSPACE**, see Chapter 3 of the Informix Guide to SQL: Reference.*

MEDIUM or HIGH with DISTRIBUTIONS ONLY

The UPDATE STATISTICS statement reads through index pages to:

- compute statistics for the query optimizer.
- locate pages that have the delete flag marked as one.

  Keys in these pages are removed from the btree cleaner list. For information on the btree cleaner list, see the *INFORMIX-Universal Server Administrator's Guide*.

Examining index information can consume considerable processing time. If you specify the DISTRIBUTIONS ONLY option with the MEDIUM or HIGH keywords, you do not collect statistics for index information.

The DISTRIBUTIONS ONLY keyword has no effect on information about tables, such as the number of pages used, the number of rows, and fragment information. UPDATE STATISTICS needs this data to construct accurate column distributions and requires little time and system resources to collect it.

In the following example, the UPDATE STATISTICS statement gathers distributions information, index information, and table information for the **customer** table:

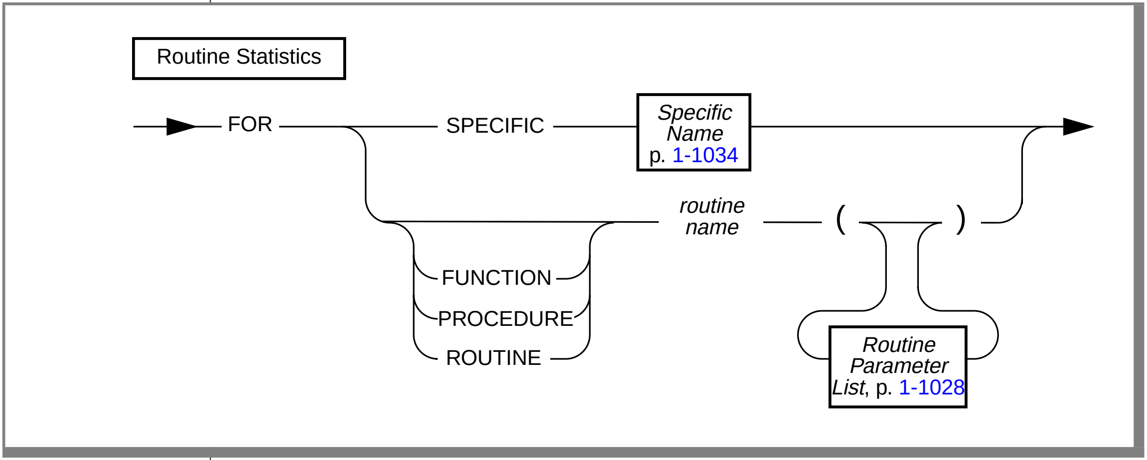
```
UPDATE STATISTICS MEDIUM FOR TABLE customer
```

However, in the following example, only distributions information and table information are gathered for the **customer** table. The DISTRIBUTIONS ONLY option prevents the construction of index information.

```
UPDATE STATISTICS MEDIUM FOR TABLE customer
    DISTRIBUTIONS ONLY
```

Updating Statistics for Routines

Use the FOR ROUTINE, FOR FUNCTION, FOR PROCEDURE, or FOR SPECIFIC clause to avoid updating tables. You specify a routine name or specific name, or a routine parameter list to limit the scope of the UPDATE STATISTICS statement to a particular definition of a routine.



Element	Purpose	Restrictions	Syntax
routine name	The name given to the routine in a CREATE FUNCTION or CREATE PROCEDURE statement	The identifier must refer to an existing user-defined routine. In an ANSI-compliant database, specify the owner as the prefix to the routine name.	Function Name, p. 1-959 or Procedure Name, p. 1-1004

Keyword	Function
SPECIFIC	Reoptimizes the execution plan for a routine identified by *specific name*.
FUNCTION	Reoptimizes the execution plan for any function with the specified routine name (and parameter types that match *routine parameter list,* if supplied).
PROCEDURE	Reoptimizes the execution plan for any procedure with the specified routine name (and parameter types that match *routine parameter list,* if supplied).
ROUTINE	Reoptimizes the execution plan for functions and procedures with the specified routine name (and parameter types that match *routine parameter list,* if supplied).

Recommended Procedure for Updating Statistics

Informix recommends the following procedure for giving the optimizer the best possible information while incurring the lowest performance penalty:

1. Run UPDATE STATISTICS in medium mode with the DISTRIBUTIONS ONLY option for each table. (If you are the database owner or DBA, and you want to gather statistics for the entire database, you can do that with a single command instead.). The default parameters are sufficient unless the table is very large. In this case, use a resolution of `1.00` and a confidence level of `0.99`.
2. Run UPDATE STATISTICS in high mode for all columns that head an index. For the fastest execution time of the UPDATE STATISTICS statement, you must execute one UPDATE STATISTICS statement in the high mode for each such column.
3. For each multicolumn index, run UPDATE STATISTICS in low mode for all its columns.

This procedure executes rapidly because it constructs the index-information statistics only once for each index.

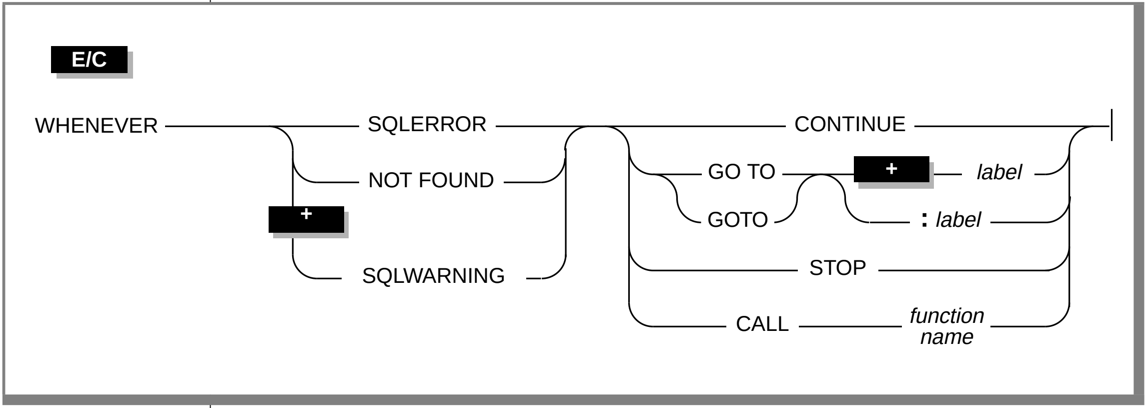
References

In the *INFORMIX-Universal Server Performance Guide*, see the discussion of UPDATE STATISTICS. In the *Informix Migration Guide*, see the discussion of how to use the **dbschema** utility to view distributions created with UPDATE STATISTICS.

WHENEVER

Use the WHENEVER statement to trap exceptions that occur during the execution of SQL statements.

Syntax



Element	Purpose	Restrictions	Syntax
function name	Function or procedure that is called when an exception occurs	Function or procedure must exist at compile time.	Function or procedure name must conform to language-specific rules for functions or procedures.
label	Statement label to which program control transfers when an exception occurs	The label must be defined in same source file.	Label must conform to language-specific rules for statement labels.

Usage

Use of 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 Clause	For More Information
Errors	SQLERROR	page 1-817
Warnings	SQLWARNING	page 1-817
Not Found Condition End of Data Condition	NOT FOUND	page 1-818

If you do not use the WHENEVER statement in a program, the program does not automatically abort when an exception occurs. Your program must explicitly check for exceptions and take whatever corrective action you desire. If you do not check for exceptions, the program simply continues running. However, as a result of the errors, the program might not perform its intended purpose.

In addition to specifying the type of exception for which to check, the WHENEVER statement also specifies what action to take when the specified exception occurs. The following table summarizes possible actions that WHENEVER can specify.

Type of Action	WHENEVER Keyword	For More Information
Continue program execution	CONTINUE	page 1-818
Stop program execution	STOP	page 1-818
Transfer control to a specified label	GOTO	page 1-818
Transfer control to a named function or procedure	CALL	page 1-819

Scope of WHENEVER

The ESQL preprocessor, not the database server, handles the interpretation of the WHENEVER statement. When the preprocessor encounters a WHENEVER statement in an ESQL source file, it inserts the appropriate code into the preprocessed code after each SQL statement based on the exception and the action that WHENEVER lists. The preprocessor defines the scope of a WHENEVER statement as from the point that it encounters the statement in the source module until it encounters one of the following conditions:

- The next WHENEVER statement with the same exception condition (SQLERROR, SQLWARNING, and NOT FOUND) in the same source module
- The end of the source module

Whichever condition the preprocessor encounters first as it sequentially processes the source module marks the end of the scope of the WHENEVER statement.

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. However, the SELECT statement 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');
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;

printf("\n\nGoing to try second insert\n\n");
EXEC SQL insert into test_color values ('blue');
EXEC SQL select paint_type from paint where color='red';
EXEC SQL disconnect all;
printf("\n\nProgram over\n\n");
}
```

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**) is less than zero and the **SQLSTATE** variable has a class code with a value greater than 02. The following statement causes a program to stop execution if an SQL error exists:

```
WHENEVER SQLERROR STOP
```

If you do not use any WHENEVER SQLERROR statements in a program, the default for WHENEVER SQLERROR is CONTINUE.

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 has 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. For more information, see the chapter on exception checking in the *INFORMIX-ESQL/C Programmer's Manual*.

The following statement causes a program to stop execution if a warning condition exists:

```
WHENEVER SQLWARNING STOP
```

If you do not use any WHENEVER SQLWARNING statements in a program, the default for WHENEVER SQLWARNING is CONTINUE.

NOT FOUND Keywords

If you use the NOT FOUND keywords, exception handling for SELECT and FETCH statements is treated differently than for 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 (**sqlca.sqlcode**) is set to 100, and the **SQLSTATE** variable has a class code of 02.

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 option.

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 particular exception occurs. The GOTO keyword is the ANSI-compliant syntax of the clause. The GO TO keywords are a non-ANSI synonym for GOTO.

The following example shows a WHENEVER statement in INFORMIX-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");
    .
    .
    .
```

You must define the labeled statement in *each* program block that contains SQL statements. If your program contains more than one function, you might need to include the labeled statement and its code in *each* function. When the preprocessor reaches the function that does not contain the labeled statement, it tries to insert the code associated with the labeled statement. However, if you do not define this labeled statement within the function, the preprocessor generates an error.

To correct this error, either put a labeled statement with the same label name in each function, 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 named function or procedure when a particular exception occurs. Do not include parentheses after the function or procedure 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 named function completes, 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 named function.

Observe the following restrictions on the named function:

- You cannot pass arguments to the named function nor can you return values from the named function. If the named function needs external information, use global variables or the GOTO clause of WHENEVER to transfer control to a label that calls the named function.
- SPL

  You cannot specify the name of a stored function as a named function. To call a stored function, use the CALL clause to execute a function that contains the EXECUTE FUNCTION statement. ♦
- Make sure that all functions that the WHENEVER...CALL statement affects can find a declaration of the named function.

References

See the EXECUTE FUNCTION, FETCH and GET DIAGNOSTICS statements in this manual.

See the chapter on exception checking and error checking in your SQL API product manual.

Segments

Segments are language elements, such as table names and expressions, that occur repeatedly in the syntax diagrams for SQL and SPL statements. These language elements are discussed separately in this section for the sake of clarity, ease of use, and comprehensive treatment.

Whenever a segment occurs within the syntax diagram for an SQL or SPL statement, the diagram references the description of the segment 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 syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

If a segment consists of multiple parts, the segment description provides the same set of information for each part. Each segment description concludes with references to related information in this manual and other manuals.

Use of Segment Descriptions

The syntax diagram within each segment description is not a standalone diagram. Instead it is a subdiagram that is subordinate to the syntax diagram for an SQL or SPL statement. A reference box in the syntax diagram for the statement refers to this subdiagram by providing the name of the segment and the page number on which the segment description begins.

First look up the syntax for the statement, and then turn to the segment description to find out the complete syntax for the segment. You will probably never need to look up the segment first and then work backward to a statement or statements that contain the segment.

For example, if you are using INFORMIX-Universal Server, and you want to enter a CREATE VIEW statement that includes a database name and database server name in the view name, first look up the syntax diagram for the CREATE VIEW statement. Then use the reference box for the View Name segment in that syntax diagram to look up the subdiagram for the View Name segment.

The subdiagram for the View Name segment shows you how to qualify the simple name of a view with the name of the database or with the name of both the database and the database server. Use the syntax in the subdiagram to enter a CREATE VIEW statement that includes the database name and database server name in the view name. The following example creates the **name_only** view 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:

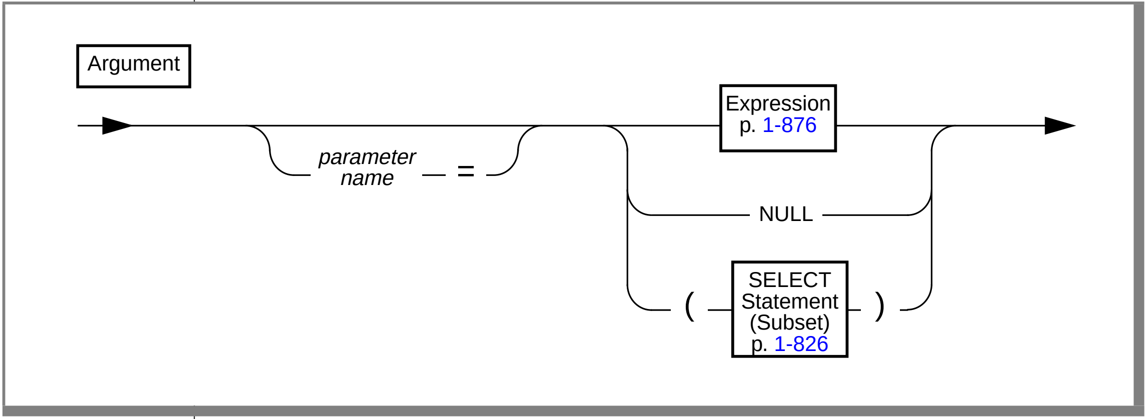
- Argument
- Collection Derived Table
- Condition
- Constraint Name
- Database Name
- Data Type
- DATETIME Field Qualifier
- Expression
- External Routine Reference
- Function Name
- Identifier
- Index Name
- INTERVAL Field Qualifier
- Literal Collection
- Literal DATETIME
- Literal INTERVAL

- Literal Number
- Literal Row
- Procedure Name
- Quoted Pathname
- Quoted String
- Relational Operator
- Return Clause
- Routine Modifier
- Routine Parameter List
- Specific Name
- Statement Block
- Synonym Name
- Table Name
- View Name

Argument

Use an Argument to pass a specific value to a routine parameter.

Syntax



Element	Purpose	Restrictions	Syntax
parameter name	The name of a routine parameter for which you supply an argument	The parameter name must match the parameter name that you specified in a corresponding CREATE FUNCTION or CREATE PROCEDURE statement.	Expression, p. 1-876

Usage

A parameter list for a routine is defined in the CREATE PROCEDURE or CREATE FUNCTION statement. If the routine has a parameter list, you can enter arguments when you execute the routine. An argument is a specific data element that matches the data type of one of the parameters for the routine.

When you execute a routine, you can enter arguments in one 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
- With no parameter name, if the arguments are in the same order as the parameters

If you use a parameter name for one argument, you must use a parameter name for all the arguments.

In the following example, both statements are valid for a function that expects three character arguments, **t**, **d**, and **n**:

```
EXECUTE FUNCTION add_col (t ='customer', d ='integer', n ='newint');
EXECUTE FUNCTION add_col ('customer','newint','integer');
```

When you use the parameter name in the argument list (called passing parameters by name), the process of routine resolution is partial, based only on the routine type (FUNCTION or PROCEDURE), the routine name, and the number of arguments.

Comparing Arguments to the Parameter List

When you create or register a routine with CREATE PROCEDURE or CREATE FUNCTION, you specify a parameter list with the names and data types of the parameters the routine expects.

If you attempt to execute a routine with more arguments than the routine expects, you receive an error.

If you execute a routine with fewer arguments than the routine expects, the 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 routine.

If missing arguments do not have default values, the database server initializes the arguments to the value UNDEFINED. However, you cannot use a variable with a value of UNDEFINED within the routine. If you do, INFORMIX-Universal Server issues an error.

Subset of SELECT Allowed in a Routine Argument

You can use any SELECT statement as the argument for a routine if it returns exactly one value of the proper data type and length. For more information, see the discussion of the SELECT statement on page 1-593.

Subset of Expressions Allowed as an Argument

You can use any expression as an argument, except an aggregate expression. If you use a subquery or function call, the subquery or function must return a single value of the appropriate data type and size. For the full syntax of an expression, see page 1-876.

References

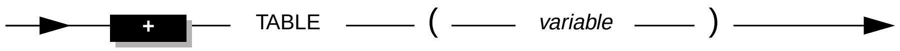
In this manual, see the CREATE FUNCTION, CREATE PROCEDURE, EXECUTE FUNCTION, EXECUTE PROCEDURE, CALL, FOREACH, and LET statements. See also the Parameter List segment.

For information about how to write external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual. In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to write SPL routines.

Collection Derived Table

The Collection Derived Table segment specifies a collection or row variable to access instead of a table name.

Syntax



Element	Purpose	Restrictions	Syntax
variable	The name of an ESQL/C or SPL collection variable or of an ESQL/C **row** variable.	The variable must be declared as a collection variable in an ESQL/C program or SPL routine, or as a **row** variable in an ESQL/C program.	Variable name must conform to language-specific rules for variable names.

Usage

The TABLE keyword introduces the name of the collection or row variable that you want to access as a collection derived table. The variable can be typed or untyped. For example, the following INSERT statement uses the *variable* in its Collection Derived Table clause:

```
INSERT INTO TABLE(variable) VALUES ...
```

E/C

In an ESQL/C program, *variable* is a host variable for either a collection or a row and is declared as either the **collection** or **row** data type. ♦

SPL

In an SPL program, *variable* is an SPL variable that is declared as a COLLECTION, MULTISET, SET, or LIST data type. ♦

When you use the Collection Derived Table segment, you access the elements of a collection or the fields of a row variable. Use of this segment does *not* affect the associated column or columns in a database. Once you have completed the modifications to the variable, save the new values in the database with the UPDATE or INSERT statement.

Accessing a Collection Variable

The TABLE keyword makes the collection variable a collection derived table, that is, a collection appears as a table in an SQL statement. You can think of a collection derived table as a table of one column, with each element of the collection being a row of the table.

Use the TABLE keyword in place of the name of a table, synonym, or view name 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. ♦

E/C

The following ESQL/C code fragment inserts the element 3 into the **a_list** collection variable and then saves this collection variable in the **list_col** column of the **tab_list** table:

```
EXEC SQL insert into table(:a_list) values(3);
EXEC SQL update tab_list set list_col = :a_list;
```

If the ESQL/C **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. Suppose the **a_list** host variable is declared as an untyped collection variable, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection a_list;
EXEC SQL END DECLARE SECTION;
```

The following code fragment obtains the collection type for the **a_list** variable before it uses the collection variable in an UPDATE statement:

```
/* select LIST column into the untyped collection variable */
/* to obtain the element type */
EXEC SQL select list_col into :a_list from tab_list;

/* insert an element into the untyped collection variable */
EXEC SQL insert into table(:a_list) values (7);

/* update the LIST column with the collection variable */
EXEC SQL update tab_list set list_col = :a_list;
```

The following SPL code fragment inserts the element 3 into the **a_list** collection variable and then saves this collection variable in the **list_col** column of the **tab_list** table:

```
INSERT INTO TABLE(a_list) VALUES(3);
UPDATE tab_list SET list_col = a_list;
```

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, you can think of the collection derived table as a table of one row, with each field of the row type being a column of the table row.

Use the TABLE keyword in place of the name of a table, synonym, or view name in the following 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;
```

♦

References

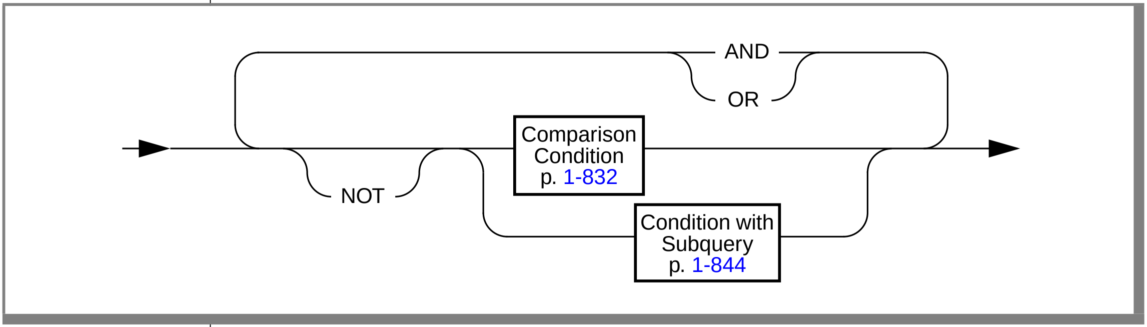
See the DECLARE, DELETE, FETCH, INSERT, SELECT, and UPDATE statements in Chapter 1 of this manual for further information about how to access collection variables.

For information on how to use collection variables in an SPL routine, see Chapter 14 in the *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 *INFORMIX-ESQL/C Programmer's Manual*.

Condition

A condition tests data to determine whether it meets certain qualifications. Use the Condition segment wherever you see a reference to a condition in a syntax diagram.

Syntax


AND
OR
NOT
Comparison Condition p. 1-832
Condition with Subquery p. 1-844

Usage

A condition is a collection of one or more search conditions, optionally connected by the logical operators AND or OR. Search conditions fall into the following categories:

- Comparison conditions (also called filters or Boolean expressions)
- Conditions with a subquery

Restrictions on a Condition

A condition can contain only an aggregate function if it is used in the HAVING clause of a SELECT statement or the HAVING clause of a subquery. You cannot use an aggregate function in a comparison condition that is part of a WHERE clause in a DELETE, SELECT, or UPDATE statement unless the aggregate is on a correlated column that originates from a parent query and the WHERE clause is within a subquery that is within a HAVING clause.

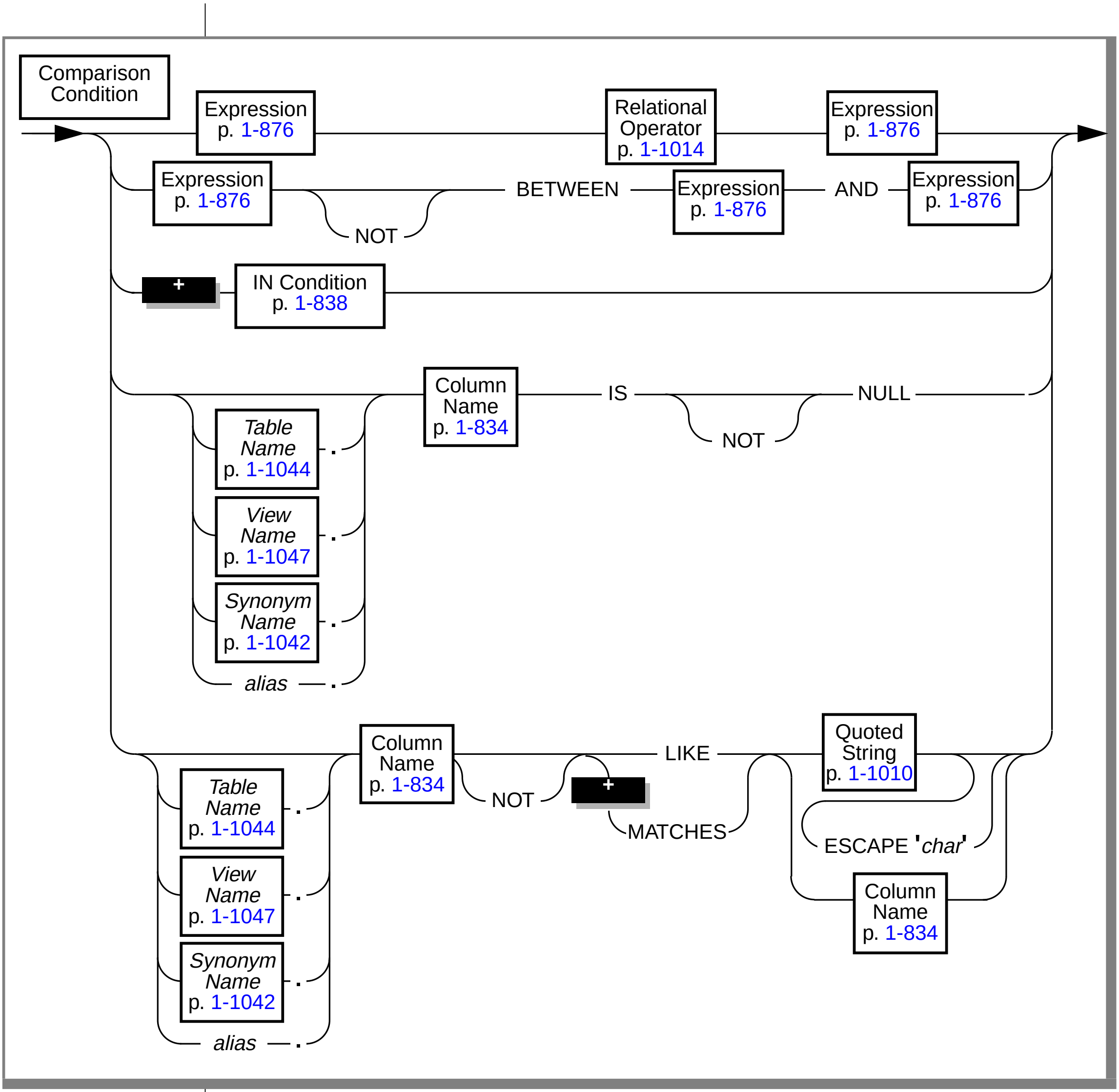
NOT Operator

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 (uses a null in the determination), the NOT operator has no effect. The following truth table shows the effect of NOT. The letter T represents a true condition, F represents a false condition, and a question mark (?) represents an unknown condition.Unknown values occur when part of an expression that uses an arithmetic operator is null.

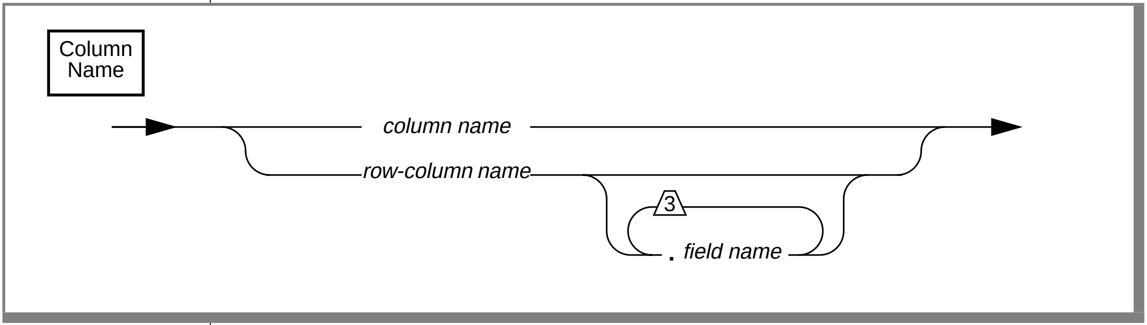
NOT	
T	F
F	T
?	?

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 evaluate to a simple true or false result. Their syntax is summarized in the following diagram and explained in detail after the diagram.


Comparison Condition
Expression p. 1-876
Relational Operator p. 1-1014
Expression p. 1-876
Expression p. 1-876
NOT
BETWEEN
Expression p. 1-876
AND
Expression p. 1-876
+
IN Condition p. 1-838
Table Name p. 1-1044
.
View Name p. 1-1047
.
Synonym Name p. 1-1042
.
alias
.
Column Name p. 1-834
IS
NOT
NULL
Table Name p. 1-1044
.
View Name p. 1-1047
.
Synonym Name p. 1-1042
.
alias
.
Column Name p. 1-834
NOT
+
LIKE
MATCHES
Quoted String p. 1-1010
ESCAPE 'char'
Column Name p. 1-834

Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement	You must have defined the alias in the FROM clause of the SELECT statement.	Identifier, p. 1-962
char	A single ASCII character that is to be used as the escape character within the quoted string in a LIKE or MATCHES condition	See "ESCAPE with LIKE" on page 1-843 and "ESCAPE with MATCHES" on page 1-843.	Quoted String, p. 1-1010



Element	Purpose	Restrictions	Syntax
column name	The name of a column that is used in an IS NULL condition or in a LIKE or MATCHES condition. See "IS NULL Condition" on page 1-840 and "LIKE and MATCHES Condition" on page 1-840 for more information on the meaning of *column name* in these conditions.	The column must exist in the specified table.	Identifier, p. 1-962
field name	The name of the field that you want to compare in the row column.	The field must be a component of the row type that *row-column name* or *field name* (for nested rows) specifies.	Identifier, p. 1-962
row-column name	The name of the row column that you specify.	The data type of the column must be a named row type or an unnamed row type.	Identifier, p. 1-962

Refer to the following sections for more information on the use of the different types of comparison conditions:

- For relational-operator conditions, refer to "Relational-Operator Condition" on page 1-836.
- For the BETWEEN condition, refer to "BETWEEN Condition" on page 1-837.
- For the IN condition, refer to "IN Condition" on page 1-838.
- For the IS NULL condition, refer to "IS NULL Condition" on page 1-840.
- For the LIKE and MATCHES condition, refer to "LIKE and MATCHES Condition" on page 1-840.

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, you do not need to surround the constant expression with quotation marks.
- If the column has a character data type, you must surround the constant expression with quotation marks.
- If the column has a date data type, you should 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. The **ship_instruct** column has a character data type. The **order_date** column has a date data type. The **ship_weight** column has a numeric data type.

```
SELECT * FROM orders
    WHERE ship_instruct = 'express'
    AND order_date > '05/01/94'
    AND ship_weight < 30
```

Relational-Operator Condition

Some relational-operator conditions are shown in the following examples:

```
city[1,3] = 'San'

o.order_date > '6/12/86'

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 **paid_date** has a null value, 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 shown in the following example. The IS NULL condition is explained fully in "IS NULL Condition" on page 840.

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date IS NULL
```

For more information, see the Relational Operator segment on page 1-1014.

BETWEEN Condition

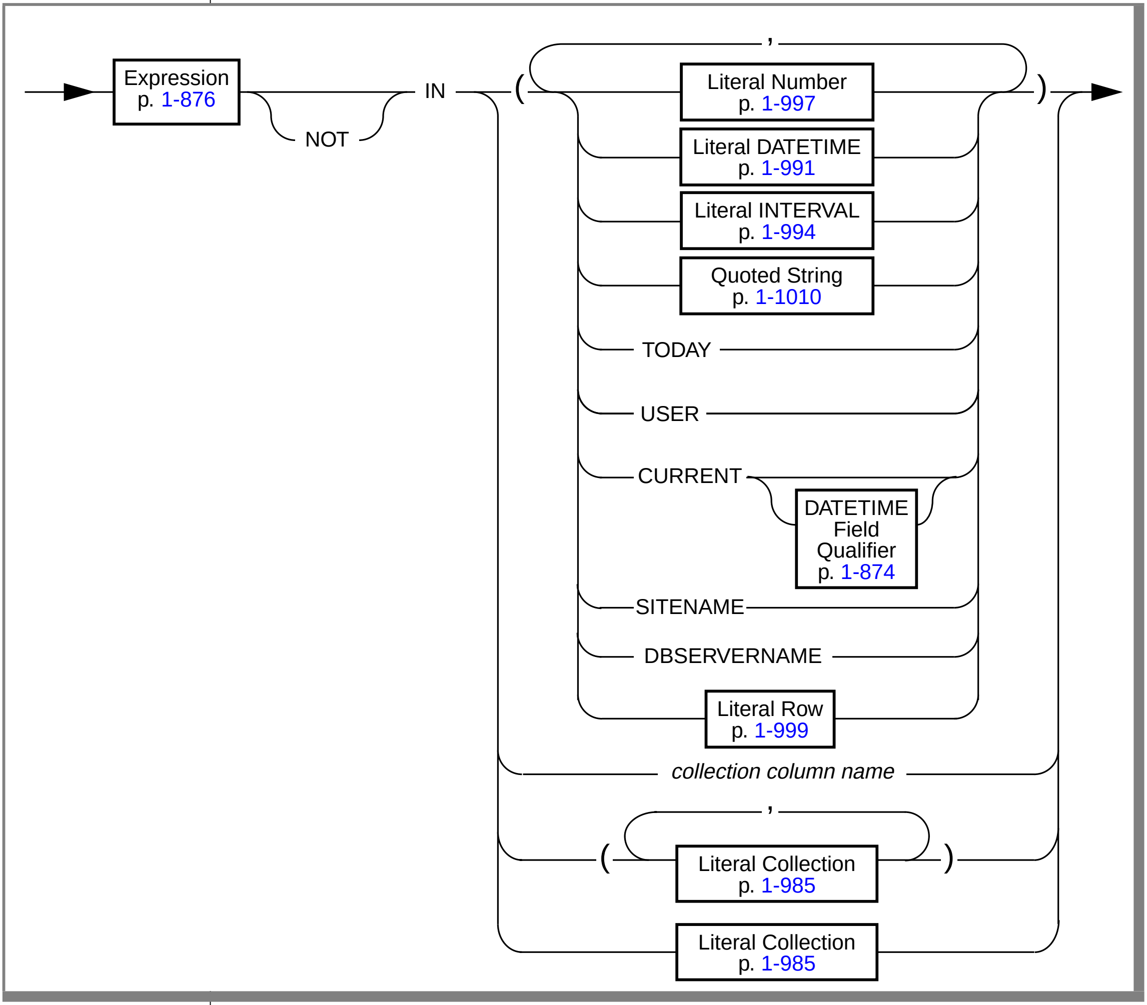
For a BETWEEN test to be true, the value of the expression on the left of the BETWEEN keyword must be in the inclusive range of the values of the two expressions on the right of the BETWEEN keyword. Null values do not satisfy the condition. You cannot use NULL for either expression that defines the range.

Some BETWEEN conditions are shown in the following examples:

```
order_date BETWEEN '6/1/93' and '9/7/93'

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 word IN is included in the list of items.



Element	Purpose	Restrictions	Syntax
collection column name	The name of a collection column that is used in an IN condition	The column must exist in the specified table.	Identifier, p. 1-962

The NOT option produces a search condition that is satisfied when the expression is not in the list of items. Null values do not satisfy the 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)

WHERE row_col.state IN ('CA', 'WA')
```

ESQL

The TODAY function is evaluated at execution time; CURRENT is evaluated when a cursor opens or when the query executes, if it is a singleton SELECT statement. ♦

The USER function is case sensitive; it perceives **minnie** and **Minnie** as different values.

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 you search 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 the IN operator to search for an element in 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 hand side of the of the IN operator is an element in the set of values on the right hand side.

IS NULL Condition

The IS NULL condition is satisfied if the column contains a null value. If you use the IS NOT NULL option, 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
```

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 keyword. 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 row type 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 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

If you use the keyword LIKE, you can use the following wildcard characters in the quoted string.

Wildcard	Meaning
%	The percent sign (%) matches zero or more characters.
_	The underscore (_) matches any single character.
\	The backslash (\) removes the special significance of the next character (used to match % or _ by writing \% or _).

Using the backslash (\) as an escape character is an Informix extension to ANSI-compliant SQL.

ANSI

If you use an escape character to escape anything other than percent sign (%), underscore (_), or the escape character itself, an error is returned. ♦

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 following condition tests for all descriptions that contain an underscore. The backslash (\) is necessary because the 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. For more information, see the *Extending INFORMIX-Universal Server: Data Types* manual.

MATCHES Operator

If you use the keyword MATCHES, you can use the following wildcard characters in the quoted string.

Wildcard	Meaning
*	The asterisk (*) matches zero or more characters.
?	The question mark (?) matches any single character.
[...]	The brackets ([...]) match any of the enclosed characters, including character ranges as in [a to z]. A caret (^) as the first character within the brackets matches any character that is not listed. Hence [^abc] matches any character that is not a, b, or c.
\	The backslash (\) removes the special significance of the next character (used to match * or ? by writing * or \?).

The following condition tests for the string tennis, alone or in 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 MATCHES operator has an associated operator function called **matches()**. You can define a **matches()** function to handle your own user-defined data types. For more information, see the *Extending INFORMIX-Universal Server: Data Types* manual.

ESCAPE with LIKE

The ESCAPE keyword lets you include an underscore (_) or a percent sign (%) in the quoted string and avoid having them be interpreted as wildcards. If you choose to use z as the escape character, the characters z_ in a string stand for the character _. Similarly, the characters z% represent the percent sign (%). Finally, the characters zz in the string stand for the single character z. The following statement retrieves rows from the **customer** table in which the **company** column includes the underscore character:

```
SELECT * FROM customer
    WHERE company LIKE '%z_%' ESCAPE 'z'
```

You can also use a single-character host variable as an escape character. The following statement shows the use of a host variable as 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 lets you include a question mark (?), an asterisk (*), and a left or right bracket ([]) in the quoted string and avoid having them be interpreted as wildcards. 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 (?):

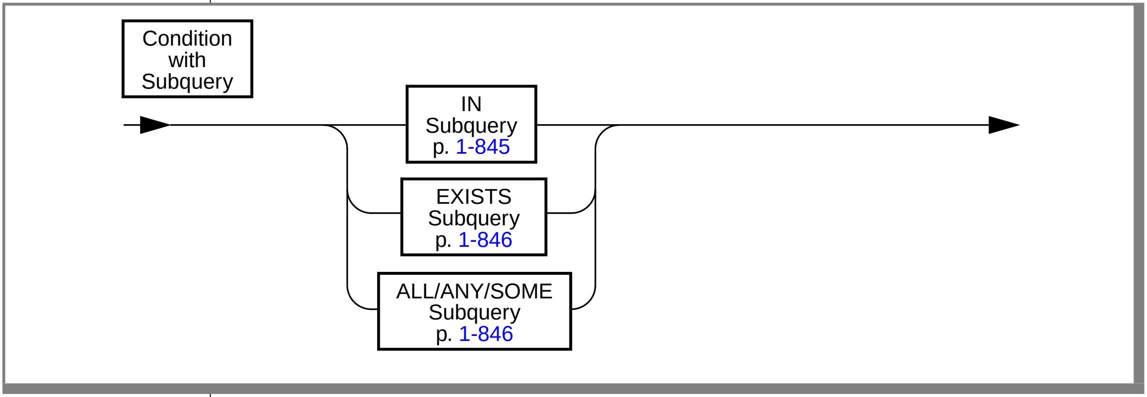
```
SELECT * FROM customer
    WHERE company MATCHES '*z?*' ESCAPE 'z'
```

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 only if its result is of the Boolean type. For example, the following example returns a value of the Boolean type:

```
funcname(x)
```

Condition with a Subquery



You can use 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:

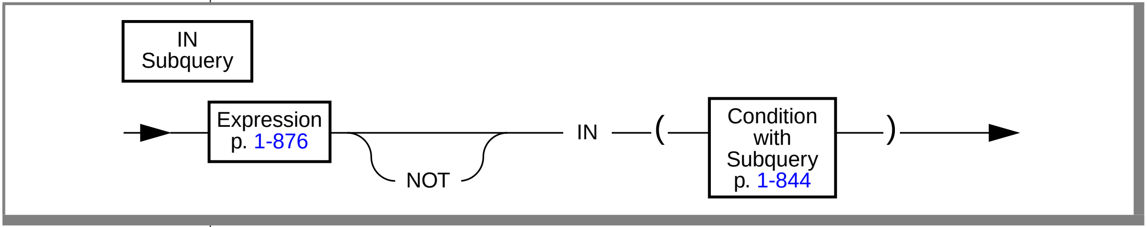
- Compare an expression to the result of another SELECT statement
- Determine whether an expression is included in the results of another SELECT statement
- Ask whether another SELECT statement selects any rows

The subquery can depend on the current row that the outer SELECT statement is evaluating; in this case, the subquery is a *correlated subquery.*

The kinds of subquery conditions are shown in the following sections with their syntax.

A subquery can return a single value, no value, or a set of values depending on the context in which it is used. 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 an ORDER BY clause. The full syntax of the SELECT statement is described on page 1-593.

IN Subquery



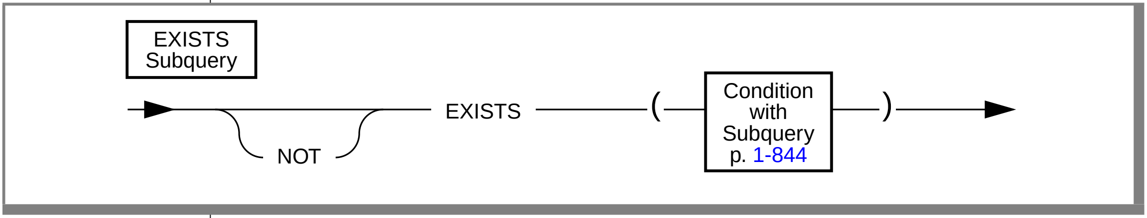
An IN subquery condition is true if the value of the expression matches one or more of the values that the subquery selects. The subquery must return only one column, but it can return more than one row. The keyword IN is equivalent to the =ANY sequence. The keywords NOT IN are equivalent to the !=ALL sequence. See "ALL / ANY / SOME Subquery" on page 1-846.

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, you can put the UNIQUE or DISTINCT keyword into the subquery with no effect on the query results, although eliminating testing duplicates can reduce the time needed for running the query.

EXISTS Subquery



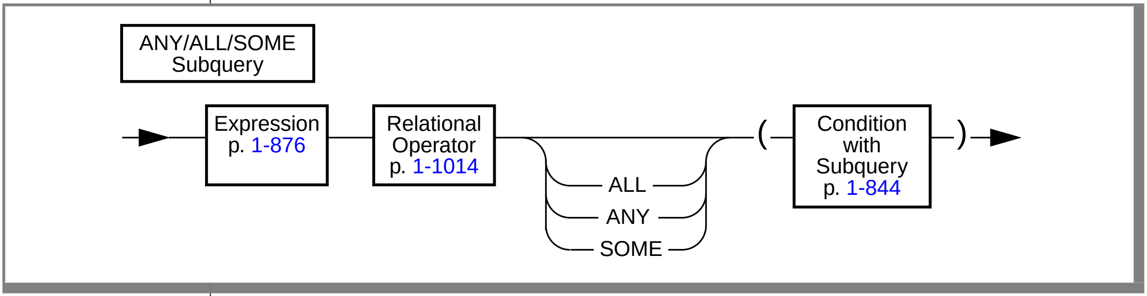
An EXISTS subquery condition evaluates to true if the subquery returns a row. With an EXISTS subquery, one or more columns can be returned. The subquery always contains a reference to a column of the table in the main query. If you use an aggregate function in an EXISTS subquery, at least one row is 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 whole row is tested; specific column values are not tested.

ALL/ANY/SOME Subquery


ANY/ALL/SOME Subquery
Expression p. 1-876
Relational Operator p. 1-1014
ALL
ANY
SOME
(
Condition with Subquery p. 1-844
)

You use the ALL, ANY, and SOME keywords to specify what makes the search 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.

Keyword	Meaning
ALL	A keyword that denotes 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.
ANY	A keyword that 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.
SOME	An alias for ANY

In the following example of the ALL subquery, 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)
```

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 any subquery value. The condition, which is found in the following example of the NOT keyword with an ANY subquery, 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 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)
```

Conditions with AND or OR Operators

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 > '1/1/93' 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. Unknown values 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	?	?

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.

References

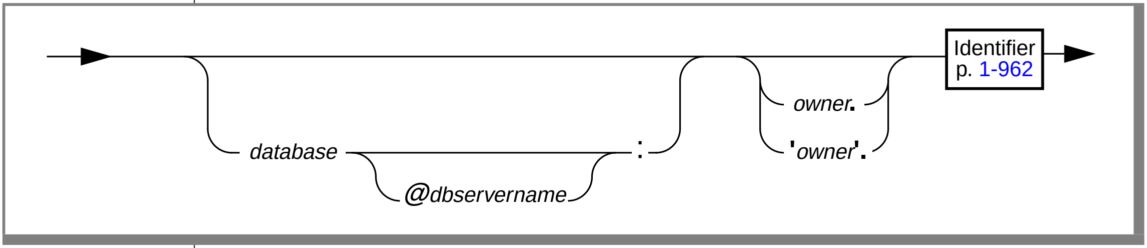
In the *Informix Guide to SQL: Tutorial*, see the discussion of conditions in the SELECT statement in Chapter 2 and Chapter 3.

In the *Guide to GLS Functionality*, see the discussion of the SELECT statement for information on the GLS aspects of conditions.

Constraint Name

The Constraint Name segment specifies the name of a constraint. Use the Constraint Name segment whenever you see a reference to a constraint name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the constraint resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the Universal Server database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that *dbservername* specifies must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the constraint	If you are using an ANSI-compliant database, you must enter the *owner.* parameter for a constraint that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the constraint is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of constraints. For more information, see the *Guide to GLS Functionality*. ♦

When you create a constraint, the name of the constraint must be unique within the database if the database is not ANSI compliant.

ANSI

When you create a constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within a database.

The *owner.name* combination is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the constraint owner is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

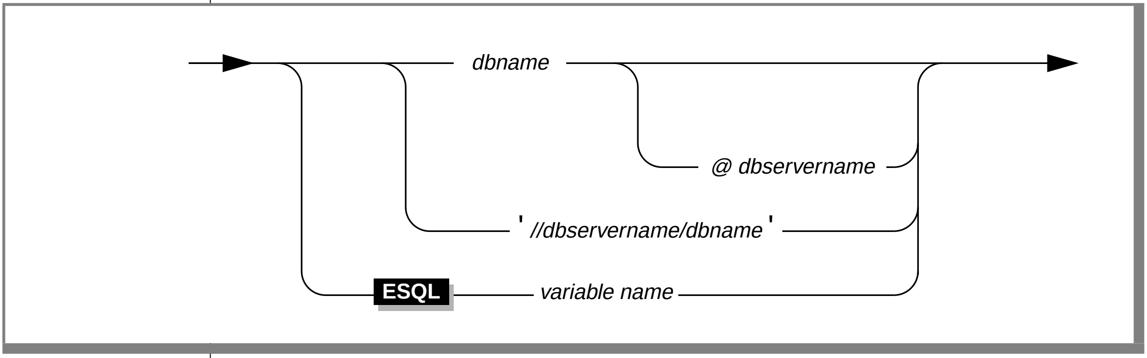
References

See the CREATE TABLE statement in this manual for information on defining constraints.

Database Name

The Database Name segment specifies the name of a database. Use the Database Name segment whenever you see a reference to a database name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dbname	The name of the database itself. This simple name does not include the pathname or the database server name.	A database name must be unique among the database names on the same database server. Database names are not case sensitive. If you are using Universal Server, the database name can have a maximum of 18 characters.	Identifier, p. 1-962
dbservername	The name of the database server on which the database that is named in *dbname* resides. The @ symbol is a literal character that introduces the database server name. Specifying a database server name allows you to choose a database on another database server as your current database. You can name the current database server using *dbservername*, although that is extra information.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file. You can put a space between *dbname* and the @ symbol, or you can omit the space. You cannot put a space between the @ symbol and *dbservername*. For restrictions on *dbservername* that are specific to syntax formats that use quotation marks and slash symbols, see "/ / dbservername / dbname Option" on page 1-854.	Identifier, p. 1-962
variable name	The name of a host variable that holds the database name	Contents must comply with restrictions on *dbservername*	Name of the host variable must conform to language-specific rules for variable names.

Usage

The simple database name is an SQL identifier, as described on page 1-962. If you are creating a database, the name that you assign to the database can be 18 characters, inclusive. Database names are not case sensitive. You cannot use delimited identifiers for a database name.

The maximum length of the database name, including *dbservername*, is 128 characters.

The following example shows a database specification:

```
empinfo@personnel
```

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of databases. For more information, see the *Guide to GLS Functionality*. ♦

@dbservername Option

If you use a database server name, do not put any spaces between the name and the @ symbol. For example, the following format is valid for the **stores7** database on the **training** database server:

```
stores7@training
```

//dbservername/dbname Option

If you use the alternative naming method, do not put spaces between the quotes, slashes, and names, as the following example shows:

```
'//training/stores7'
```

variable name Option

ESQL

You can use a variable within an SQL API to hold the name of a database. ♦

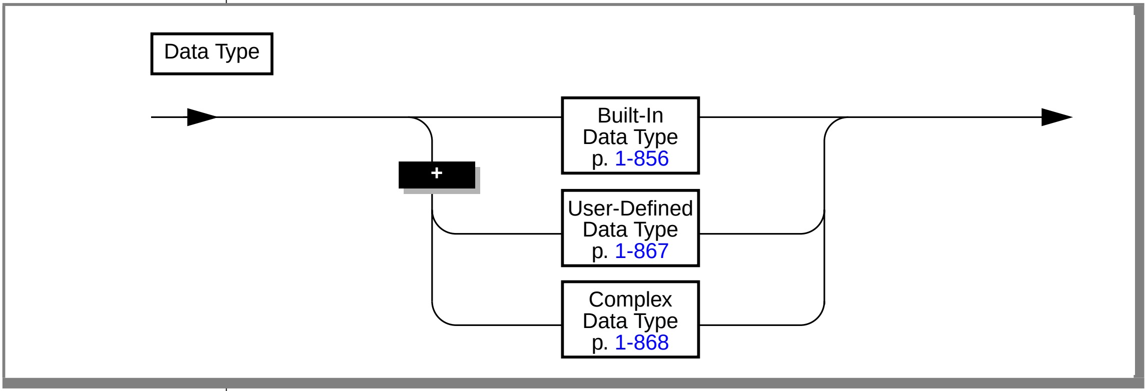
References

See the CREATE DATABASE and RENAME DATABASE statements in this manual for information on naming databases.

Data Type

The Data Type segment specifies the data type of a column or value. Use the Data Type segment whenever you see a reference to a data type in a syntax diagram.

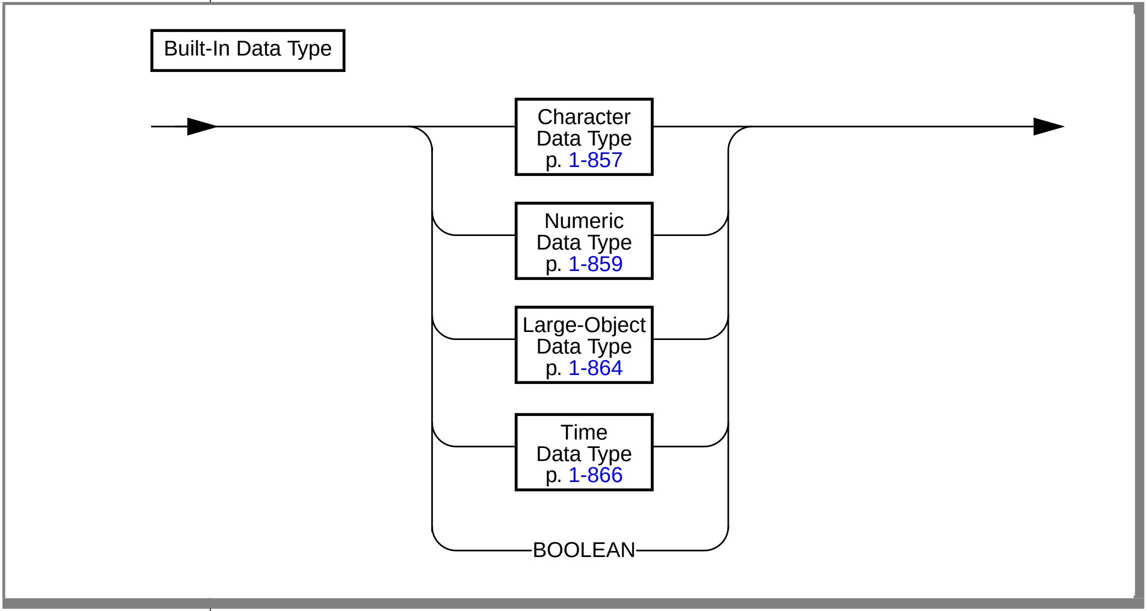
Syntax



Usage

The following sections summarize each of the categories of data types that the Universal Server supports. For more information, see the discussion of all data types in the *Informix Guide to SQL: Reference*.

Built-In Data Type


Built-In Data Type
Character Data Type p. 1-857
Numeric Data Type p. 1-859
Large-Object Data Type p. 1-864
Time Data Type p. 1-866
BOOLEAN

Built-in data types are data types that are fundamental to the database server. These data types are built into the database server in the sense that the knowledge for how to interpret and transfer these data types is part of the database server software.

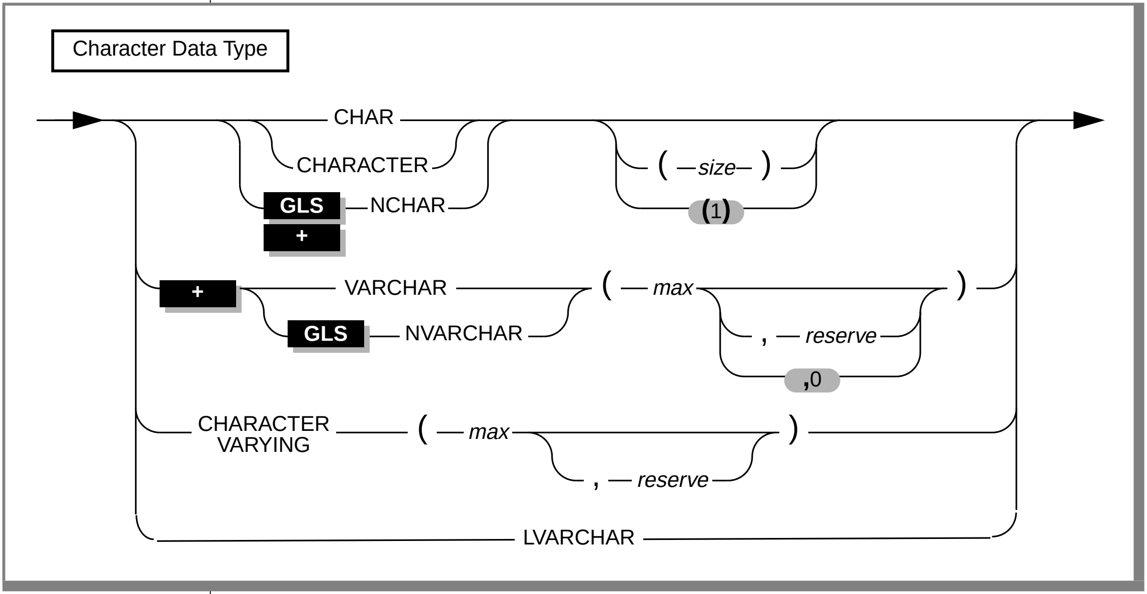
Universal Server supports the following categories of built-in data types:

- Character data types
- Numeric data types
- Large-object data types
- Time data types

The following sections describe each of the data-type categories in more detail.

In addition, Universal Server supports the BOOLEAN data type. For more information on the BOOLEAN data type, see Chapter 2 of the *Informix Guide to SQL: Reference* and Chapter 9 of the *Informix Guide to SQL: Tutorial*.

Character Data Types



Element	Purpose	Restrictions	Syntax
max	Maximum size of a CHARACTER VARYING or VARCHAR or NVARCHAR column in bytes	You must specify an integer value between 1 and 255 bytes inclusive. If you place an index on the column, the largest value you can specify for *max* is 254 bytes.	Literal Number, p. 1-997
size	Number of bytes in the CHAR or NCHAR column	You must specify an integer value between 1 and 32,767 bytes inclusive.	Literal Number, p. 1-997
reserve	Amount of space in bytes reserved for a CHARACTER VARYING or VARCHAR or NVARCHAR column even if the actual number of bytes stored in the column is less than *reserve*	You must specify an integer value between 0 and 255 bytes. However, the value you specify for *reserve* must be less than the value you specify for *max*.	Literal Number, p. 1-997

The following table summarizes the character data types that Universal Server supports.

Data Type	Purpose
CHAR	Stores single-byte or multibyte text strings of up to 32,767 bytes of text data and supports code-set collation of text data.
CHARACTER	Is an ANSI-compliant synonym for CHAR.
CHARACTER VARYING	Is a synonym for VARCHAR that complies with ANSI standards.
LVARCHAR	Stores variable length strings that are potentially longer than 255 bytes.
NCHAR	Store single-byte or multibyte text strings of up to 32,767 bytes of text data and supports localized collation of the text data
NVARCHAR	Stores single-byte or multibyte text strings of varying length and up to 255 bytes of text data; it supports localized collation of the text data.
VARCHAR	Stores single-byte or multibyte text strings of varying length and up to 255 bytes of text data; it supports code-set collation of the text data.

The TEXT and CLOB data types also support character data. For more information, see "Large-Object Data Types" on page 1-864.

For more information on individual data types, see the description of the above data types in Chapter 2 of the *Informix Guide to SQL: Reference*.

Fixed- and Varying- Length Data Types

Universal 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 a fixed-length character data types. For example, a CHAR(25) column requires 25 bytes of storage for all its column values so the string "`This is a text string`" uses 25 bytes of storage. Use the ANSI-compliant CHARACTER VARYING data type to specify varying length character data.

A varying-length column requires only the number of bytes that its data uses. The VARCHAR and 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 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 varying data while the LVARCHAR data type can store up to 32 kilobytes of text data.

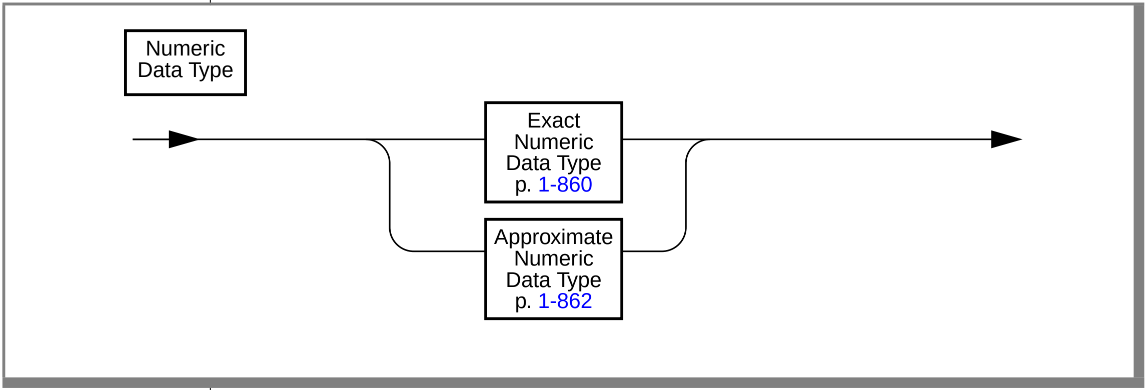
NCHAR and NVARCHAR Data Types

The character data types CHAR, LVARCHAR, and VARCHAR support code-set collation of the text data. That is, 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 the 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.

For more information, see the *Guide to GLS Functionality*.

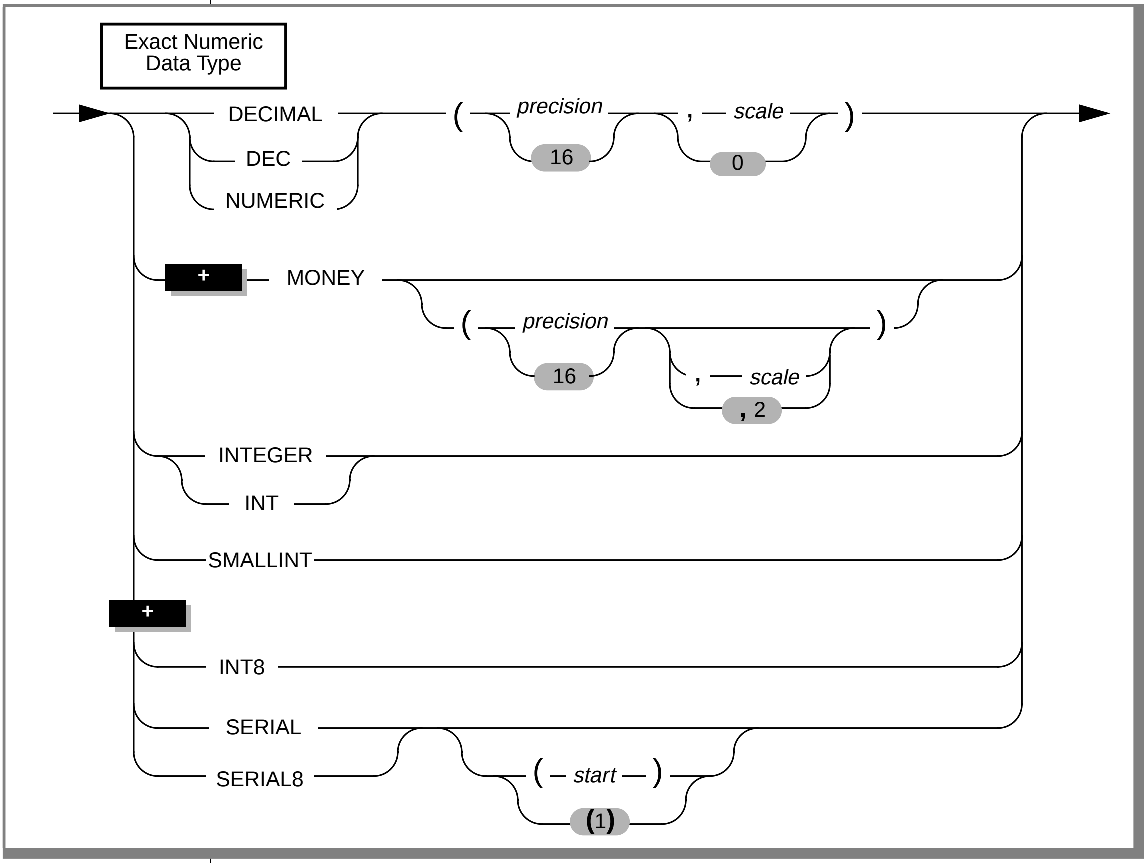
Numeric Data Types


Numeric Data Type
Exact Numeric Data Type p. 1-860
Approximate Numeric Data Type p. 1-862

Numeric data types allow the database server to store numbers such as integers and real numbers in a column. These data types fall into the following two categories:

- Exact numeric data types
- Approximate numeric data types

Exact Numeric Data Types



Element	Purpose	Restrictions	Syntax
precision	Total number of significant digits in a decimal or money data type	You must specify an integer between 1 and 32, inclusive.	Literal Number, p. 1-997
scale	Number of digits to the right of the decimal point	You must specify an integer between 1 and *precision*.	Literal Number, p. 1-997
start	Starting number for values in a SERIAL or SERIAL8 column	For SERIAL columns you must specify a number greater than 0 and less than 2,147,483,647. For SERIAL8 columns you must specify a number greater than 0 and less than 9,223,372,036,854,775,807.	Literal Number, p. 1-997

An exact numeric data type stores a numeric value with a specified precision and scale. The precision of a number 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 that Universal Server supports.

Data Type	Purpose
DEC(p,s)	Is a synonym for DECIMAL(p,s).
DECIMAL(p,s)	Stores fixed-point decimal (real) values in the range. The *p* parameter indicates the precision of the decimal value and the *s* parameter indicates the scale. If no precision is specified, the system default of 16 is used. If no scale is specified, the system default of 0 is used.
INT	Is a 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 values -2,147,483,647 to 2,147,483,647).
INT8	Stores an 8-byte integer value. 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).
MONEY(p,s)	Stores fixed-point currency values. Has the same internal data type as a fixed-point DECIMAL value.

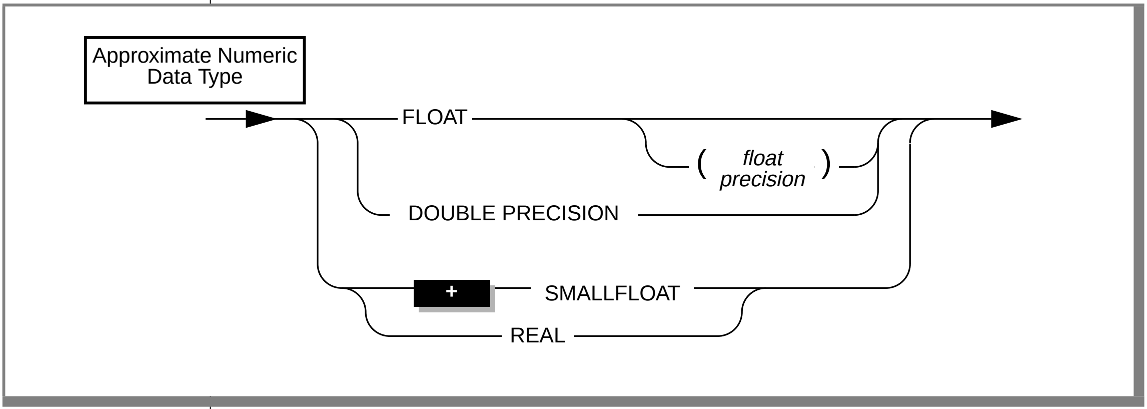
(1 of 2)

Data Type	Purpose
NUMERIC(p,s)	Is an 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	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 (-32,767 to 32,767).

(2 of 2)

For more information, see the entries for these data types in Chapter 2 of the *Informix Guide to SQL: Reference*.

Approximate Numeric Data Types



Element	Purpose	Restrictions	Syntax
float precision	The float precision is ignored.	You must specify a positive integer.	Literal Number, p. 1-997

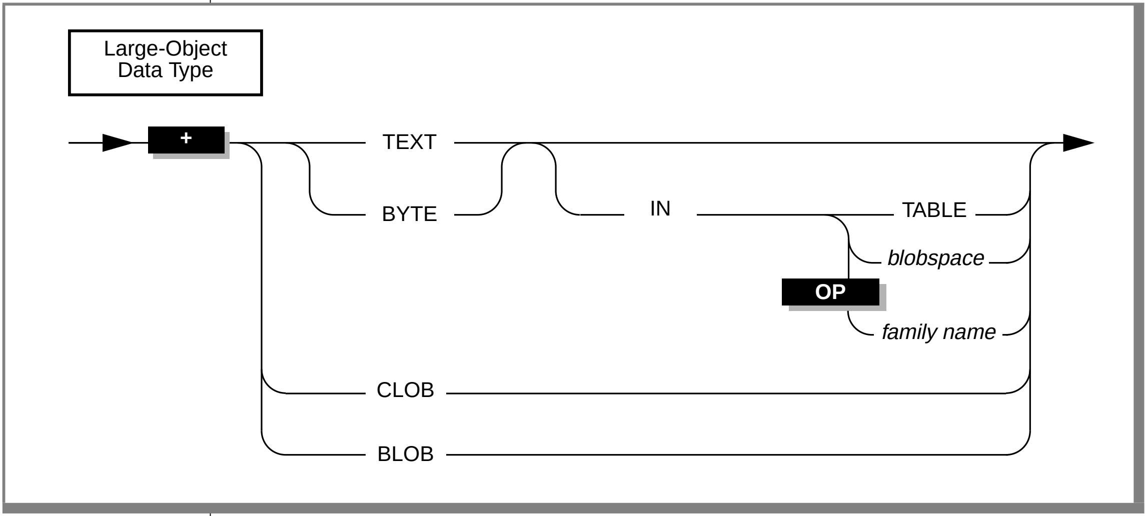
An approximate numeric data type represents numeric values approximately. Use them for very large and very small numbers that can tolerate some degree of rounding during arithmetic operations.

The following table summarizes the approximate numeric data types that Universal Server supports.

Data Type	Purpose
DOUBLE PRECISION	Is an ANSI-compliant synonym for FLOAT.
FLOAT	Stores double-precision floating-point numbers with up to 16 significant digits.
REAL	Is an ANSI-compliant synonym for SMALLFLOAT.
SMALLFLOAT	Stores single-precision floating-point numbers with approximately 8 significant digits.

For more information, see the entries for these data types in Chapter 2 of the *Informix Guide to SQL: Reference*.

Large-Object Data Types



Element	Purpose	Restrictions	Syntax
blobspace	Name of an existing blobspace	The blobspace must exist.	Identifier, p. 1-962
family name	Quoted string constant that specifies a family name or variable name in the optical family	The family name or variable name must exist.	Quoted String, p. 1-1010 For additional information about optical families, see the *INFORMIX-OnLine/ Optical User Manual*.

Large-object data types allow the database server to store extremely large column values such as images and documents independently of the column. These data types fall into the following two categories:

- Simple-large-object data types: TEXT and BYTE
- Smart-large-object data types: CLOB and BLOB

Simple-Large-Object Data Types

A simple-large-object data type stores text or binary data in blobspaces or in tables. (For information on how to create blobspaces, see the *INFORMIX-Universal Server Administrator's Guide*.) The database server can access a simple-large-object value in one piece. These data types are not recoverable.

The following table summarizes the simple-large-object data types that Universal Server supports.

Data Type	Purpose
TEXT	Stores text data of up to 2**31 bytes.
BYTE	Stores text data of up to 2**31 bytes.

For more information, see the entries for these data types in Chapter 2 of the *Informix Guide to SQL: Reference*.

Smart-Large-Object Data Types

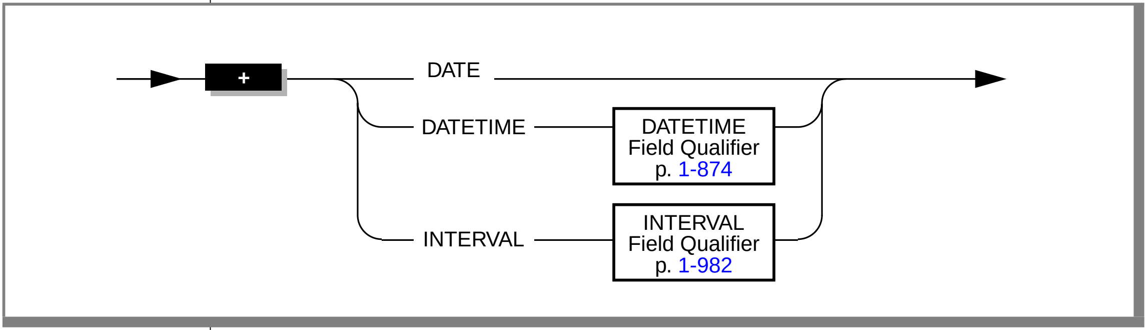
A smart-large-object data type stores text or binary data in sbspaces. (For information about how to create sbspaces, see the *INFORMIX-Universal Server Administrator's Guide*.) The database server can provide random access to a smart-large-object value. That is, it can access any portion of the smart-large-object value. These data types are recoverable.

The following table summarizes the smart-large-object data types that Universal Server supports.

Data Type	Purpose
CLOB	Stores text data of up to 4 terabytes ($4*2^{40}$ bytes).
BLOB	Stores binary data of up to 4 terabytes ($4*2^{40}$ bytes).

For more information, see the entries for these data types in Chapter 2 of *Informix Guide to SQL: Reference*. For information about the SQL functions you use to import, export, and copy smart large objects, see "Smart-Large-Object Functions" on page 1-920 in this manual and Chapter 9 of the *Informix Guide to SQL: Tutorial*.

Time Data Types

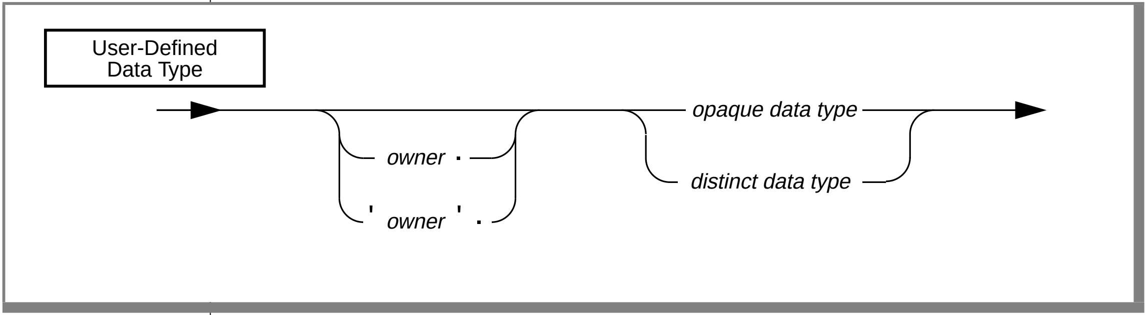

+
DATE
DATETIME
DATETIME Field Qualifier p. 1-874
INTERVAL
INTERVAL Field Qualifier p. 1-982

The time data types allow the database server to store increments of time. The following table summarizes the time data types that Universal Server supports.

Data Type	Purpose
DATE	Stores a date value (*mm/dd/yy*) as a Julian date.
DATETIME	Stores a date and time value (*mm/dd/yy hh:mm:ss.fff*) in an internal format.
INTERVAL	Stores a unit of time such as seconds, hours / minutes, or year / month / day.

For more information, see the entries for these data types in Chapter 2 of the *Informix Guide to SQL: Reference*.

User-Defined Data Type



Element	Purpose	Restrictions	Syntax
opaque data type	The name of the opaque data type	The name must be different from all other data types in the database.	Identifier, p. 1-962
distinct data type	The name of a distinct data type that has the same structure as an existing data type	The name must be different from all other data types in the database.	Identifier, p. 1-962
owner	The user name of the owner of the data type	If you are using an ANSI compliant database, you must enter the *owner.type* name to use a user-defined data type that you do not own. If you put quotation marks around the name you enter in owner, the named is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

A user-defined data type is a data type that a user defines for the database server. Universal Server supports the following categories of user-defined data types:

- Opaque data types
- Distinct data types

The following sections describe the user-defined data types in greater detail.

Opaque Data Types

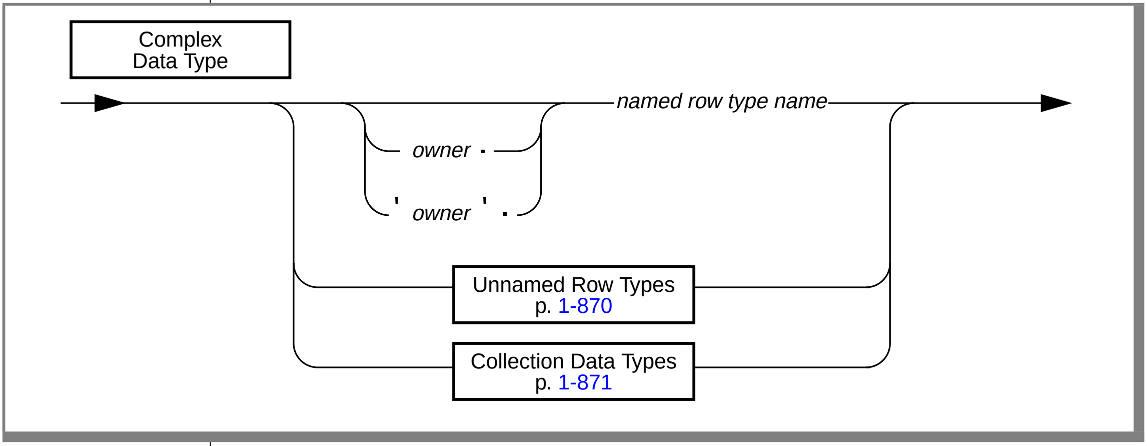
An opaque data type is a user-defined data type that can be used in the same way as a built-in data type. To create an opaque type, you must use the CREATE OPAQUE TYPE statement. Because an opaque type is encapsulated, you create functions to access the individual components of an opaque type. The internal storage details of the type are hidden, or opaque.

For more information, see the CREATE OPAQUE TYPE statement. For complete information about how to create an opaque type and its support functions, see the *Extending INFORMIX-Universal Server: Data Types* manual.

Distinct Data Types

A distinct data type is 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 the CREATE DISTINCT TYPE statement.

Complex Data Type



Element	Purpose	Restrictions	Syntax
named row type name	The name of the named row type	The name must be different from all other data types in the database.	Identifier, p. 1-962 Data type, p. 1-855
owner	The user name of the owner of the data type	If you are using an ANSI-compliant database, you must enter the *owner.type* name to use a named row type that you do not own. If you put quotation marks around the name you enter in owner, the named is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Complex data types are data types that you create from built-in types, opaque types, distinct types, or other complex types. When you create a complex type, you define the components of the complex type. However, unlike an opaque type, a complex type is not encapsulated. You can use SQL to access the individual components of a complex data type.

Universal Server supports the following categories of complex data types:

- Row types
  - Named row types
  - Unnamed row types
- Collection data types
  - SET
  - MULTISET
  - LIST

For a full discussion of complex data types, see Chapter 10 of the *Informix Guide to SQL: Tutorial*.

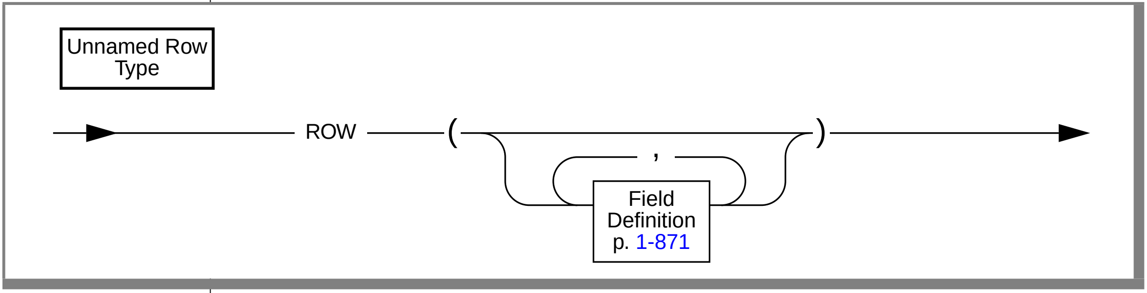
Named Row Types

You can assign a named row type to a table or a column. To use a named row type to create a typed table or define a column, the named row type must already exist. To create a named row type, you use the CREATE ROW TYPE statement. For a description of the CREATE ROW TYPE statement, see page 1-194.

For a complete description of named row types, see Chapter 10 of the *Informix Guide to SQL: Tutorial* and Chapter 2 of the *Informix Guide to SQL: Reference*.

Unnamed Row Types

An unnamed row type is a group of fields that you create with the ROW constructor. You can use an unnamed row type to define a column. The syntax that you use to define a column as an unnamed row type is shown in the following diagram.

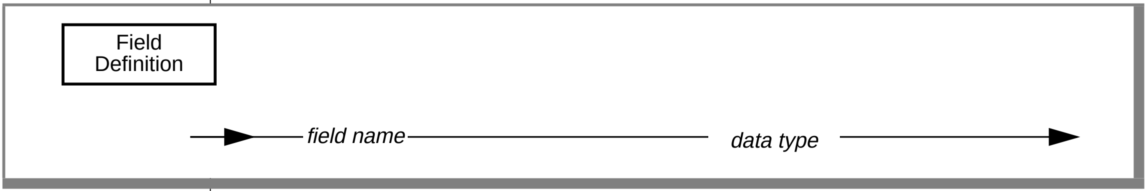


An unnamed row type is identified by its structure. For additional information about unnamed row types and how to create them, see Chapter 10 of the *Informix Guide to SQL: Tutorial* and Chapter 2 in the *Informix Guide to SQL: Reference*.

For the syntax you use to specify row values for an unnamed row type, see "Expression" on page 1-876.

Field Definition

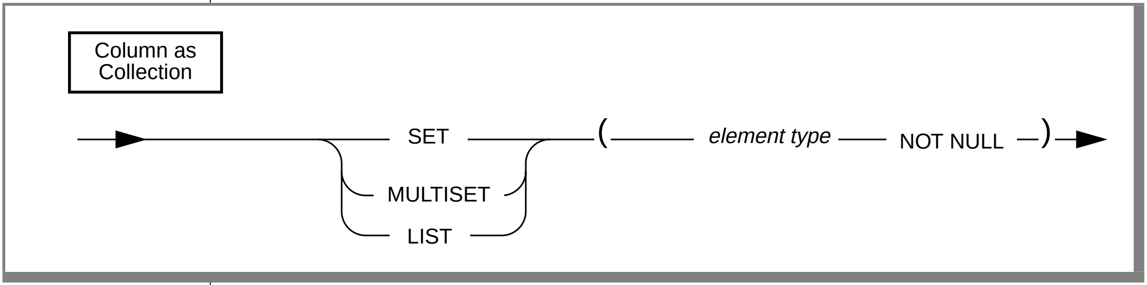
The syntax you use to define the fields of an unnamed row type is shown in the following diagram.



Element	Purpose	Restrictions	Syntax
field name	The name of a field in the row.	The name must be unique within the row type.	Identifier, p. 1-962
data type	The data type of the field.	The field can be any data type except TEXT, BYTE, SERIAL, or SERIAL8.	Data Type, p. 1-855

Collection Data Types

The syntax you use to define a column as a collection type is shown in the following diagram.



Element	Purpose	Restrictions	Syntax
element type	Specifies the data type of the elements of the collection.	The element type can be any data type except TEXT, BYTE, SERIAL, or SERIAL8.	Data Type, p. 1-855

A collection type contains elements that can be of a built-in type, an opaque type, a distinct type, or a row type. A collection type can also contain another collection type within it. You can use a collection type to define a column. The *element type* of a collection specifies the type of data that the collection can contain. For example, if the element type of a collection type is INTEGER, every element in the collection must be of type INTEGER. If the element type of a collection type is a row type, every element in the collection must be of the row type.

To create a collection data type, you must specify the following:

- A type constructor that establishes whether the collection is a SET, MULTISET, or LIST
- An element type that specifies the type of data that the collection can contain

Privileges on a collection type are those of the column. You cannot specify privileges on specific elements of a collection.

For the syntax you use to specify collection values for a collection data type, see the "Literal DATETIME" on page 1-991.

SET Collection Types

A SET is an unordered collection of elements in which each element is unique. You define a column as a SET collection type when you want to store collections whose elements contain no duplicate values and no specific order associated with them.

MULTISET Collection Types

A MULTISET is an unordered collection of elements in which elements can have duplicate values. You 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.

LIST Collection Types

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 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.

References

See the CREATE TABLE statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of complex data types in Chapter 10.

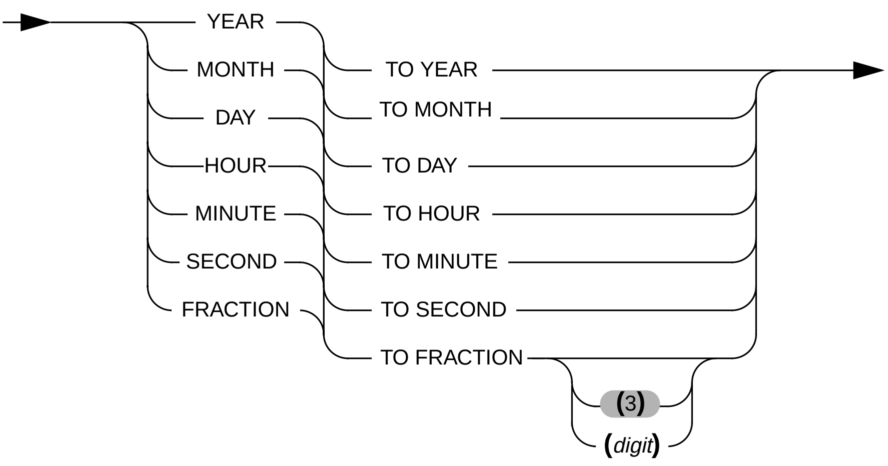
In the *Informix Guide to SQL: Reference*, see the discussions of individual data types in Chapter 2.

In the *Guide to GLS Functionality*, see the discussion of the NCHAR and NVARCHAR data types and the GLS aspects of other character data types.

DATETIME Field Qualifier

A DATETIME field qualifier specifies the largest and smallest unit of time in a DATETIME column or value. Use the DATETIME Field Qualifier segment whenever you see a reference to a DATETIME field qualifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	A single integer that specifies the precision of a decimal fraction of a second. The default precision is 3 digits (a thousandth of a second).	You must specify an integer between 1 and 5, inclusive.	Literal Number, p. 1-997

Usage

Specify the largest unit for the first DATETIME value; after the word TO, specify the smallest unit for the value. The keywords imply that the following values are used in the DATETIME object.

Unit of Time	Meaning
YEAR	Specifies a year, numbered from A.D. 1 to 9999
MONTH	Specifies a month, numbered from 1 to 12
DAY	Specifies a day, numbered from 1 to 31, as appropriate to the month in question
HOUR	Specifies an hour, numbered from 0 (midnight) to 23
MINUTE	Specifies a minute, numbered from 0 to 59
SECOND	Specifies a second, numbered 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).

The following examples show DATETIME qualifiers:

```
DAY TO MINUTE

YEAR TO MINUTE

DAY TO FRACTION(4)

MONTH TO MONTH
```

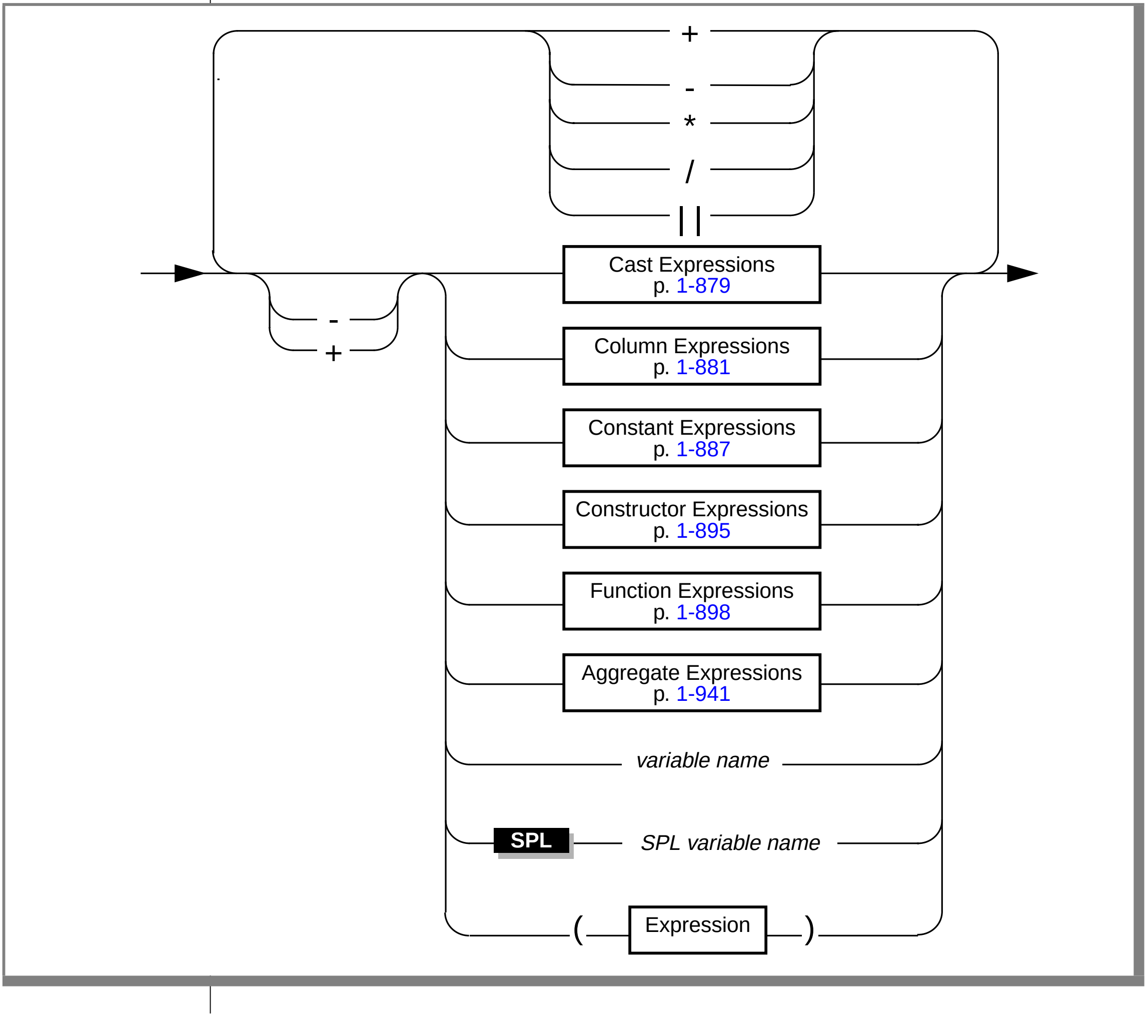
References

In the *Informix Guide to SQL: Reference*, see the DATETIME data type in Chapter 2 for an explanation of the DATETIME field qualifier.

Expression

An expression is one or more pieces of data that is contained in or derived from the database or database server. Use the Expression segment whenever you see a reference to an expression in a syntax diagram.

Syntax



+
-
*
/
||
-
+
Cast Expressions p. 1-879
Column Expressions p. 1-881
Constant Expressions p. 1-887
Constructor Expressions p. 1-895
Function Expressions p. 1-898
Aggregate Expressions p. 1-941
variable name
SPL
SPL variable name
(
Expression
)

Element	Purpose	Restrictions	Syntax
SPL variable name	The name of a variable that is stored in an SPL routine. The value stored in the variable is one of the expression types that is shown in the syntax diagram.	The expression that is stored in *SPL variable name* must conform to the rules for expressions of that type.	Identifier, p. 1-962
variable name	The name of a program variable or host variable. The value stored in the variable is one of the expression types shown in the syntax diagram.	The expression that is stored in *variable name* must conform to the rules for expressions of that type.	Identifier, p. 1-962

Usage

An expression comprises many basic items. Each item is described in the following list.

Expression Item	Purpose
Concatenation operator	Provides the ability to concatenate two string values
Cast operators	Provide the ability to explicit cast from one data type to another
Column expressions	Provide the ability to qualify a column name
Constant expressions	Provide the ability to specify a literal value or a built-in function that returns a built-in value for many data types
Constructor expressions	Provide the ability to dynamically create values for complex data types
Function expressions	Provide the ability to call the built-in functions or user-defined functions

(1 of 2)

Expression Item	Purpose
User-defined functions	Provide the ability to define statement-local variables with a user-defined function
Aggregate functions	Provide the ability to call the built-in aggregate functions
Arithmetic operators	Provide support for arithmetic operations on two items (binary operators) or one item (unary operators) of an expression

(2 of 2)

The following sections describe the syntax of each of these expression items. You can also use SPL variables or host variables in an expression.

Concatenating Expressions

You can use the concatenation operator (||) to concatenate two expressions. The following examples are 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 returned by the TODAY function to the string **Date**.

```
lname[1,3] || zipcode

:file_variable || '.dbg'

'Date:' || TODAY
```

ESQL

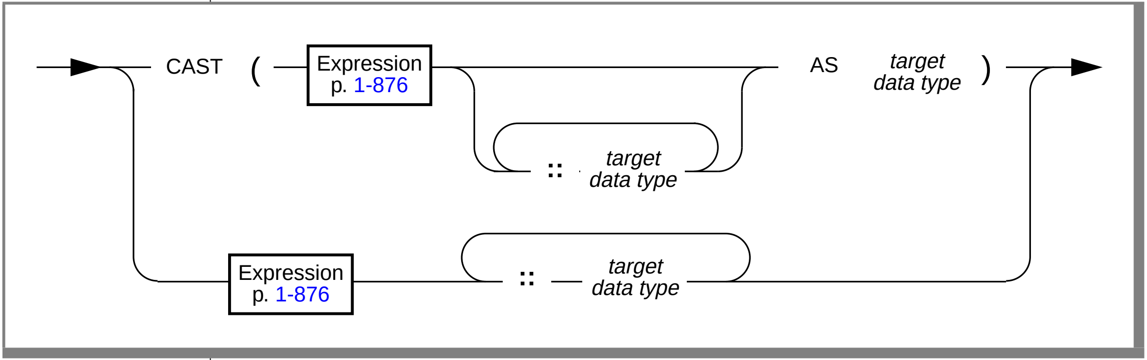
You cannot use the concatenation operator in an embedded-language-only statement. The SQL API-only statements appear in the following list.

ALLOCATE COLLECTION	EXECUTE
ALLOCATE DESCRIPTOR	EXECUTE IMMEDIATE
ALLOCATE ROW	FETCH
CLOSE	FLUSH
CONNECT	FREE
DEALLOCATE COLLECTION	GET DESCRIPTOR
DEALLOCATE DESCRIPTOR	OPEN
DEALLOCATE ROW	PREPARE
DECLARE	PUT
DESCRIBE	SET CONNECTION
DISCONNECT	SET DESCRIPTOR

♦

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 the *Extending INFORMIX-Universal Server: Data Types* manual.

Cast Expressions



Element	Purpose	Restrictions	Syntax
target data type	The data type that results after the cast is applied.	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 type cannot be an unnamed row type or collection data type. An explicit or implicit cast must exist that can convert the data type of the expression to the target data type.	Data type, p. 1-855

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 must 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 has been defined to perform the conversion between two data types, the statement returns an error.

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.

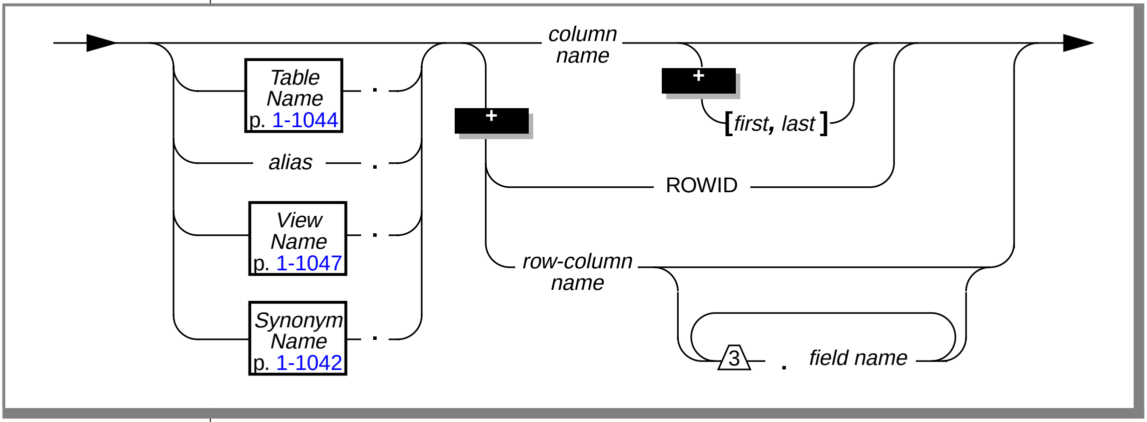
```
CAST (x + y) AS user_type
(x + y)::user_type
```

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
```

Column Expressions

The possible syntax for column expressions is shown in the following diagram.



Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement. This alternative name is established in the FROM clause of the SELECT statement.	The restrictions depend on the clause of the SELECT statement in which *alias* occurs.	Identifier, p. 1-962
column name	The name of the column that you are specifying	The restrictions depend on the statement in which *column name* occurs. Smart large objects cannot be used in many types of expressions. See "Using Smart Large Objects" on page 1-885 for more information.	Identifier, p. 1-962
field name	The name of the row field that you are accessing in the row column	The field must be a member of the row that *row-column name* or *field name* (for nested rows) specifies	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
first	The position of the first character in the portion of the column that you are selecting	The column must be one of the following types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-997
last	The position of the last character in the portion of the column that you are selecting	The column must be one of the following types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-997
row-column name	The name of the row column that you specify	The data type of the column must be a named row type or an unnamed row type.	Identifier, p. 1-962

(2 of 2)

The following examples show column expressions:

```
company
```

```
items.price
```

```
cat_advert [1,15]
```

Use a table or alias name 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 allows you to qualify an SQL identifier with another SQL identifier. 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*

The previous forms of dot notation are called *column projections*.

You can also use dot notation to directly access the fields of a row column, as follows:

```
row_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 accesses field **length** of the **rect** column:

```
SELECT rect.length FROM rectangles
WHERE area = 64
```

If the row definitions are nested, you can specify multiple levels of field names. For example, consider the following two tables:

```
CREATE TABLE tab_b (c ROW(d INTEGER, e CHAR(2));
CREATE TABLE tab_c (d INTEGER);
```

The following SELECT statement references field **d** of row column **c** in table **b**.

```
SELECT *
FROM tab_b,tab_c
WHERE tab_b.c.d = 10
```

When the meaning of a particular identifier is ambiguous, Universal Server uses the following precedence rules to determine which database object the identifier specifies in a dot notation of *name1.name2.name3.name4*:

1. schema *name1*.table *name2*.column *name3*.field *name4*
2. table *name1*.column *name2*.field *name3*.field *name4*
3. column *name1*.field *name2*.field *name3*.field *name4*

For more information about precedence rules and how to use dot notation with row columns, see Chapter 12 of the *Informix Guide to SQL: Tutorial*.

Using Subscripts on Character Columns

You can use subscripts on CHAR, VARCHAR, BYTE, and TEXT columns. The subscripts indicate the starting and ending character positions that are contained in the expression. Together the column subscripts define a column substring. The column substring is 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]
```

GLS

Column subscripting also works on NCHAR and NVARCHAR columns. For information on the GLS aspects of column subscripts and substrings, see the *Guide to GLS Functionality*. ♦

Using Rowids

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. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the **rowid** column.

The following examples show possible uses of 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
```

In Universal Server only, the last SELECT statement 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 ROWID keyword in the select list of a query that contains an aggregate function.

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 accept a handle value for a BLOB or CLOB to be inserted or updated.

  INSERT and UPDATE 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 column value.

To access the data of a smart-large-object column, you must use one of the following application programming interfaces (APIs):

- From within an INFORMIX-ESQL/C program, use the ESQL/C library functions that access smart large objects.

  For more information, see the *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 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 across several columns. This storage arrangement reduces the amount of disk space that the CLOB or BLOB data takes. However, 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.

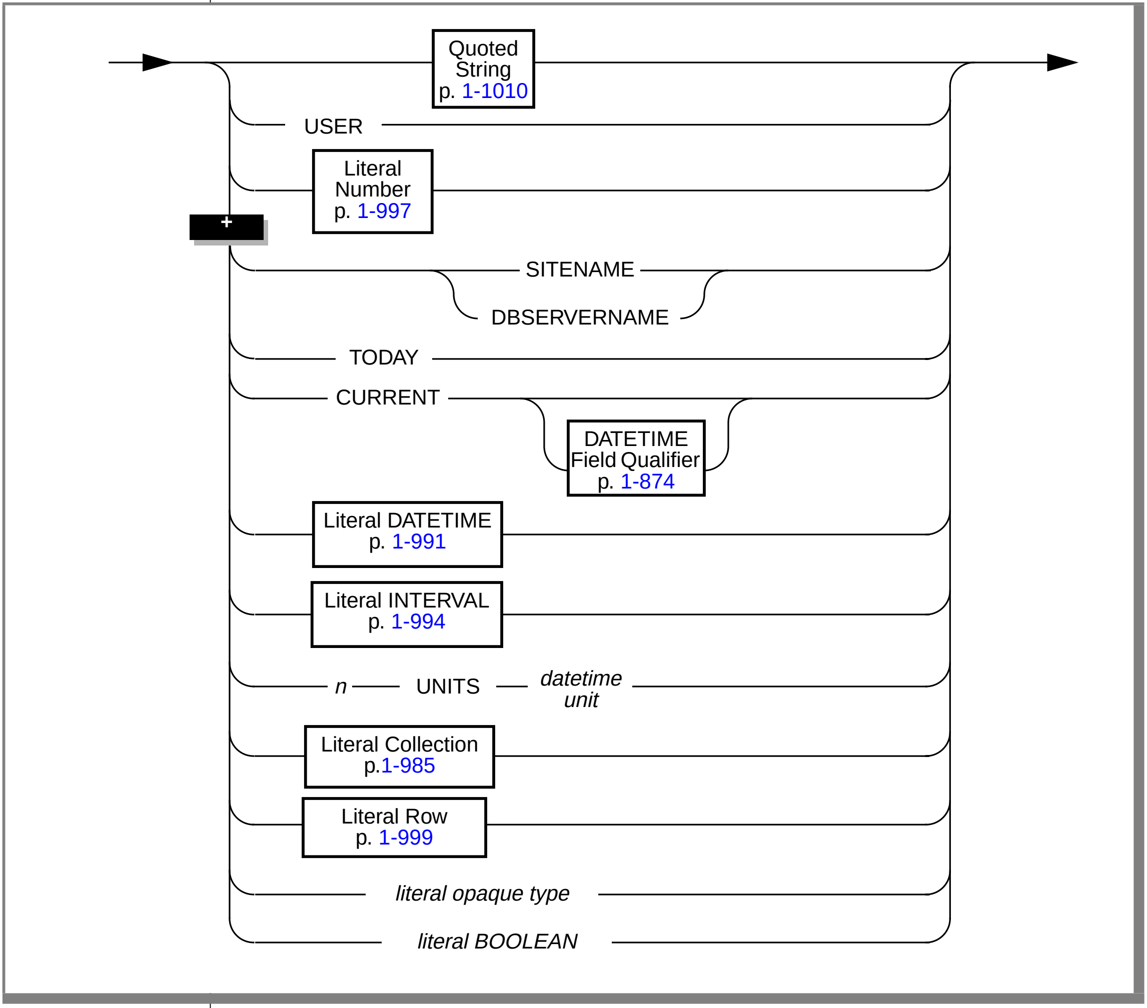
  If two columns share the same smart-large-object value, the data might be locked by either column that needs to access it.

- 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 SQL functions LOTOFILE, FILETOCLOB, and FILETOBLOB to access smart-large-object values. For more information on these functions, see "Smart-Large-Object Functions" on page 1-920. For more information on the BLOB and CLOB data types, see Chapter 2 of the *Informix Guide to SQL: Reference*.

Constant Expressions

The following diagram shows the possible syntax for constant expressions.



Element	Purpose	Restrictions	Syntax
datetime unit	One of the units that is used to specify an interval precision; that is, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION. If the unit is YEAR, the expression is a year-month interval; otherwise, the expression is a day-time interval.	The datetime unit must be one of the keywords that is listed in the Purpose column. You can enter the keyword in uppercase or lowercase letters. You cannot put quotation marks around the keyword.	See the Restrictions column.
n	A literal number that you use to specify the number of datetime units. See "The UNITS Keyword" on page 1-894 for more information on this parameter.	If *n* is not an integer, it is rounded down to the nearest whole number when it is used. The value that you specify for *n* must be appropriate for the datetime unit that you choose.	Literal Number, p. 1-997,
literal opaque type	The literal representation for an opaque data type	The literal must be recognized by the input support function of the associated opaque type.	Defined by the developer of the opaque type.
literal BOOLEAN	The literal representation of a BOOLEAN value	A literal BOOLEAN can be only 't' (TRUE) or 'f'(FALSE).	Quoted string, p. 1-1010.

The following examples show constant expressions:

```
DBSERVERNAME

TODAY

'His first name is'

CURRENT YEAR TO DAY

INTERVAL (4 10:05) DAY TO MINUTE

DATETIME (4 10:05) DAY TO MINUTE

5 UNITS YEAR

{2, 3, 4}
```

The following list provides references for further information:

- For quoted strings as expressions, see "Quoted String as an Expression".
- For the USER function in an expression, see "USER Function" on page 1-890.
- For the SITENAME and DBSERVERNAME functions in an expression, refer to "SITENAME and DBSERVERNAME Functions" on page 1-890.
- For literal numbers as expressions, see "Literal Number as an Expression" on page 1-891.
- For the TODAY function in an expression, see "TODAY Function" on page 1-891.
- For the CURRENT function in an expression, see "CURRENT Function" on page 1-892.
- For literal DATETIME as an expression, see "Literal DATETIME as an Expression" on page 1-893.
- For literal INTERVAL as an expression, see "Literal INTERVAL as an Expression" on page 1-893.
- For the UNITS keyword in an expression, see "The UNITS Keyword" on page 1-894.
- For literal collections as expressions, see "Literal Collection as an Expression" on page 1-894.

Quoted String as an Expression

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 = '1993-1-1 10:45'
    WHERE customer_num = 120 AND call_code = 'B'
```

USER Function

The USER function returns a string that contains the login name of the current user (that is, the person running the process). The following statements show how you might use the USER function:

```
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 function does not change the case of a user ID. If you use USER in an expression and the present user is **Robertm**, the USER function returns **Robertm**, not **robertm**. If you specify user as the default value for a column, the column must be CHAR, VARCHAR, NCHAR, or NVARCHAR data type, and it must be at least eight characters long.

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 function returns, with respect to case. ♦

SITENAME and DBSERVERNAME Functions

The SITENAME and DBSERVERNAME functions return the database server name, as defined in the ONCONFIG file for the Universal Server installation where the current database resides or as specified in the **INFORMIXSERVER** environment variable. The two function names, SITENAME and DBSERVERNAME, are synonymous.

You can use the DBSERVERNAME function 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 be CHAR, VARCHAR, NCHAR, or NVARCHAR data type and must be at least 18 characters long.

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
```

Literal Number as an Expression

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
```

TODAY Function

Use the TODAY function 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 function 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 Function

The CURRENT function 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 qualifiers are YEAR TO FRACTION(3). You can use the CURRENT function in any context in which you can use a literal DATETIME (see page 1-991). 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 the CURRENT function 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 calls to CURRENT from an EXECUTE PROCEDURE statement return the value when the stored procedure starts.

The CURRENT function 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.

The CURRENT function might not execute in the physical order in which it appears in a statement. You should not use the CURRENT function 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, the CURRENT function returns a zero for the FRACTION field.

In the following example, the first statement uses the CURRENT function in a WHERE condition. The second statement uses the CURRENT function as the input for the DAY function. The last query selects rows whose **call_dtime** value is within a range from the beginning of 1993 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 '1993-1-1 00:00:00' AND CURRENT
```

Literal DATETIME as an Expression

The following examples show literal DATETIME as an expression:

```
SELECT DATETIME (1993-12-6) YEAR TO DAY FROM customer

UPDATE cust_calls SET res_dtime = DATETIME (1992-07-07 10:40)
           YEAR TO MINUTE
    WHERE customer_num = 110
    AND call_dtime = DATETIME (1992-07-07 10:24) YEAR TO MINUTE

SELECT * FROM cust_calls
    WHERE call_dtime
    = DATETIME (1995-12-25 00:00:00) YEAR TO SECOND
```

Literal INTERVAL as an Expression

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.

The UNITS Keyword

The UNITS keyword enables you to display a simple interval or increase or decrease a specific interval or datetime value.

If *n* is not an integer, it is rounded down to the nearest whole number when it is used.

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'
```

Literal Collection as an Expression

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 DATETIME” on page 1-991.

Literal Row as an Expression

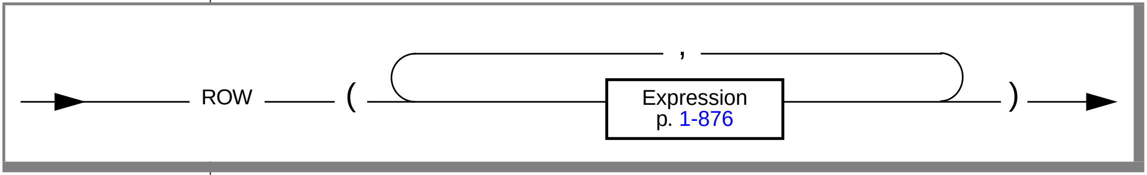
The following examples show literal collections as expressions:

```
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 1-999.

Constructor Expressions

A *constructor* is a function that the database server uses to create an instance of a particular data type. Universal Server supports a ROW constructor. The syntax for expressions that use a ROW constructor is shown in the following diagram.



You can use any kind of expression with a ROW constructor, including literals, functions, and variables. The following examples show row expressions:

```
ROW(5, 6.77, 'HMO')

ROW(col1.lname, 45000)

ROW('john davis', TODAY)

ROW(USER, SITENAME)
```

Using ROW Constructors

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 column.To create a value for either a named row type or unnamed row type, you must do the following:

- Begin the expression with the ROW keyword.
- Specify a value for each field of the row type.
- Enclosed 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.

The following statement uses 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 shown in the following examples:

```
ROW(4,5)::row_t
CAST (ROW(3,4) AS row_t)
```

Use a ROW constructor anytime you want to generate a row type value. In the following example, a ROW constructor specifies a row type value that is cast as type **person_t**:

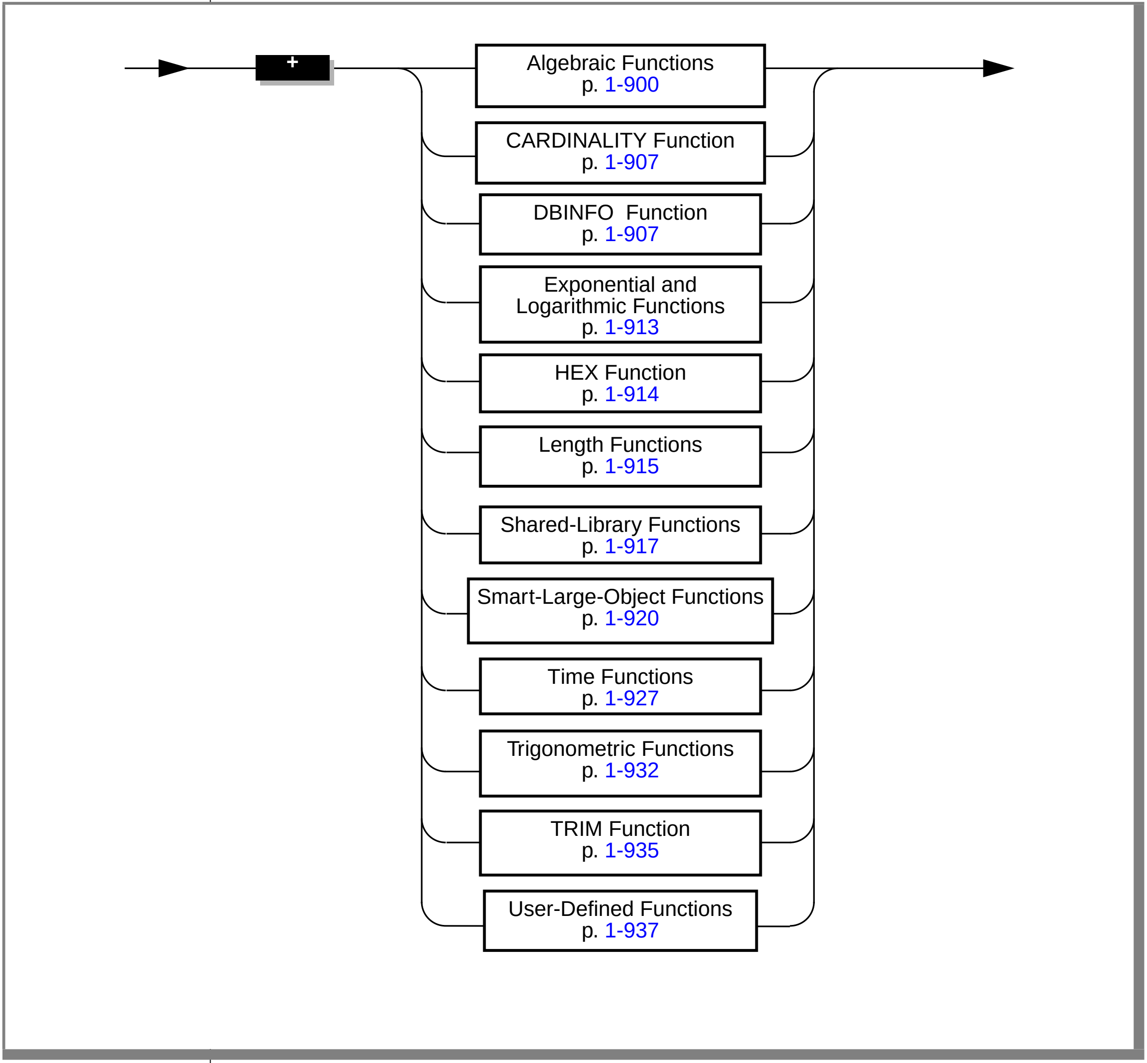
```
SELECT *
FROM person_tab
WHERE col1 = ROW('charlie','hunter')::person_t
```

See the INSERT, UPDATE, and SELECT statements in this manual. See the CREATE ROW TYPE statement for information on named row types.

In the *Informix Guide to SQL: Reference*, see the ROW data type in Chapter 2 for information on unnamed row types. In the *Informix Guide to SQL: Tutorial*, see Chapter 10 for information on named row types and unnamed row types.

Function Expressions

A function expression can call built-in functions or user-defined functions, as the following diagram shows.

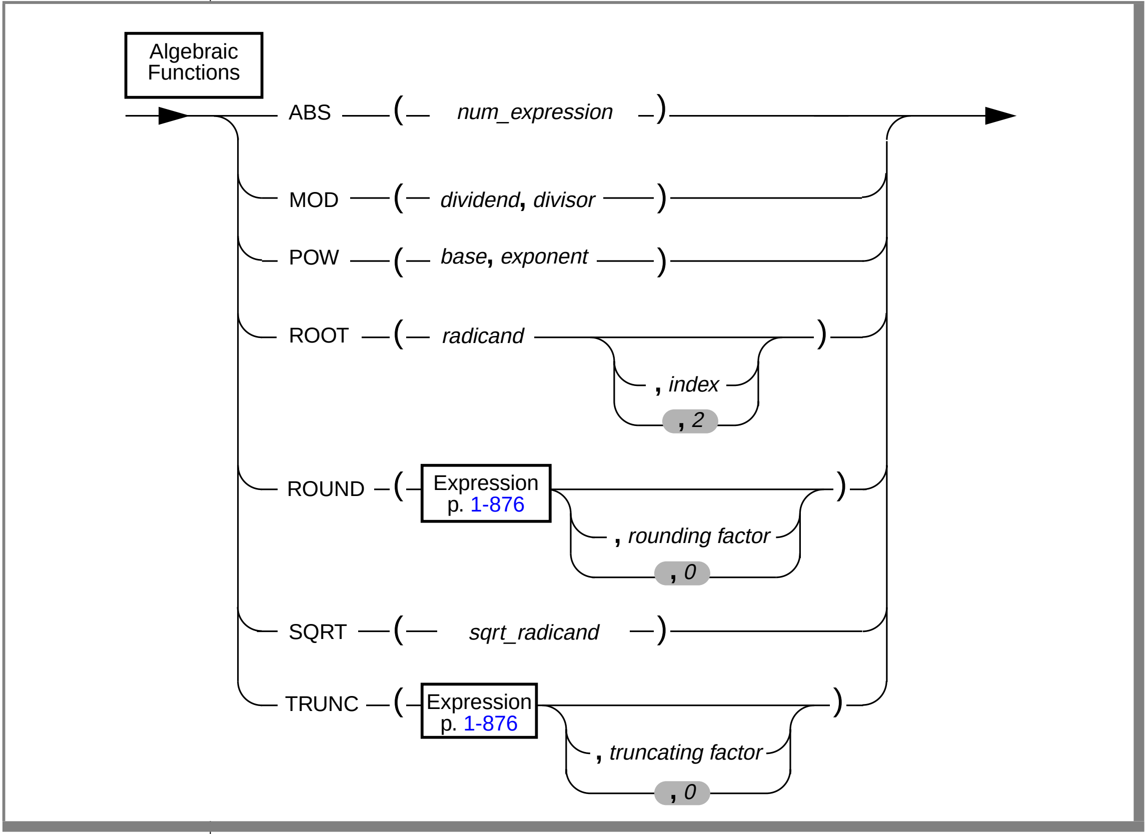


The following examples show function expressions:

```
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 (4,3)

MOD (10,3)
```

Algebraic Functions

An algebraic function takes one or more arguments, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
base	A value to be raised to the power that is specified in *exponent*. The *base* value is the first argument that is supplied to the POW() function. See "POW() Function" on page 1-904 for further information on *base*.	You can enter in *base* any real number or any expression that evaluates to a real number.	Expression, p. 1-876
dividend	A value to be divided by the value in *divisor*. The *dividend* value is the first argument supplied to the MOD() function. See "MOD() Function" on page 1-903 for further information on *dividend*.	You can enter in *dividend* any real number or any expression that evaluates to a real number.	Expression, p. 1-876
divisor	The value by which the value in *dividend* is to be divided. The *divisor* value is the second argument that is supplied to the MOD() function. See "MOD() Function" on page 1-903 for further information on *divisor*	You can enter in *divisor* any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 1-876
exponent	The power to which the value that is specified in *base* is to be raised. The *exponent* value is the second argument that is supplied to the POW() function. See "POW() Function" on page 1-904 for further information on *exponent*.	You can enter in *exponent* any real number or any expression that evaluates to a real number.	Expression, p. 1-876
index	The type of root to be returned, where 2 represents square root, 3 represents cube root, and so on. The *index* value is the second argument that is supplied to the ROOT() function. The default value of *index* is 2. See "ROOT() Function" on page 1-904 for further information on *index*.	You can enter in *index* any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 1-876

(1 of 3)

Element	Purpose	Restrictions	Syntax
num_expression	A numeric expression for which an absolute value is to be returned. The expression serves as the argument for the ABS() function. See "ABS() Function" on page 1-903 for further information on *num_expression*.	The value of *num_expression* can be any real number.	Expression, p. 1-876
radicand	An expression whose root value is to be returned. The *radicand* value is the first argument that is supplied to the ROOT() function. See "ROOT() Function" on page 1-904 for further information on *radicand*.	You can enter in *radicand* any real number or any expression that evaluates to a real number.	Expression, p. 1-876
rounding factor	The number of digits to which a numeric expression is to be rounded. The *rounding factor* value is the second argument that is supplied to the ROUND() function. The default value of *rounding factor* is zero. This default means that the numeric expression is rounded to zero digits or the ones place. See "ROUND() Function" on page 1-904 for further information on *rounding factor*.	The value you specify in *rounding factor* must be an integer between +32 and -32, inclusive. See "ROUND() Function" on page 1-904 for further information on this restriction.	Literal Number, p. 1-997
sqrt_radicand	An expression whose square root value is to be returned. The *sqrt_radicand* value is the argument that is supplied to the SQRT() function. See "SQRT() Function" on page 1-905 for further information on *sqrt_radicand*.	You can enter in *sqrt_radicand* any real number or any expression that evaluates to a real number.	Expression, p. 1-876

(2 of 3)

Element	Purpose	Restrictions	Syntax
truncating factor	The position to which a numeric expression is to be truncated. The *truncating factor* value is the second argument that is supplied to the TRUNC() function.The default value of *truncating factor* is zero. This default means that the numeric expression is truncated to zero digits or the ones place. See "TRUNC() Function" on page 1-906 for further information on *truncating factor.*	The value you specify in *truncating factor* must be an integer between +32 and -32, inclusive. See "TRUNC() Function" on page 1-906 for further information on this restriction.	Literal Number, p. 1-997

(3 of 3)

ABS() Function

The ABS() function gives the absolute value for a given expression. The function requires a single numeric argument. The value returned is the same as the argument type. The following example shows all orders of more than $20 paid in cash (+) or store credit (-). The **stores7** database does not contain any negative balances; however, 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 modulus or remainder value for two numeric expressions. You provide integer expressions for the dividend and divisor. The divisor cannot be 0.

In earlier Informix products, the MOD() function returned an INT value. However, in Universal Server, the MOD() function returns an INT8 value.

The following example uses a 30-day billing cycle to determine how far today is into the 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 return type is FLOAT. The following example returns all the information for circles whose areas ($\pi\ r^2$) are less than 1,000 square units:

```
SELECT * FROM circles WHERE (3.1417 * POW(radius,2)) < 1000
```

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 FLOAT. The first SELECT statement in the following example takes the square root of the expression. The second SELECT statement 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.

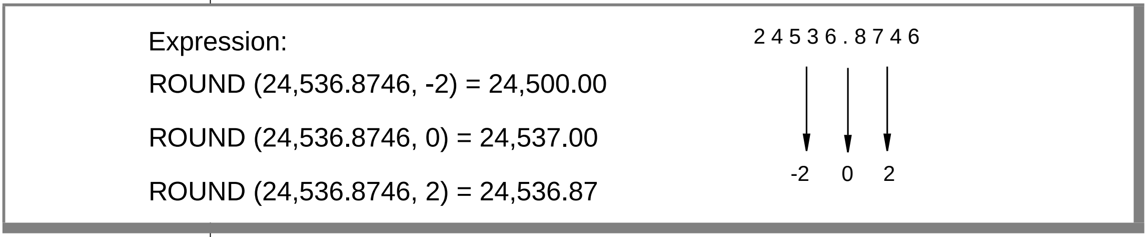
ROUND() Function

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 ones place. The digit limitation 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 1-3 shows.

Figure 1-3
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 following 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 one'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 of the decimal point, as Figure 1-4 shows.

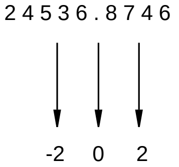
Figure 1-4
TRUNC() Function

Expression:

TRUNC (24536.8746, -2) =24500

TRUNC (24536.8746, 0) = 24536

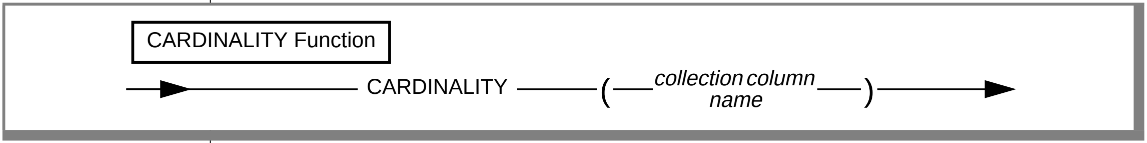
TRUNC (24536.8746, 2) = 24536.87



If you use a MONEY data type as the argument for the TRUNC() function and you truncate to zero places, the .00 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
```

CARDINALITY() Function



Element	Purpose	Restrictions	Syntax
collection column name	The name of an existing collection column	You must specify an integer or an expression that evaluates to an integer.	Expression, p. 1-876

The CARDINALITY() function returns the number of elements in a collection column (SET, MULTISET, LIST). 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
```

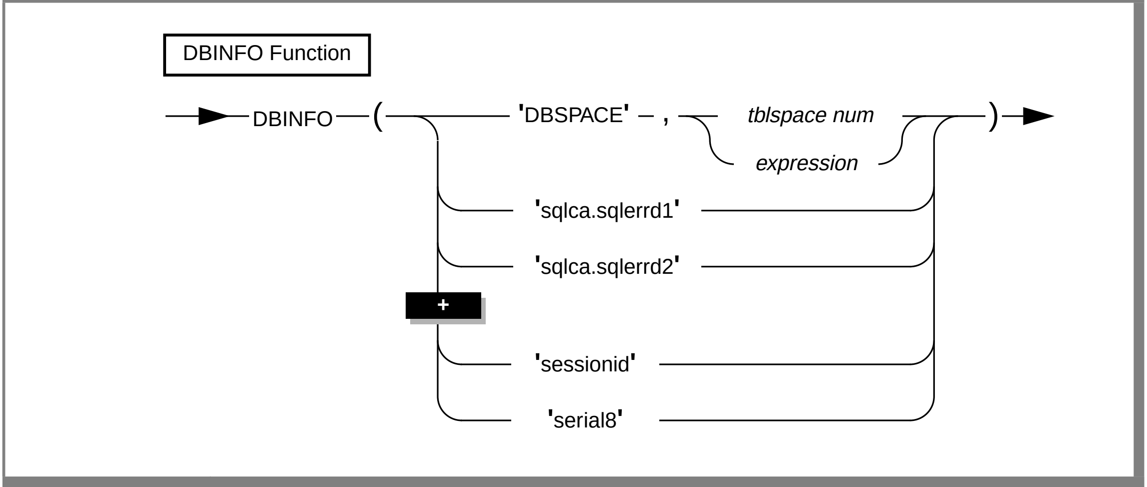
If the collection contains duplicate elements, CARDINALITY() counts each individual element.

DBINFO() Function

Use the DBINFO() function for any of the following purposes:

- To locate the name of a dbspace corresponding to a tblspace number or expression
- To find out the last SERIAL value inserted in a table
- To find out the number of rows processed by selects, inserts, deletes, updates, and execute procedure statements
- To find out the session ID of the current session
- To find out the last SERIAL8 value inserted in a table

You can use the DBINFO() function anywhere within SQL statements and within routines.



Element	Purpose	Restrictions	Syntax
expression	An expression that evaluates to *tblspace num*	The expression can contain procedure variables, host variables, column names, or subqueries, but it must evaluate to a numeric value.	Expression, p. 1-876
tblspace num	The tblspace number (partition number) of a table. The DBSPACE option of the DBINFO() function returns the name of the dbspace that corresponds to the specified tblspace number.	The specified tblspace number must exist. That is, it must occur in the **partnum** column of the **systables** table for the database.	Literal Number, p. 1-997

Using the 'DBSPACE' Option

The 'DBSPACE' option returns a character string that contains the name of the dbspace corresponding to a tblspace number. You must supply an additional parameter, either *tblspace num* or an expression that evaluates to *tblspace num*. The following 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;
```

If the statement returns a partition number of 16777289, you insert that value into the second argument to find which dbspace contains the **customer** table, as shown in the following example:

```
SELECT DBINFO ('DBSPACE', 16777289) FROM systables;
```

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 an INSERT statement that inserts a SERIAL value.

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 page 1-907.*

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 FUNCTION and EXECUTE PROCEDURE statements processed. To ensure valid results, use this option after SELECT and EXECUTE PROCEDURE statements have completed executing. In addition, if you use this option within cursors, make sure that all rows are fetched before the cursors are closed to ensure valid results.

The following example shows a stored procedure that uses the 'sqlca.sqlerrd2' option to determine the number of rows that are deleted from a table:

```
CREATE PROCEDURE del_rows (pnumb int)
RETURNING int;

DEFINE nrows int;

DELETE FROM sec_tab WHERE partnum=pnumb;
LET nrows = DBINFO('sqlca.sqlerrd2');
RETURN nrows;

END PROCEDURE
```

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 Universal 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, the **sysmaster** database, and the SMI tables, see the *INFORMIX-Universal Server Administrator's Guide*.

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'
```

The following table shows the result of this query.

my_sessionid
14

In the preceding example, the SELECT statement queries against the **systables** system catalog table. However, you can 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
```

The following table shows the result of this query.

user_sessionid
14

You can use the DBINFO() function not only in SQL statements but also in stored procedures. The following example shows a stored procedure that returns the value of the current session ID to the calling program or procedure:

```
CREATE PROCEDURE get_sess()
RETURNING INT;
RETURN DBINFO('sessionid');
END PROCEDURE;
```

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 page 1-907.*

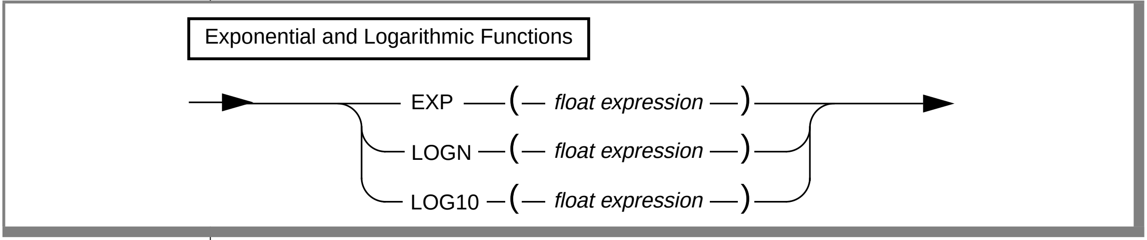
The following example uses the 'serial8' option:

```
.
.
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.

Exponential and Logarithmic Functions

Exponential and logarithmic functions take at least one argument. The return type is FLOAT. The following example shows exponential and logarithmic functions.



Element	Purpose	Restrictions	Syntax
float expression	An expression that serves as an argument to the EXP(), LOGN(), or LOG10() functions. For information on the meaning of *float expression* in these functions, see the individual heading for each function on the following pages.	The domain of the expression is the set of real numbers, and the range of the expression is the set of positive real numbers.	Expression, p. 1-876

EXP() Function

The EXP() function returns the exponential value of two numeric expressions. You provide a constant and float expression in the form `e(n)=en`. The following example returns the exponent of 3 for each row of the **angles** table:

```
SELECT EXP(3) FROM angles
```

LOGN() Function

The LOGN() function returns the natural log of a numeric expression. The logarithmic value is the inverse of the exponential value. The following SELECT statement returns the natural log of population for each row of the **history** table:

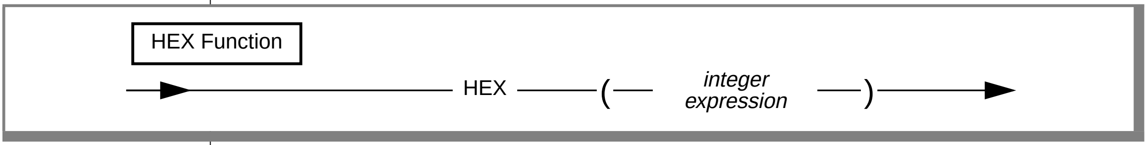
```
SELECT LOGN(population) FROM history WHERE country='US'
    ORDER BY date
```

LOG10() Function

The LOG10() function returns the log of a value to the 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
```

HEX() Function



Element	Purpose	Restrictions	Syntax
integer expression	A numeric expression for which you want to know the hexadecimal equivalent	You must specify an integer or an expression that evaluates to an integer.	Expression, p. 1-876

The HEX() function returns the hexadecimal encoding of an integer expression. The following 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. (See Chapter 1 of the *Informix Guide to SQL: Reference* for more information about encoded information.)

```
SELECT colname, coltype, HEX(collength)
    FROM syscolumns C, systables T
    WHERE C.tabid = T.tabid AND T.tabname = 'orders'
```

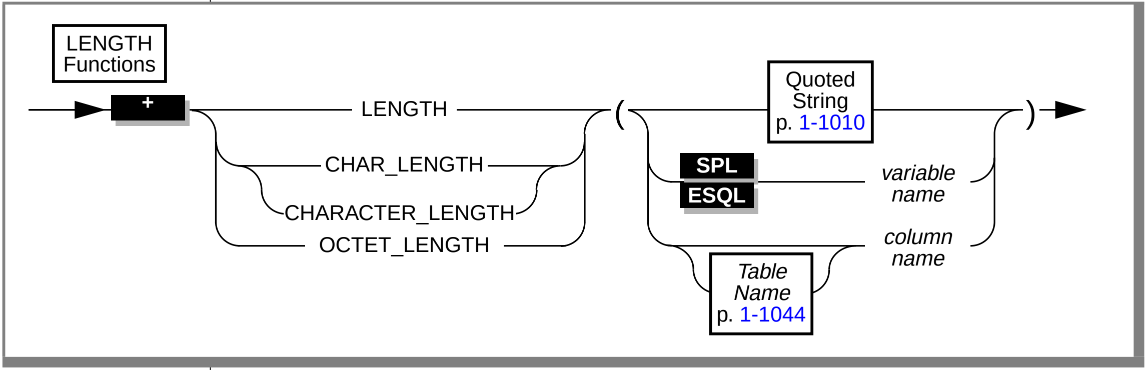
The following example lists the names of all the tables in the current database and their corresponding tblspace number in hexadecimal format. This example is particularly useful because the two most significant bytes in the hexadecimal number constitute the dbspace number. They are used to identify the table in **oncheck** output.

```
SELECT tabname, HEX(partnum) FROM systables
```

The HEX() function can operate on an expression, as the following example shows:

```
SELECT HEX(order_num + 1) FROM orders
```

Length Functions



Element	Purpose	Restrictions	Syntax
column name	The name of a column in the specified table.	The column must have a character data type.	Identifier, p. 1-962
variable name	A host variable or procedure variable that contains a character string.	The host variable or procedure variable must have a character data type.	The name of the host variable must conform to language-specific rules for variable names. .

You can use length functions to determine the length of a column, string, or variable. The length functions are LENGTH(), OCTET_LENGTH(), and CHAR_LENGTH(). Each of these functions has a distinct purpose.

LENGTH() Function

The LENGTH() function returns the number of bytes in a character column, not including any trailing spaces. With TEXT or BYTE columns, the LENGTH() function returns the full number of bytes in the column, including trailing spaces.

The following 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
```

ESQL

You can use the LENGTH() function to return the length of a character variable. ♦

GLS

For information on GLS aspects of the LENGTH function, see the *Guide to GLS Functionality*.

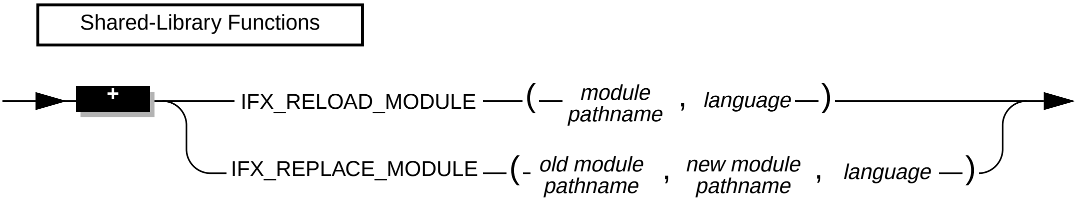
OCTET_LENGTH() Function

The OCTET_LENGTH() function returns the number of bytes in a character column, including any trailing spaces. See the *Guide to GLS Functionality* for a discussion of the OCTET_LENGTH() function.

CHAR_LENGTH() Function

The CHAR_LENGTH() function returns the number of characters (not bytes) in a character column. See the *Guide to GLS Functionality* for a discussion of the CHAR_LENGTH() function. ♦

Shared-Library Functions



Element	Purpose	Restrictions	Syntax
language	A character string that specifies the language of the user-defined routines in the *module pathname* shared library.	Must be either "c" (for external routines) or "spl" (for SPL routines).	Quoted String, p. 1-1010
module pathname	The full pathname of the shared library that you want to reload.	The shared library must exist with the specified pathname.	Quoted String, p. 1-1010
new module pathname	The full pathname of the new shared library to replace the shared library that *old module pathname* specifies.	The shared library must exist with the specified pathname.	Quoted String, p. 1-1010
old module pathname	The full pathname of the shared library to replace with the shared library that *new module path* specifies.	The shared library must exist with the specified pathname.	Quoted String, p. 1-1010

To execute an external routine, the database server loads the shared library that contains this function into shared memory. A shared-library function allows you to update the copy of a shared library that the database server has loaded into memory.

IFX_RELOAD_MODULE Function

The IFX_RELOAD_MODULE function reloads a loaded shared library into the shared memory for the database server. This function replaces the existing version with a new version that has the same name and location. The function returns an integer value to indicate the status of the update, as follows:

- Zero (0) to indicate success
- A negative integer to indicate an error

For example, to reload a new version of the **circle.so** shared library that resides in the **/usr/app/opaque_types** directory, you can use the EXECUTE FUNCTION statement to execute the IFX_RELOAD_MODULE function, as follows:

```
EXECUTE FUNCTION ifx_reload_module("/usr/apps/opaque_types/circle.so",
    "c")
```

E/C

To execute the IFX_RELOAD_MODULE function in an INFORMIX-ESQL/C application, you must associate the function with a cursor. ♦

For more information on how to use IFX_RELOAD_MODULE to update a shared library, see the chapter on how to design a user-defined routine in the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

IFX_REPLACE_MODULE Function

The IFX_REPLACE_MODULE function replaces a loaded shared library with a new version that has a different name or location. The function returns an integer value to indicate the status of the update, as follows:

- Zero (0) to indicate success
- A negative integer to indicate an error

For example, to replace the **circle.so** shared library that resides in the **/usr/app/opaque_types** directory with one that resides in the **/usr/app/shared_libs** directory, you can use the following EXECUTE FUNCTION statement to execute the IFX_REPLACE_MODULE:

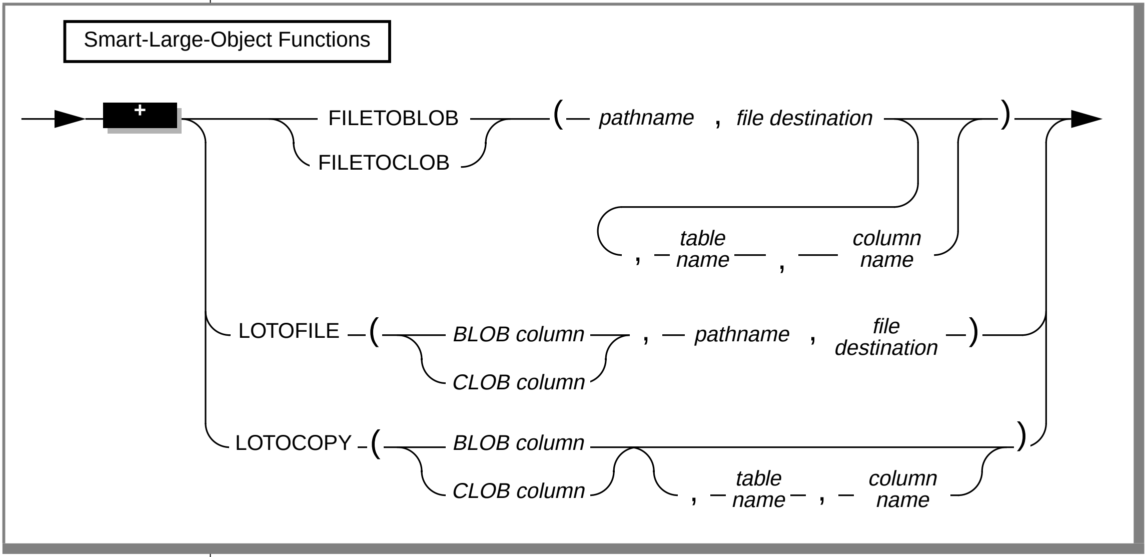
```
EXECUTE FUNCTION ifx_replace_module("/usr/apps/opaque_types/circle.so",
   "/usr/apps/shared_libs/circle.so", "c")
```

E/C

To execute the IFX_REPLACE_MODULE function in an INFORMIX-ESQL/C appliation, you must associate the function with a cursor. ♦

For more information on how to use IFX_REPLACE_MODULE to update a shared library, see the chapter on how to design a user-defined routine in the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

Smart-Large-Object Functions



Element	Purpose	Restrictions	Syntax
BLOB column	The name of a column of type BLOB	If you specify the *table name* and *column name*, a BLOB column must exist in that table.	Identifier, p. 1-962
CLOB column	The name of a column of type CLOB	If you specify the *table name* and *column name*, a CLOB column must exist in that table.	Identifier, p. 1-962
column name	The name of a column within *table name* whose storage characteristics are used for the copy of the BLOB or CLOB value	This column must have CLOB or BLOB as its data type.	Identifier, p. 1-962

(1 of 2)

Element	Purpose	Restrictions	Syntax
file destination	The string "`server`" or "`client`" to indicate the computer on which to put or get the smart large object	The only valid values are the strings "`server`" or "`client`".	Quoted String, p. 1-1010
pathname	The directory path and filename to locate the smart large object. See the example on page 1-923.	The pathname must exist on the computer designated by *file destination*.	Quoted String, p. 1-1010
table name	The name of the table that contains *column name*, whose storage characteristics are used for the copy of the BLOB or CLOB value	The table must exist in the database and it must contain a CLOB or BLOB column.	Identifier, p. 1-962

(2 of 2)

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 name* and *column name* parameters are optional:

- If you omit *table name* and *column name*, FILETOBLOB creates a BLOB value with the system-specified storage defaults and 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 *INFORMIX-Universal Server Administrator's Guide*.
- If you specify a *table name* and *column name*, 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, the FILETOCLOB function returns a handle value to the new CLOB value. Neither of these functions actually store the smart-large-object value into a column in the database. 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 **haven.rsm** file:

```
INSERT INTO candidate (cand_num, cand_lname, resume)
    VALUES (0, 'Haven', FILETOCLOB('haven.rsm', 'client'))
```

In the preceding example, the FILETOCLOB function reads the **haven.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.

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 the operating-system file to create 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.

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 must smart large objects would have an identifier with significantly fewer digits.

For example, suppose you specify a *pathname* value as follows:

```
'/tmp/resume'
```

If the CLOB column has an identifier of **203b2**, the LOTOFILE function would create the file:

```
/tmp/resume.203b2
```

To change this 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. 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, the LOTOFILE function ignores them.

- An exclamation point (!) at the end of the filename indicates that the filename does not need to be unique.

  For example, suppose you specify a pathname value as follows:

  ```
  '/tmp/resume.txt!'
  ```

  The LOTOFILE function does not use the smart-large-object identifier in the filename so it generates the following file:

  ```
  /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. ♦

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 name* and *column name* parameters are optional:

- If you omit *table name* and *column name*, 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.

  It obtains the system-specific storage defaults from either the ONCONFIG file or the sbspace. For more information on system-specified storage defaults, see the *INFORMIX-Universal Server Administrator's Guide*.
- When you specify a *table name* and *column name*, the LOCOPY function uses the storage characteristics from the specified *column name* for the BLOB or CLOB value that it creates.

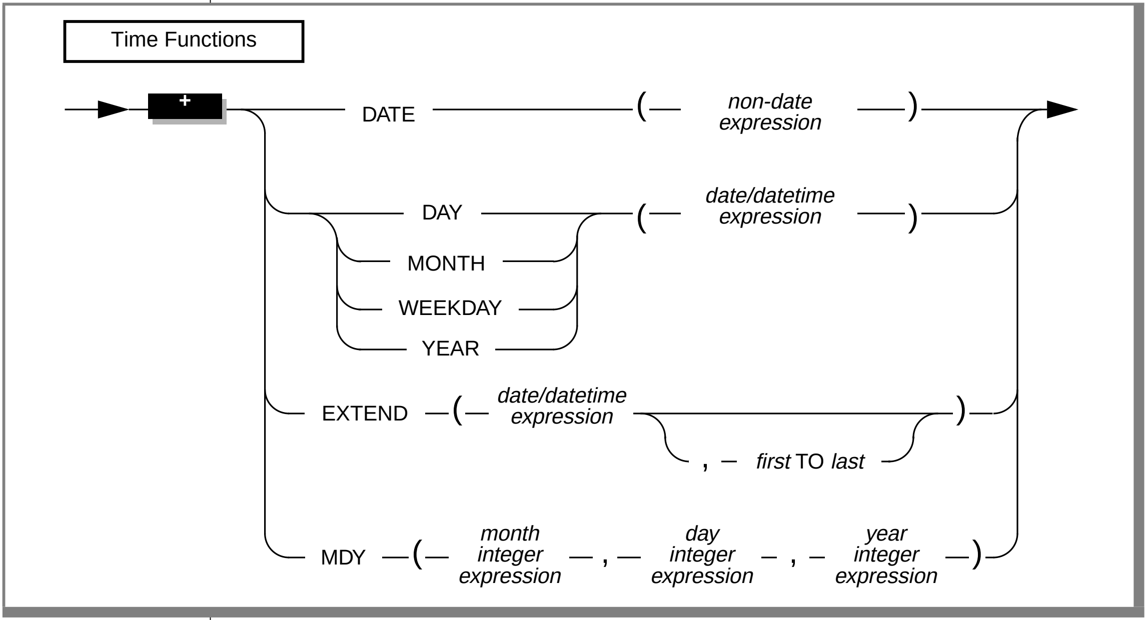
The LOCOPY function returns a handle value (a pointer) to the new BLOB or CLOB value. This function does *not* actually store the new smart-large-object value into a column in the database. You must assign the BLOB or CLOB value to the appropriate column.

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 interview 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 interview 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.

Time Functions



Element	Purpose	Restrictions	Syntax
date/datetime expression	An expression that serves as an argument in the following functions: DAY(), MONTH(), WEEKDAY(), YEAR(), and EXTEND()	The expression must evaluate to a DATE or DATETIME value.	Expression, p. 1-876
day integer expression	An expression that represents the number of the day of the month	The expression must evaluate to an integer not greater than the number of days in the specified month.	Expression, p. 1-876
first	A qualifier that specifies the first field in the result. If you do not specify *first* and *last* qualifiers, the default value of *first* is YEAR.	The qualifier can be any DATETIME qualifier, as long as it is larger than *last*.	DATETIME Field Qualifier, p. 1-874

(1 of 2)

Element	Purpose	Restrictions	Syntax
last	A qualifier that specifies the last field in the result. If you do not specify *first* and *last* qualifiers, the default value of *last* is FRACTION(3).	The qualifier can be any DATETIME qualifier, as long as it is smaller than *first*.	DATETIME Field Qualifier, p. 1-874
month integer expression	An expression that represents the number of the month	The expression must evaluate to an integer between 1 and 12, inclusive.	Expression, p. 1-876
non-date expression	An expression whose value is to be converted to a DATE data type	You can specify any expression that can be converted to a DATE data type. Usually you specify an expression that evaluates to a CHAR, DATETIME, or INTEGER value.	Expression, p. 1-876
year integer expression	An expression that represents the year	The expression must evaluate to a four-digit integer. You cannot use a two-digit abbreviation.	Expression, p. 1-876

(2 of 2)

DATE() Function

The DATE() function returns a DATE type value that corresponds to the non-date expression with which you call it. 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 non-date expression:

```
WHERE order_date < DATE('12/31/93')
```

When the DATE() function interprets a CHAR non-date 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('02/01/1995') FROM ship_info
```

This SELECT statement generates an error because the DATE function cannot convert this non-date 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 (1995), 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 non-date expression would need to be '95/02/01'. For information on the format of **DBDATE**, see Chapter 3 of the *Informix Guide to SQL: Reference*.

When you specify a positive INTEGER value for the non-date expression, the DATE function interprets the value as the number of days after the default date of 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 non-date 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 that corresponds 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 represents Sunday, one represents Monday, and so on. The following 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 / 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 representing the month, day, and year. The first expression must evaluate to an integer representing the number of the month (1 to 12).

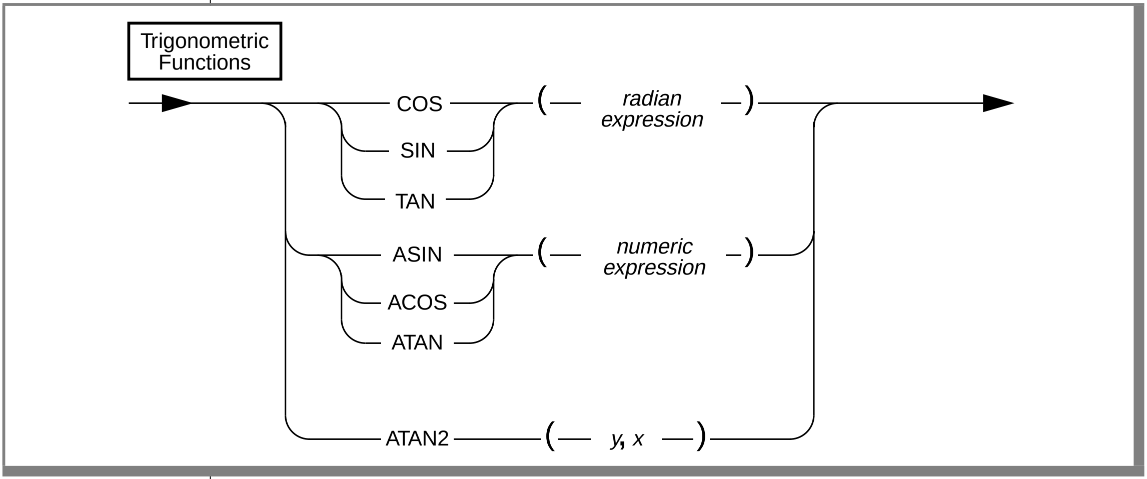
The second expression must evaluate to an integer that represents the number of the day of the month (1 to 28, 29, 30, or 31, as appropriate for the month.)

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'
```

Trigonometric Functions

A trigonometric function takes an argument, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
numeric expression	A numeric expression that serves as an argument to the ASIN(), ACOS(), or ATAN() functions	The expression must evaluate to a value between -1 and 1, inclusive.	Expression, p. 1-876
radian expression	An expression that evaluates to the number of radians. See "Formulas for Radian Expressions" on page 1-933 for further information on *radian expression*.	The expression must evaluate to a numeric value.	Expression, p. 1-876
x	An expression that represents the *x* coordinate of the rectangular coordinate pair (*x*, *y*)	The expression must evaluate to a numeric value.	Expression, p. 1-876
y	An expression that represents the *y* coordinate of the rectangular coordinate pair (*x*, *y*)	The expression must evaluate to a numeric value.	Expression, p. 1-876

Formulas for Radian Expressions

The COS(), SIN(), and TAN() functions take the number of radians (*radian expression*) as an argument.

If you are using degrees and want to convert degrees to radians, use the following formula:

```
# degrees * p/180= # radians
```

If you are using radians and want 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*180/3.1417) FROM anglestbl
```

SIN() Function

The SIN() function returns the sine of a radian expression. The following 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. The following 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*, θ) associated with (*x*, *y*). The following example compares *angles* to θ for the rectangular coordinates (4, 5):

```
WHERE angles > ATAN2(4,5)     --determines θ for (4,5) and
                                compares to angles
```

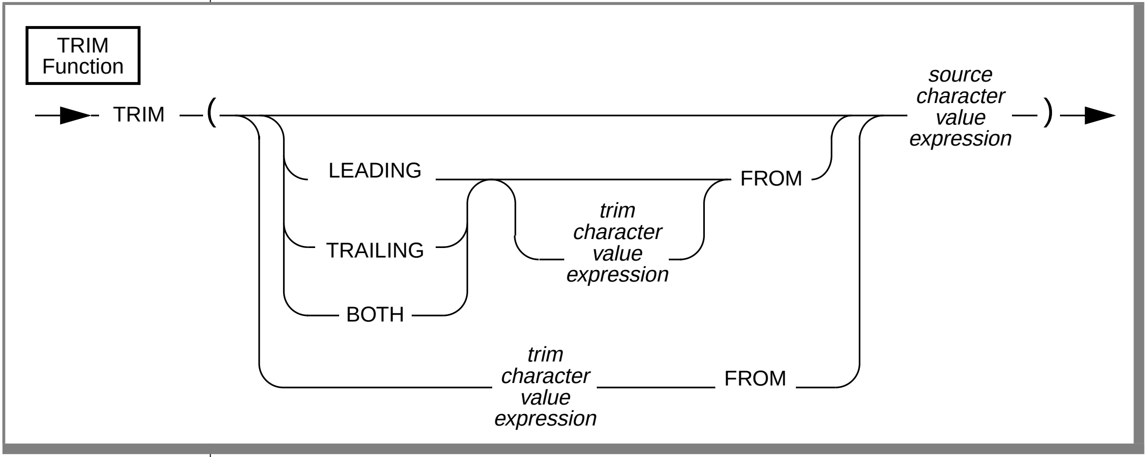
You can determine the length of the radial coordinate *r* using the expression shown in the following example:

```
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 shown in the following example:

```
SQRT(POW(4,2) + POW(5,2))     --determines r for (4,5)
```

TRIM() Function


TRIM Function
TRIM
(
LEADING
TRAILING
BOTH
trim character value expression
FROM
trim character value expression
FROM
source character value expression
)

Element	Purpose	Restrictions	Syntax
trim character value expression	An expression that evaluates to a single character or null	This expression must be a character expression.	Quoted String, p. 1-1010
source character value expression	An arbitrary character string expression, including a column or another TRIM() function	This expression cannot be a host variable.	Quoted String, p. 1-1010

Use the TRIM() function to remove leading or trailing (or both) pad characters from a string. The TRIM() function returns a VARCHAR string that is identical to the character string passed to it, except that any leading or trailing pad characters, if specified, are removed. If no trim specification (LEADING, TRAILING, or BOTH) is specified, then BOTH is assumed. If no *trim character value expression* is used, a single space is assumed. If either the *trim character value expression* or the *source character value expression* evaluates to null, the result of the trim function is null. The maximum length of the resultant string must be 255 or less, because the VARCHAR data type supports only 255 characters.

Some generic uses for the TRIM() function are shown in the following example:

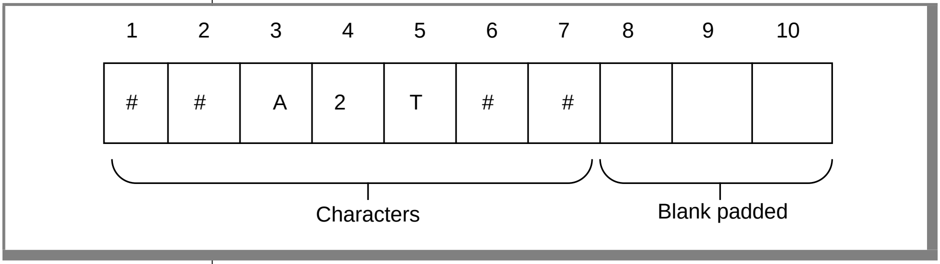
```
SELECT TRIM (c1) FROM tab;
SELECT TRIM (TRAILING '#' FROM c1) FROM tab;
SELECT TRIM (LEADING FROM c1) FROM tab;
UPDATE c1='xyz' FROM tab WHERE LENGTH(TRIM(c1))=5;
SELECT c1, TRIM(LEADING '#' FROM TRIM(TRAILING '%' FROM
    '###abc%%%')) FROM tab;
```

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 you are using and the data type of the *source character value expression*. See the *Guide to GLS Functionality* for further information on the GLS aspects of the TRIM() function in ESQL/C. ♦

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 1-5 shows this concept for the column entry '##A2T##', where the column is defined as CHAR(10).



1	2	3	4	5	6	7	8	9	10
#	#	A	2	T	#	#			

Figure 1-5
Column Entry in a Fixed-Length Character Column

If you want to trim the *trim character value 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 '#' trimmed are those that precede the other characters. The SELECT statement is shown, followed by Figure 1-6 on page 1-937, which presents the result.

```
SELECT TRIM(BOTH '#' FROM col1) FROM taba
```

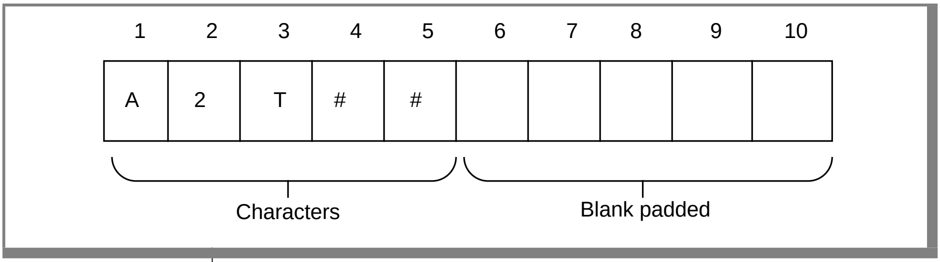
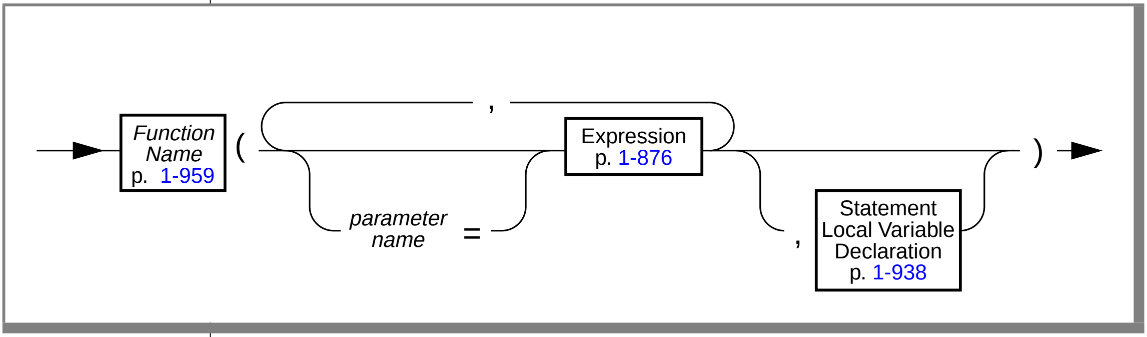


Figure 1-6
Result of TRIM() Operation

The following SELECT statement removes all occurrences of '#':

```
SELECT TRIM(LEADING '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba
```

User-Defined Functions



Element	Purpose	Restrictions	Syntax
parameter name	The name of a parameter for which you supply an argument to the function.	If you use the *parameter name* = option for any argument in the called function, you must use it for all arguments.	Identifier, p. 1-962

A user-defined function is a function that you write in SPL or in a language external to the database, such as the C language. User-defined functions contrast with functions that are built in to the database server. Unlike built-in functions, user-defined functions can only be used by the creator of the function, the DBA, and users who have been granted the Execute privilege on the function.

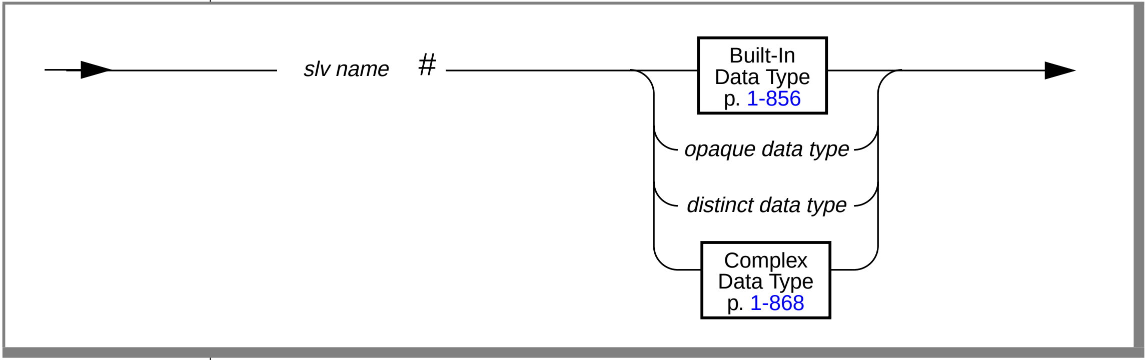
The database server identifies a function by the function name, the number of arguments the function accepts, the data type of each argument, and the order in which the arguments are listed. Because Universal Server allows function overloading, you can define more than one function of the same name, provided that the parameters differ in data type or order.

The following examples show some user-defined function expressions. The first example omits the *parameter name* option, and the second example uses the *parameter name* option:

```
read_address('Miller')
read_address(lastname = 'Miller')
```

If the function has an OUT parameter defined with the CREATE FUNCTION statement, you can declare a Statement Local Variable in the function expression, as described in the following sections.

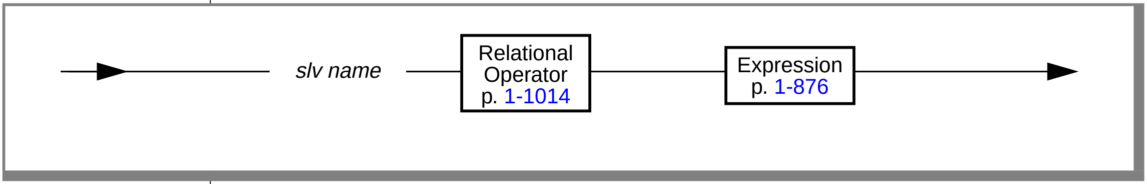
Statement Local Variable Declaration



Element	Purpose	Restrictions	Syntax
slv name	The 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. 1-962
opaque data type	The name of an opaque data type.	The opaque data type must already exist in the database.	Identifier, p. 1-962
distinct data type	The name of a distinct data type.	The distinct data type must already exist in the database.	Identifier, p. 1-962

The Statement Local Variable Declaration declares a Statement Local Variable in a function expression in an SQL statement. You can then use the value the function returns through the Statement Local Variable elsewhere in the statement, as described in the section "Using Statement Local Variables."

Statement Local Variable Expression



Element	Purpose	Restrictions	Syntax
slv name	The name of a statement local variable that has been defined.	The *slv name* exists only for the life of the statement. The *slv name* must be unique within the statement.	Identifier, p. 1-962

Using Statement Local Variables

A Statement Local Variable transmits a value from a function call in a statement to another part of the SQL statement. You can use a Statement Local Variable in any WHERE clause.

To use a Statement Local Variable with a call to a user-defined function, you must take three steps:

- Declare an OUT parameter when you register the function with CREATE FUNCTION.
- Declare the Statement Local Variable in a function expression in the WHERE clause, using the syntax of the Statement Local Variable Declaration.
- Use the Statement Local Variable in the WHERE clause, with the syntax of the Statement Local Variable Expression.

For example, if you register a function with the following CREATE FUNCTION statement, you can use its **y** parameter as a Statement Local Variable in a WHERE clause:

```
CREATE FUNCTION find_location(a FLOAT, b FLOAT, OUT y INT)
RETURNING VARCHAR(20
EXTERNAL NAME "/usr/lib/local/find.so"
LANGUAGE C
END FUNCTION;
```

In this example, **find_location()** accepts two FLOAT values that represent a latitude and a longitude and returns the name of the nearest city, along with an extra value of type INT 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 a Statement Local Variable 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 may be a heavily populated area) whose population rank is between 1 and 100. The statement will then return the zip code that corresponds to that city.

In the example, `rank # INT` is the Statement Local Variable Declaration and `rank < 101` is the Statement Local Variable Expression.

The data type you use when you declare the Statement Local Variable in a statement must be the same as the data type of the 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.

Each function can have only one OUT parameter and one Statement Local Variable. However, you can use more than one Statement Local Variable in a WHERE clause, if they are produced by different functions.

Important: *A Statement Local Variable is valid only for the life of a single SQL statement.*

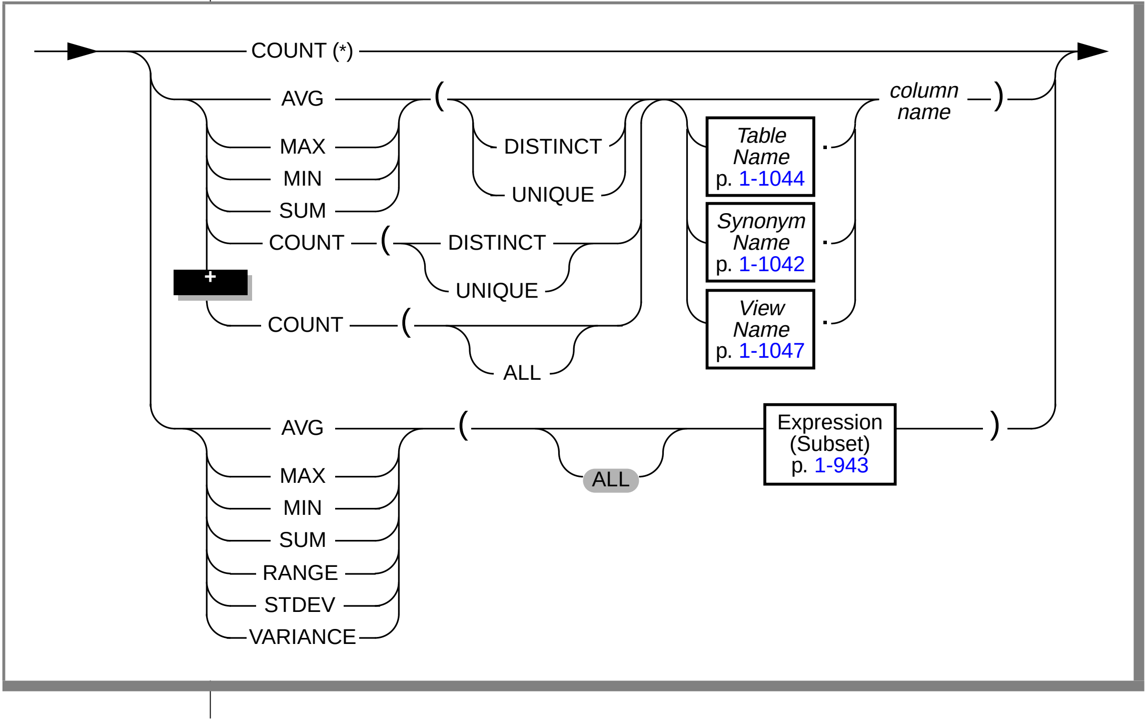
For more information on OUT parameters and Statement Local Variables, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data.

You cannot use an aggregate expression in a condition that is part of a WHERE clause unless the aggregate expression is used within a subquery.

The following diagram shows the syntax of aggregate function expressions.



Element	Purpose	Restrictions	Syntax
column name	The name of the column to which the specified aggregate function is applied	If you specify an aggregate expression and one or more columns in the SELECT clause of a SELECT statement, you must put all the column names that are not used within the aggregate expression or a time expression in the GROUP BY clause. You cannot apply an aggregate function to a BYTE or TEXT column. See "Subset of Expressions Allowed in an Aggregate Expression" on page 1-943 for other general restrictions. For restrictions that depend on the keywords that precede *column name*, see the headings for individual keywords on the following pages.	Identifier, p. 1-962

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, you must put all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause.

Subset of Expressions Allowed in an Aggregate Expression

The argument of an aggregate function cannot itself contain an aggregate function. You cannot use the aggregate functions found in the following list:

- MAX(AVG(order_num))
- An aggregate function in a WHERE clause unless it is contained in a subquery or if the aggregate is on a correlated column originating from a parent query and the WHERE clause is within a subquery that is within a HAVING clause
- An aggregate function on a BYTE or TEXT column

You cannot use a collection column as an argument to the following aggregate functions:

- AVG
- SUM
- MIN
- MAX

For the full syntax of expressions, see page 1-876.

Including or Excluding Duplicates in the Row Set

The DISTINCT keyword causes the function to be applied to only unique values from the named column. The UNIQUE keyword is a synonym for the DISTINCT keyword.

The ALL keyword is the opposite of the DISTINCT keyword. If you specify the ALL keyword, all the values that are selected from the named column or expression, including any duplicate values, are used in the calculation.

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 Option	Description
COUNT (*)	This option returns the number of rows that satisfy the query. If you do not specify a WHERE clause, this option returns the total number of rows in the table.
COUNT DISTINCT or COUNT UNIQUE	This option returns the number of unique non-null values in the specified column.
COUNT (*column name*) or COUNT (ALL *column name*)	This option 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. The following examples pose queries against the **orders** table in the demonstration database. Most of the examples query against the **ship_instruct** column in this table. 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 *Informix Guide to SQL: Reference*.

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 (*) keyword 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(*) keyword 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(*) keyword includes the null columns in the count unless the WHERE clause explicitly omits them.

In the following example, the user wants to know the total number of rows in the **orders** table. So the user uses 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. So the user uses 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 specifies the COUNT (*) function in the select list and 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

COUNT DISTINCT and UNIQUE Keywords

The COUNT DISTINCT keywords return the number of unique values in the column or expression, as the following example shows. If the COUNT 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 keyword returns a zero for that column.

The UNIQUE keyword has exactly the same meaning as the DISTINCT keyword when the UNIQUE keyword is used within the COUNT function. The UNIQUE keyword returns the number of unique non-null values in the column or expression.

The following example uses the UNIQUE keyword, but it is equivalent to the preceding example that uses the DISTINCT keyword:

```
SELECT COUNT (UNIQUE item_num) FROM items
```

In the following example, the user wants to know how many unique non-null values are in the **ship_instruct** column of the **orders** table. So the user enters 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

COUNT column name Option

The COUNT *column name* option returns the total number of non-null values in the column or expression, as the following example shows:

```
SELECT COUNT (item_num) FROM items
```

You can include the ALL keyword before 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 name* option:

```
SELECT COUNT (ALL item_num) FROM items
```

In the following example the user wants to know how many non-null values are in the **ship_instruct** column of the **orders** table. So the user enters the COUNT *column name* function in the select list of the SELECT statement.

```
SELECT COUNT(ship_instruct) AS total_notnulls
    FROM orders
```

The following table shows the result of this query.

total_notnulls
21

The user can also find out how many non-null values are in the **ship_instruct** column by including 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.

all_notnulls
21

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 (mean) is greater than 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 specified column is null. If every column value is null, the AVG function returns a null for that column.

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 specified 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 specified 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 shown in the following example. 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 specified 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 character column.

RANGE() Function

The RANGE() function computes the range for a sample of a population. It computes 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 shown in the following example:

```
SELECT RANGE(age) FROM u_pop
    GROUP BY birth
```

Nulls are ignored unless every value in the specified 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 for a sample of a population. It is the square root of the VARIANCE() function.

You can apply the STDEV() function only to numeric columns. The following query finds the standard deviation on a population:

```
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 shown in the following example:

```
SELECT STDEV(age) FROM u_pop
    GROUP BY birth
    WHERE STDEV(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the STDEV() function 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.*

VARIANCE() Function

The VARIANCE() keyword returns the variance for a sample of values as an unbiased estimate of the variance of the population. It computes the following value:

```
(SUM(Xi**2) - (SUM(Xi)**2)/N)/(N-1)
```

In this example, *Xi* is each value in the column and *N* is the total number of values in the column. You can apply the VARIANCE() function only to numeric columns. The following query finds the variance on 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 shown in the following example:

```
SELECT VARIANCE(age) FROM u_pop
    GROUP BY birth
    WHERE VARIANCE(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the VARIANCE() function returns a null for that column.

Important: *All computations 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.*

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 **a_number**. The contents of this table are as follows.

a_number
2
2
2

(1 of 2)

a_number
3
3
4
(null)

(2 of 2)

You can use aggregate functions to obtain different types of information about the **a_number** column and the **testtable** table. In the following example, the user specifies the AVG function to obtain the average of all the non-null values in the **a_number** column:

```
SELECT AVG(a_number) 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 one shown in the preceding example. If you enter a series of SELECT statements that have different aggregate functions in the select list and do not have a WHERE clause, you receive the results that the following table shows.

Function	Results
COUNT(*)	7
AVG	2.66666666666667
AVG (DISTINCT)	3.00000000000000
MAX	4
MAX(DISTINCT)	4

(1 of 2)

Function	Results
MIN	2
MIN(DISTINCT)	2
SUM	16
SUM(DISTINCT)	9
COUNT(DISTINCT)	3
COUNT(ALL)	6
RANGE	2
STDEV	0.81649658092773
VARIANCE	0.66666666666667

(2 of 2)

Error Checking with Aggregate Functions

ESQL

Aggregate functions always return one row; if no rows are selected, the function returns a null. You can use the COUNT (*) keyword 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 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. See the GET DIAGNOSTICS statement in this manual. ♦

Using Arithmetic Operators with Expressions

You can combine expressions with arithmetic operators to make complex expressions. To combine expressions, connect them with the following binary arithmetic operators.

Arithmetic Operation	Arithmetic Operator	Operator Function
Addition	+	**plus()**
Subtraction	-	**minus()**
Multiplication	*	**times()**
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.

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 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 binary arithmetic operator functions to handle your own user-defined data types. For more information, see the *Extending INFORMIX-Universal Server: Data Types* manual.

Informix also provides the following unary arithmetic operators:

Arithmetic Operation	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 arithmetic operator functions to handle your own user-defined data types. For more information on how to write versions of operator functions, see the *Extending INFORMIX-Universal Server: Data Types* manual.

If any value that participates in an arithmetic expression is null, the value of the entire expression is null, as shown in the following example:

```
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.

References

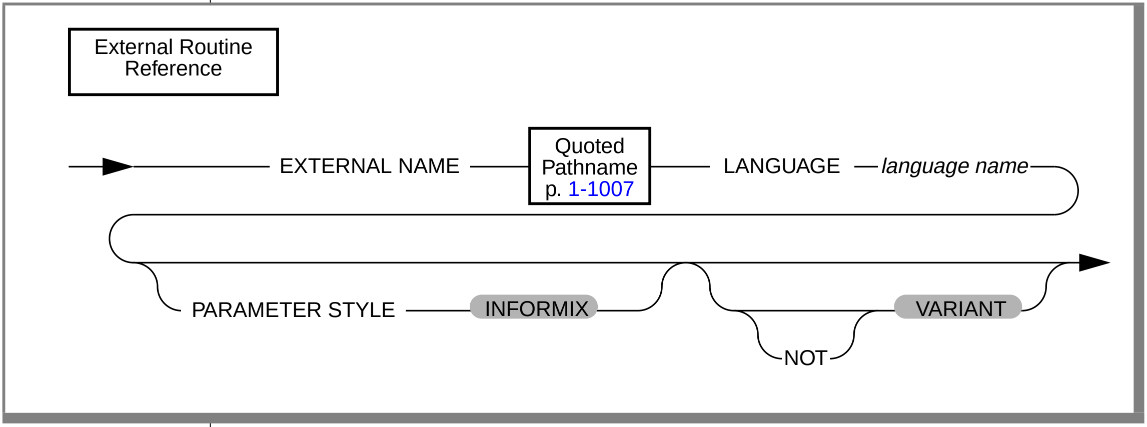
In the *Informix Guide to SQL: Tutorial*, see Chapter 2 for a discussion of expressions in the SELECT statement.

In the *Guide to GLS Functionality*, see the discussions of column expressions, the discussion of length functions, and the discussion of the TRIM() function.

External Routine Reference

Use an External Routine Reference when you write an external routine.

Syntax



Element	Purpose	Restrictions	Syntax
language name	The name of the language used to write the external routine	The *language name* must be the name of a supported external language. (The name of the language must be C.)	Identifier, p.1-962

Usage

The External Routine Reference applies only to external functions and provides the following information about an external routine:

- The pathname to the executable object code, stored in a shared library
- The name of the language in which the routine is written
- The parameter style of the routine
- The VARIANT or NOT VARIANT option, if you specify one

Parameter Style

By default, the parameter style is INFORMIX. If you specify an OUT parameter, the OUT argument is passed by reference.

VARIANT or NOT VARIANT

The VARIANT and NOT VARIANT options apply only to functions. You cannot use VARIANT or NOT VARIANT with procedures.

A variant function does not always return the same value for the same arguments. For example, a function that returns the current date and time or a set of rows from a table is a variant function. In INFORMIX-Universal Server,external functions are variant by default.

A non-variant function always returns the same value when passed the same arguments. To register a non-variant function, add the NOT VARIANT option in the External Routine Reference or in the Routine Modifier clause that is discussed on page 1-1022. If you specify the option in both places, you must use the same option in each.

Example

The following example registers an external function named **equal()** that takes two values of **point** data type as arguments. In this example, **point** is an opaque 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 library 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(point, point)** is executing.

References

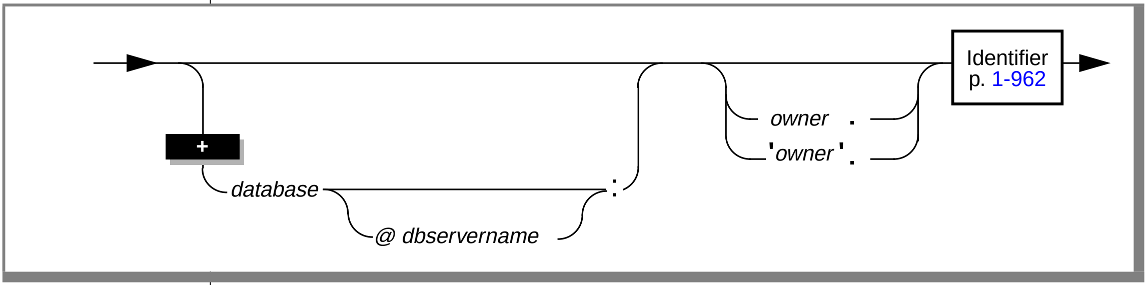
In this manual, see the CREATE FUNCTION and CREATE PROCEDURE statements.

For information about how to create and register external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

Function Name

The Function Name segment specifies the name of a function.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the function resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the server that is home to *database*. The **@** symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the function	If you are using an ANSI-compliant database, you must specify an owner if you do not own the function. If you do not specify an owner, the default owner is the current user.	Must conform to the conventions of your operating system.

Usage

In a statement that calls for a Function Name, you can enter an identifier with an optional owner name, database name, and server name. The database and server names allow you to use a function stored on a remote database. A function name with a database name, server name, and owner name is called a fully qualified function name.

The actual name of the function is an SQL identifier.

ANSI

The owner name is case sensitive. In an ANSI database, if you type quotation marks around the name, it is stored as you type it. If you do not use quotation marks, the name is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of functions. For more information, see the *Guide to GLS Functionality*. ♦

Routine Overloading

Due to routine overloading, a function name does not need to be unique in INFORMIX-Universal Server. You can define more than one function with the same name and different parameter lists.

Functions are uniquely identified by their signature. A function's signature includes the following items:

- The routine type (FUNCTION or PROCEDURE)
- The routine name
- The number, data type, and order of the function's parameters

If a function name is not unique, Universal Server uses routine resolution to identify the instance of the function to execute. For information about routine resolution, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

Qualified Function Name

When you add the database name and server name options, you use a fully qualified function name to specify a remote function. You can use those options under the following conditions:

- All arguments passed to the function have built-in data types.
- The instance of the function that is invoked has built-in data types for all of its parameters.
- Any values the function returns have built-in data types.

References

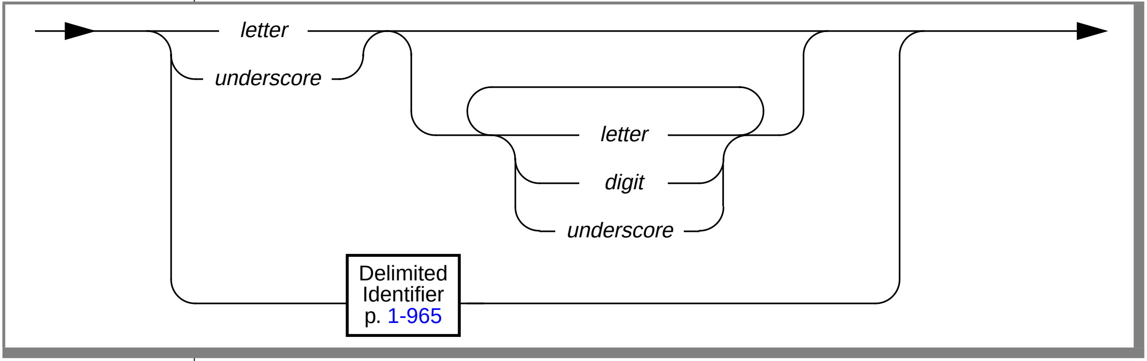
In this manual, see the CALL, CREATE FUNCTION, DROP FUNCTION, DROP ROUTINE, and EXECUTE FUNCTION statements.

For information about how to create and use external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual. In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to create and use SPL routines.

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	An integer that forms part of the identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 1-997
letter	A letter that forms part of the identifier	If you are using the default locale, a *letter* must be an uppercase or lowercase character in the range 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. See "Support for Non-ASCII Characters in Identifiers" on page 1-965 for further information.	Letters are literal values that you enter from the keyboard.
underscore	An underscore character that forms part of the identifier	You cannot substitute a space character, dash, hyphen, or any other nonalphanumeric character for the underscore character.	The underscore character (_) is a literal value that you enter from the keyboard.

Usage

An identifier can contain up to 18 bytes, inclusive.

Use of Reserved Words as Identifiers

Although you can use almost any word as an identifier, syntactic ambiguities can result from using reserved words as identifiers in SQL statements. The statement might fail or might not produce the expected results. See "Potential Ambiguities and Syntax Errors" on page 1-968 for a discussion of the syntactic ambiguities that can result from using reserved words as identifiers and an explanation of workarounds for these problems.

Delimited identifiers provide the easiest and safest way to use a reserved word as an identifier without causing syntactic ambiguities. No workarounds are necessary when you use a reserved word as a delimited identifier. See "Delimited Identifiers" on page 1-965 for the syntax and usage of delimited identifiers.

Tip: *If you receive an error message that seems unrelated to the statement that caused the error, check to determine whether the statement uses a reserved word as an undelimited identifier.*

ANSI-Reserved Words

The following list specifies all the ANSI-reserved words (that is, reserved words in the ANSI SQL standard).

ADA
all
and
any
as
asc
authorization
avg
begin
between
by
execute
exists
fetch
float
for
fortran
found
from
go
goto
group
order
pascal
pli
precision
primary
procedure
privileges
public
real
rollback
schema

(1 of 2)

char
character
check
close
cobol
commit
continue
count
create
current
cursor
dec
decimal
declare
delete
desc
distinct
double
end
escape
exec
having
in
indicator
insert
int
integer
into
is
language
like
max
min
module
not
null
numeric
of
on
open
option
or
section
select
set
smallint
some
sql
sqlcode
sqlerror
sum
table
to
union
unique
update
user
values
view
whenever
where
with
work

(2 of 2)

You can flag identifiers as ANSI-reserved words by taking the following steps:

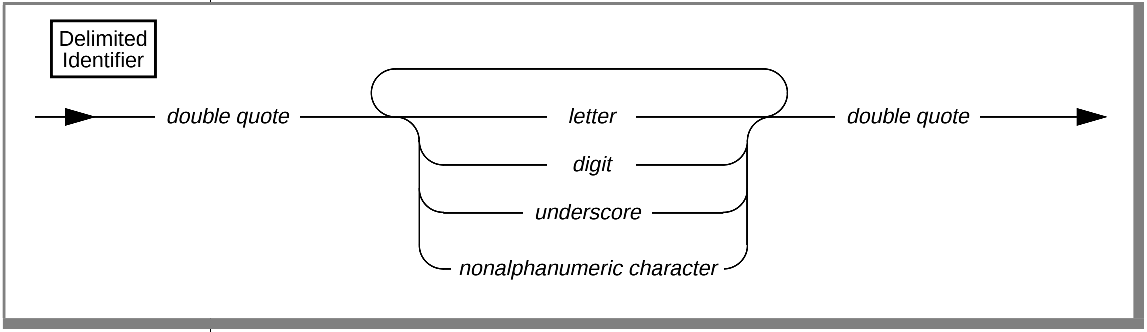
- Set the **DBANSIWARN** environment variable or use the **-ansi** flag at compile time to receive compile-time warnings.
- Set the **DBANSIWARN** environment variable at runtime to receive warning flags set in the SQLWARN array of **sqlca**.

Support for Non-ASCII Characters in Identifiers

GLS

If you are using a nondefault locale, you can use any alphabetic character that your locale recognizes as a *letter* in an SQL identifier name. You can use a non-ASCII character as a letter as long as your locale supports it. This feature enables you to use non-ASCII characters in the names of database objects such as indexes, tables, and views. For a list of SQL identifiers that support non-ASCII characters, see the *Guide to GLS Functionality*. ♦

Delimited Identifiers



Element	Purpose	Restrictions	Syntax
digit	An integer that forms part of the delimited identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 1-997
double quote	The double-quote character that marks a string as a delimited identifier	If the **DELIMIDENT** environment variable is not set, values within double quotes are treated as quoted strings by the database server.	The double quote character (") is a literal value that you enter from the keyboard.
letter	A letter that forms part of the delimited identifier	Letters in delimited identifiers are case-sensitive. If you are using the default locale, a *letter* must be an uppercase or lowercase character in the range a-z (in the ASCII code set). If you are using a nondefault locale, *letter* must be an alphabetic character that the locale supports. See "Support for Non-ASCII Characters in Delimited Identifiers" on page 1-967 for further information.	Letters are literal values that you enter from the keyboard.
nonalpha-numeric character	A nonalphanumeric character, such as # or $ or space, that forms part of the delimited identifier	If you are using the ASCII code set, you can specify any ASCII nonalphanumeric character.	Nonalphanumeric characters are literal values that you enter from the keyboard.
underscore	An underscore (_) that forms part of the delimited identifier	You can use a dash, hyphen, or any other appropriate character in place of the underscore character.	The underscore (_) is a literal value that you enter from the keyboard.

Delimited identifiers allow you to specify names for database objects that are otherwise identical to SQL reserved keywords, such as TABLE, WHERE, DECLARE, and so on. The only database object for which you cannot use delimited identifiers is database name.

Delimited identifiers are case sensitive.

Delimited identifiers are compliant with the ANSI standard.

Support for Nonalphanumeric Characters

You can use delimited identifiers to specify nonalphanumeric characters in the names of database objects. However, you cannot use delimited identifiers to specify nonalpha characters in the names of storage objects such as dbspaces and blobspaces.

Support for Non-ASCII Characters in Delimited Identifiers

GLS

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. See the *Guide to GLS Functionality* for a list of identifiers that support non-ASCII characters and for information on non-ASCII characters in delimited identifiers. ♦

Effect of DELIMIDENT Environment Variable

To use delimited identifiers, you must set the **DELIMIDENT** environment variable. When you set the **DELIMIDENT** environment variable, database objects in double quotes (") are treated as identifiers and database objects in single quotes (') are treated as strings. If the **DELIMIDENT** environment variable is not set, values within double quotes are also treated as strings.

If the **DELIMIDENT** variable 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'
```

Examples of Delimited Identifiers

The following example shows how to create a table with a case-sensitive table name:

```
CREATE TABLE "Power_Ranger" (...)
```

The following example shows how to create a table whose name includes a space character. If the table name were not in double quotes ("), you could not use a space character or any other nonalpha character except an underscore (_) in the name.

```
CREATE TABLE "My Customers" (...)
```

The following example shows how to create a table that uses a keyword as the table name:

```
CREATE TABLE "TABLE" (...)
```

Using Double Quotes Within a Delimited Identifier

If you want to include a double-quote (") within a delimited identifier, you must precede the double-quote (") with another double-quote ("), as shown in the following example:

```
CREATE TABLE "My""Good""Data" (...)
```

Potential Ambiguities and Syntax Errors

Although you can use almost any word as an SQL identifier, syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. The following sections outline some potential pitfalls and workarounds.

Using Functions as Column Names

The following two examples show a workaround for using a 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 following 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 shown in the following example:

```
SELECT "avg" from mytab -- successful
```

The workaround in 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 shown in the following example:

```
CREATE TABLE mytab (user char(10),
    CURRENT DATETIME HOUR TO SECOND,TODAY DATE)

INSERT INTO mytab VALUES('josh','11:30:30','1/22/89')

SELECT user,current,today FROM mytab
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the SQL functions USER, CURRENT, and TODAY. Thus, instead of returning `josh`, `11:30:30`,`1/22/89`, 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, there might be more than one suitable workaround.

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 rest of the examples in this section show workarounds for using the keywords UNIQUE or DISTINCT as a column name in a CREATE TABLE statement.

Using **unique** as a column name causes the following example to fail because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER) -- fails
```

The workaround shown in the following example uses two SQL statements. The first statement creates the column **mycol**; the second 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 shown in the following example:

```
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;
```

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: *Informix recommends that you use primary keys as an access method rather than exploiting the rowid column.*

Using Keywords as Table Names

The examples in this section show workarounds that involve owner naming when you use the keyword STATISTICS or OUTER as a table name. This workaround also applies to the use of 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 workaround in the following 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 the following 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 show how the AS or FROM keyword can be used 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 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
```

The following 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

The 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 following two examples show how to use the keyword WITH as a table alias.

Using **with** as a table alias causes the following example to fail because the database server interprets the keyword 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 an entity 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 error 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

If you use any of the following keywords as identifiers for variables in a routine, you can create ambiguous syntax.

CURRENT
DATETIME
GLOBAL
INTERVAL
NULL
OFF
ON
PROCEDURE
SELECT

Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT

A routine cannot insert a variable using the CURRENT, DATETIME, INTERVAL, or NULL keyword as the name.

For example, if you define a variable called **null**, when you try to insert the value **null** into a column, you receive a syntax error, as shown in the following example:

```
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 shown in the preceding example:

```
 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 it in an IN condition.

Using ON, OFF, or PROCEDURE with TRACE

If you define a procedure variable called *on*, *off*, or *procedure,* and you attempt to use it in a TRACE statement, the value of the variable does not trace. 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 shown in the following example 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 shown in the following example:

```
DEFINE "global" INT; -- successful
```

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 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, seen 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
```

The SET Keyword in the ON EXCEPTION Statement

If you use a statement that begins with the keyword SET inside the statement ON EXCEPTION, you must enclose it in a BEGIN...END block. The following list shows some of the SQL statements that begin with the keyword SET.

SET
SET DEBUG FILE
SET EXPLAIN
SET ISOLATION
SET LOCK MODE
SET LOG
SET OPTIMIZATION
SET PDQPRIORITY

The following examples show 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 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 block:

```
ON EXCEPTION IN (-107)
    BEGIN
    SET LOCK MODE TO WAIT; -- ok
    END
END EXCEPTION
```

References

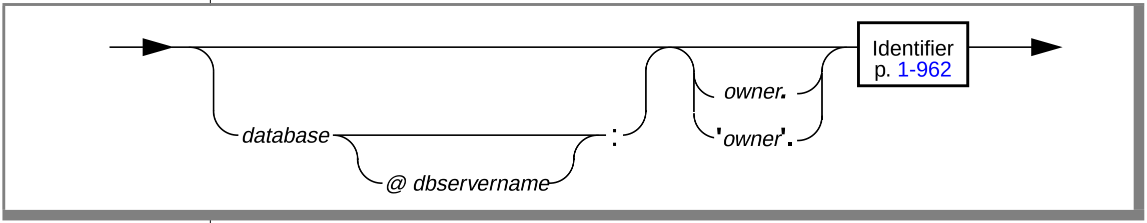
In the *INFORMIX-Universal Server Administrator’s Guide*, see the owner-naming discussion.

In the *Guide to GLS Functionality*, see the discussion of identifiers that support non-ASCII characters and the discussion of non-ASCII characters in delimited identifiers.

Index Name

The Index Name segment specifies the name of an index. Use the Index Name segment whenever you see a reference to an index name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the index resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the Universal Server database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the index	If you are using an ANSI-compliant database, you must specify the owner for an index that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the index is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of indexes. For more information, see the *Guide to GLS Functionality*. ♦

If you are creating an index, the *name* must be unique within a database.

ANSI

The *owner.name* combination is case sensitive. 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. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

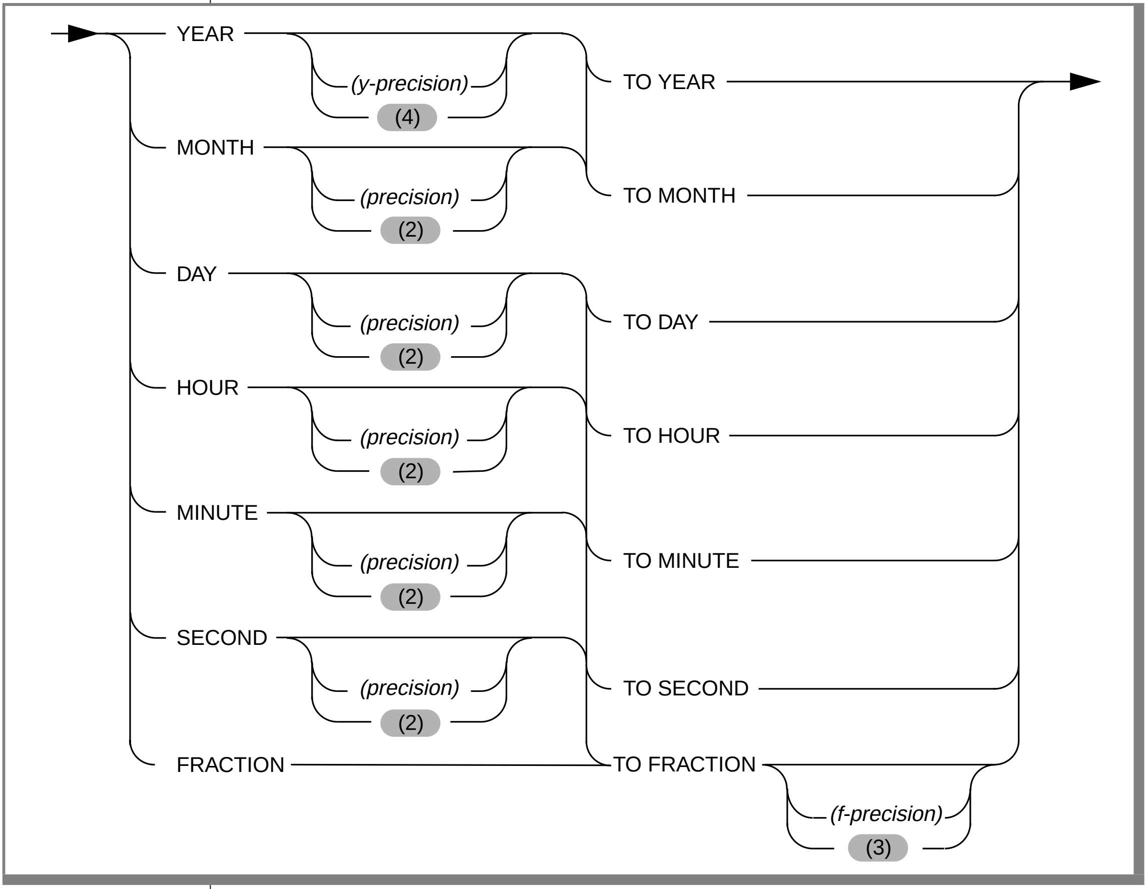
References

See the CREATE INDEX statement in this manual for information on defining indexes.

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
f-precision	The maximum number of digits used in the fraction field. The default value of *f-precision* is 3.	The maximum value that you can specify in *f-precision* is 5.	Literal Number, p. 1-997
precision	The number of digits in the largest number of months, days, hours, or minutes that the interval can hold. The default value of *precision* is 2.	The maximum value that you can specify in *precision* is 9.	Literal Number, p. 1-997
y-precision	The number of digits in the largest number of years that the interval can hold. The default value of *y-precision* is 4.	The maximum value that you can specify in *y-precision* is 9.	Literal Number, p. 1-997

Usage

The next two examples show INTERVAL data types of the YEAR TO MONTH type. 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 contain only one field, 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 qualifiers:

```
YEAR(5) TO MONTH
```

```
DAY (5) TO FRACTION(2)
```

```
DAY TO DAY
```

```
FRACTION TO FRACTION (4)
```

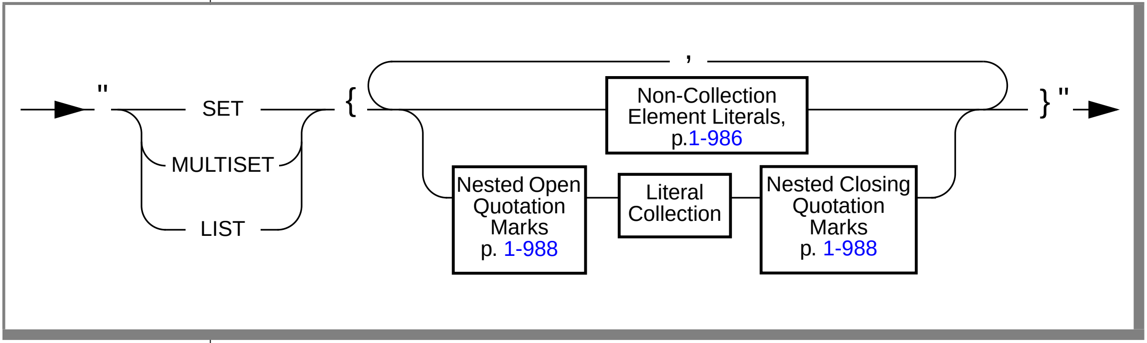
References

In the *Informix Guide to SQL: Reference*, see the INTERVAL data type in Chapter 2 for information about specifying INTERVAL field qualifiers and using INTERVAL data in arithmetic and relational operations.

Literal Collection

The Literal Collection segment specifies the syntax for values of the collection data types: SET, LIST, and MULTISET.

Syntax


"
SET
MULTISET
LIST
{
,
Non-Collection Element Literals, p.1-986
Nested Open Quotation Marks p. 1-988
Literal Collection
Nested Closing Quotation Marks p. 1-988
}
"

Usage

You can specify literal collection values for each of the collection data types: SET, MULTISET, or LIST. The entire literal collection value must be enclosed in quotes; each literal within the literal collection must also be enclosed in quotes, following the rule explained on page 1-988.

To specify a single literal-collection value, specify the collection type and the literal values. The following SQL statement inserts four integer values into the **set_col** column that is declared as `SET(INT NOT NULL)`:

```
INSERT INTO table1 (set_col) VALUES ("SET{6, 9, 9, 4}")
```

You specify an empty collection with a set of empty braces ({}). The following INSERT statement inserts an empty list into a collection column **list_col** that is declared as LIST(INT NOT NULL):

```
INSERT INTO table2 (list_col) VALUES ("LIST{}")
```

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:

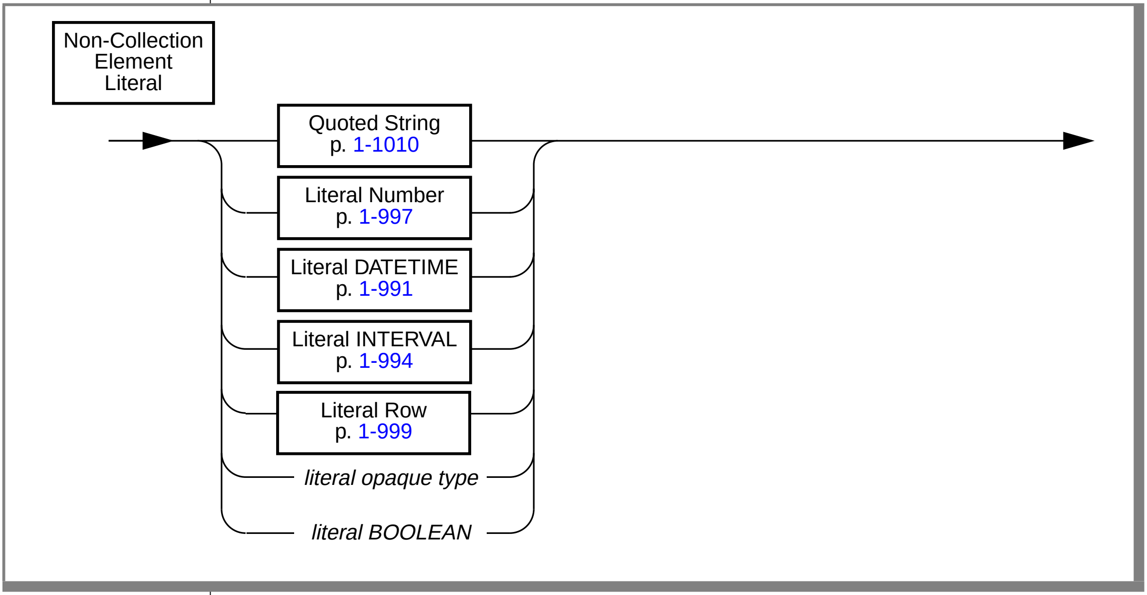
```
nest_col SET(MULTISET (INT NOT NULL) NOT NULL)
```

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 quotes in INSERT statements, see "Nested Quotation Marks" on page 1-988.

Non-Collection Element Literal



Element	**Purpose**	**Restrictions**	**Syntax**
literal opaque type	The 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 type.
literal BOOLEAN	The literal representation of a BOOLEAN value	A literal BOOLEAN value can only be 't' (TRUE) or 'f' (FALSE) and must be specified as a quoted string.	Quoted String, p. 1-1010

Elements of a collection can be literal values for the following built-in data types:

- The CHAR, LVARCHAR, and VARCHAR data types have literal values specified as quoted strings.

  GLS

  The NCHAR and NVARCHAR data types also use quoted strings for literal values. ♦

  For more information, see "Quoted String" on page 1-1010.
- The DECIMAL, FLOAT, INT8, INTEGER, MONEY, SMALLFLOAT and SMALLINT data types have literal values specified by the Literal Number syntax.

  For more information, see "Literal Number" on page 1-997.
- The DATE data type has a literal value specified as a quoted string.

  For more information, see "Quoted String" on page 1-1010.
- The DATETIME data type has literal values specified by the Literal DATETIME syntax.

  For more information, see "Literal DATETIME" on page 1-991.
- The INTERVAL data type has literal values specified by the Literal INTERVAL syntax.

  For more information, see "Literal INTERVAL" on page 1-994.
- The BOOLEAN data type has the literal values of `'t'` and `'f'`.

  For more information, see the description of the BOOLEAN data type in the *Informix Guide to SQL: Reference*.

Important: *You cannot specify the simple-large-object data types (BYTE and TEXT) as the element type for a collection.*

Collection elements can also be literal values for the following user-defined data types:

- An opaque data type has a literal value that corresponds to the string representation for the opaque data type. The input support function of the opaque type defines this string representation.
- A row type, named or unnamed, has a literal value specified as field values enclosed with parentheses.

  When the collection element type is a named row type, you do not have to cast the values that you insert to the named row type.

  For more information, see "Literal Row" on page 1-999
- Another collection type (SET, MULTISET, or LIST) has a literal value whose specification depends on the context its use.

  A collection whose element type is another collection is called a *nested collection*. For information on literal collection value as a column value or as a collection-variable value, see "Example of Nested Quotation Marks" on page 1-989.

Nested Quotation Marks

Whenever you nest collection literals, you use nested quotation marks. In these cases, you must follow the rule for nesting quotation marks. Otherwise, the server cannot correctly parse the strings.

The general rule is that you must double the number of quotes for each new level of nesting. For example, if you use double quotes for the first level, you must use two double quotes for the second level, four double quotes 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 quotes for the second level and four single quotes for the third level.

There is no limit to the number of levels you can nest, as long as you follow this rule.

Example of Nested Quotation Marks

The following example illustrates the case for two levels of nested collection literals, using double quotes. Table **tab5** is a one-column table whose 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 any individual literal value, the opening quotation marks and the closing quotation marks must match. In other words, if you open a literal with two double quotes, you must close that literal with two double quotes (""a literal value"").

The rules for nested quotation marks apply to all literals—collection literals and non-collection literals—that are nested in a single collection value.

E/C

To specify nested quotes within an SQL statement in an ESQL/C program, you 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 *INFORMIX-ESQL/C Programmer's Manual*. ♦

References

See the INSERT, UPDATE, and SELECT statements in this manual. See also the Row Literal segment.

In the *Informix Guide to SQL: Tutorial*, see Chapter 10 and Chapter 12 for information about how to create and use collection data types.

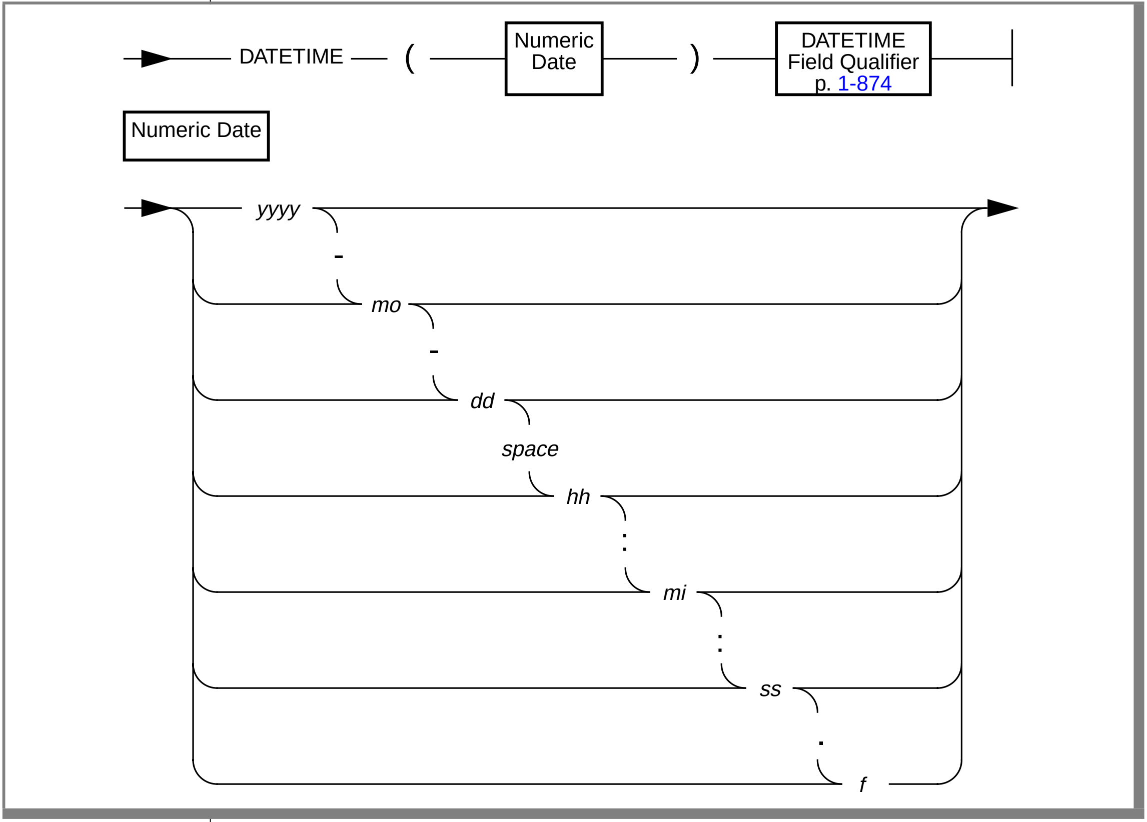
In the *Informix Guide to SQL: Reference*, see the SET, MULTISET, and LIST data types in Chapter 2.

In the *Guide to GLS Functionality*, see the discussion of customizing NCHAR and NVARCHAR data types.

Literal DATETIME

The Literal DATETIME segment specifies a literal DATETIME value. Use the Literal DATETIME segment whenever you see a reference to a literal DATETIME in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	The day expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-997
f	The decimal fraction of a second expressed in digits	You can specify up to 5 digits.	Literal Number, p. 1-997
hh	The hour expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-997
mi	The minute expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-997
mo	The month expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-997
space	A space character	You cannot specify more than 1 space character.	The space character is a literal value that you enter by pressing the space bar on the keyboard.
ss	The second expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-997
yyyy	The year expressed in digits	You can specify up to 4 digits. If you specify 2 digits, the database server uses the setting of the **DBCENTURY** environment variable to extend the year value. If the **DBCENTURY** environment variable is not set, the database server uses the current century to extend the year value.	Literal Number, p. 1-997

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.

The following examples show literal DATETIME values:

```
DATETIME (93-3-6) YEAR TO DAY

DATETIME (09:55:30.825) HOUR TO FRACTION

DATETIME (93-5) YEAR TO MONTH
```

The following example shows a literal DATETIME value used with the EXTEND function:

```
EXTEND (DATETIME (1993-8-1) YEAR TO DAY, YEAR TO MINUTE)
    - INTERVAL (720) MINUTE (3) TO MINUTE
```

References

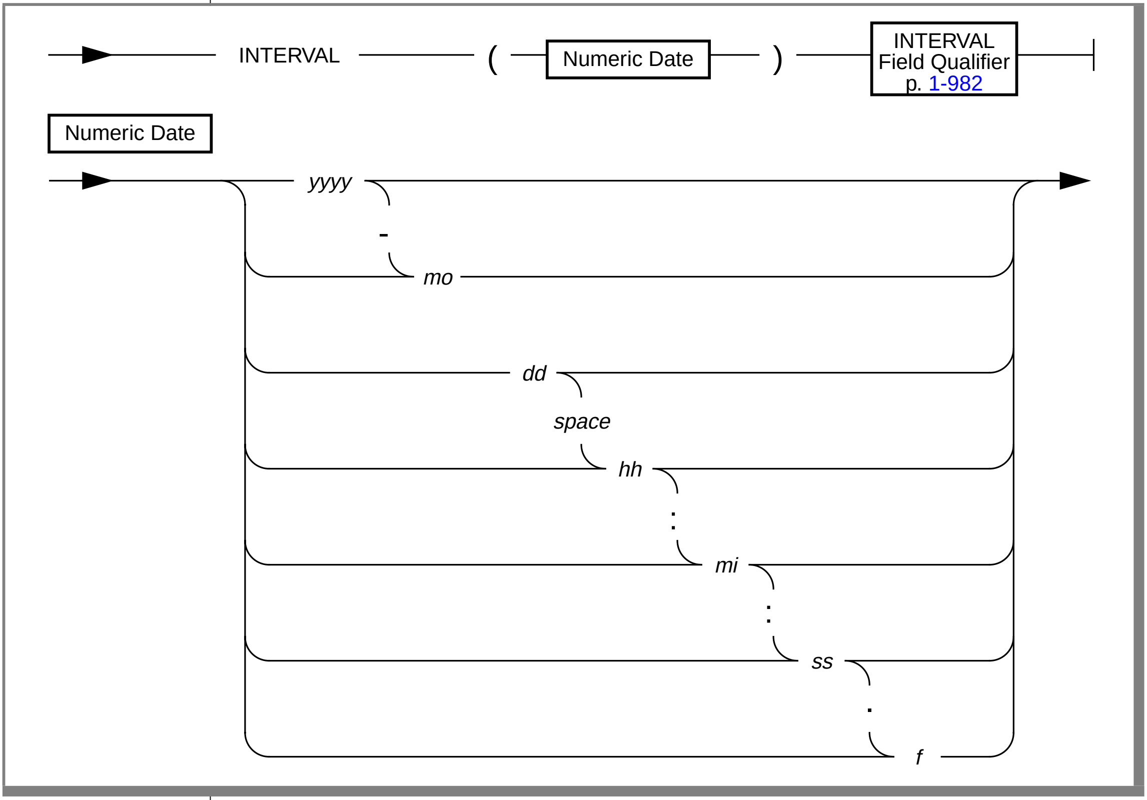
In the *Informix Guide to SQL: Reference*, see the DATETIME data type in Chapter 2 and the **DBCENTURY** environment variable in Chapter 3.

In the *Guide to GLS Functionality*, see the discussion of customizing DATETIME values for a locale.

Literal INTERVAL

The Literal INTERVAL segment specifies a literal INTERVAL value. Use the Literal INTERVAL segment whenever you see a reference to a literal INTERVAL in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	The number of days	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997
f	The decimal fraction of a second	You can specify up to 5 digits, depending on the precision given to the fractional portion in the INTERVAL field qualifier.	Literal Number, p. 1-997
hh	The number of hours	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997
mi	The number of minutes	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997
mo	The number of months	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997
space	A space character	You cannot use any other character in place of the space character.	The space character is a literal value that you enter by pressing the space bar on the keyboard.
ss	The number of seconds	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997
yyyy	The number of years	The maximum number of digits allowed is 4, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-997

Usage

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
```

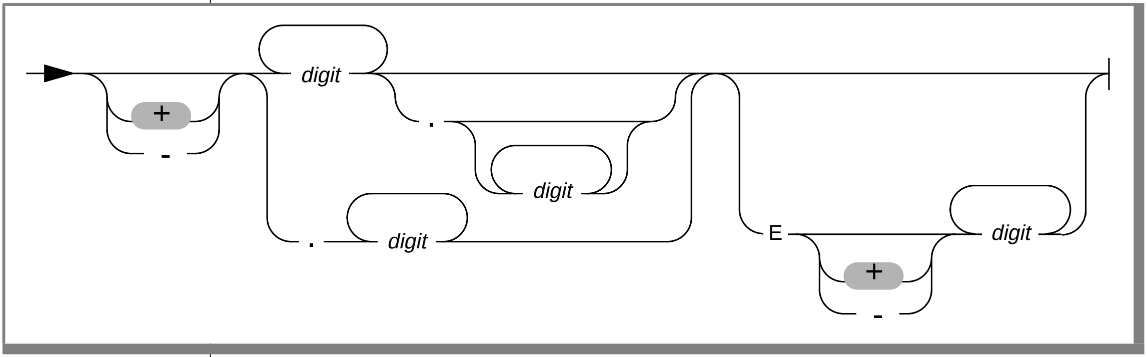
References

In the *Informix Guide to SQL: Reference*, see the INTERVAL data type in Chapter 2 for information about using INTERVAL data in arithmetic and relational operations.

Literal Number

A literal number is an integer or noninteger (floating) constant. Use the Literal Number segment whenever you see a reference to a literal number in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	A digit that forms part of the literal number. See "Floating and Decimal Numbers" on page 1-998 for the significance of digits that follow the decimal point or the E symbol.	You must specify a value between 0 and 9, inclusive.	Digits are literal values that you enter from the keyboard.

Usage

Literal numbers do not contain embedded commas; you cannot use a comma to indicate a decimal point. You can precede literal numbers with a plus or a minus sign.

Integers

Integers do not contain decimal points. The following examples show some integers:

```
10          -27          25567
```

Floating and Decimal Numbers

Floating and decimal numbers contain a decimal point and/or exponential notation. The following examples show floating and decimal numbers:

```
123.456     1.23456E2    123456.0E-3
```

The digits to the right of the decimal point in these examples are the decimal portions of the numbers.

The E that occurs in two of the 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 10 to the minus fifth power.

Literal Numbers and the MONEY Data Type

When you use a literal number as a MONEY value, do not precede it with a money symbol or include commas.

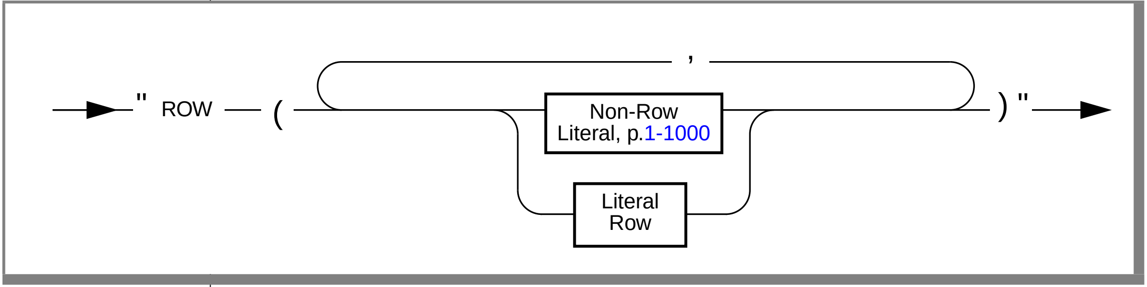
References

See the discussions of numeric data types, such as DECIMAL, FLOAT, INTEGER, and MONEY, in Chapter 2 of the *Informix Guide to SQL: Reference*.

Literal Row

The Literal Row segment specifies the syntax for literal values of named row types and unnamed row types.

Syntax

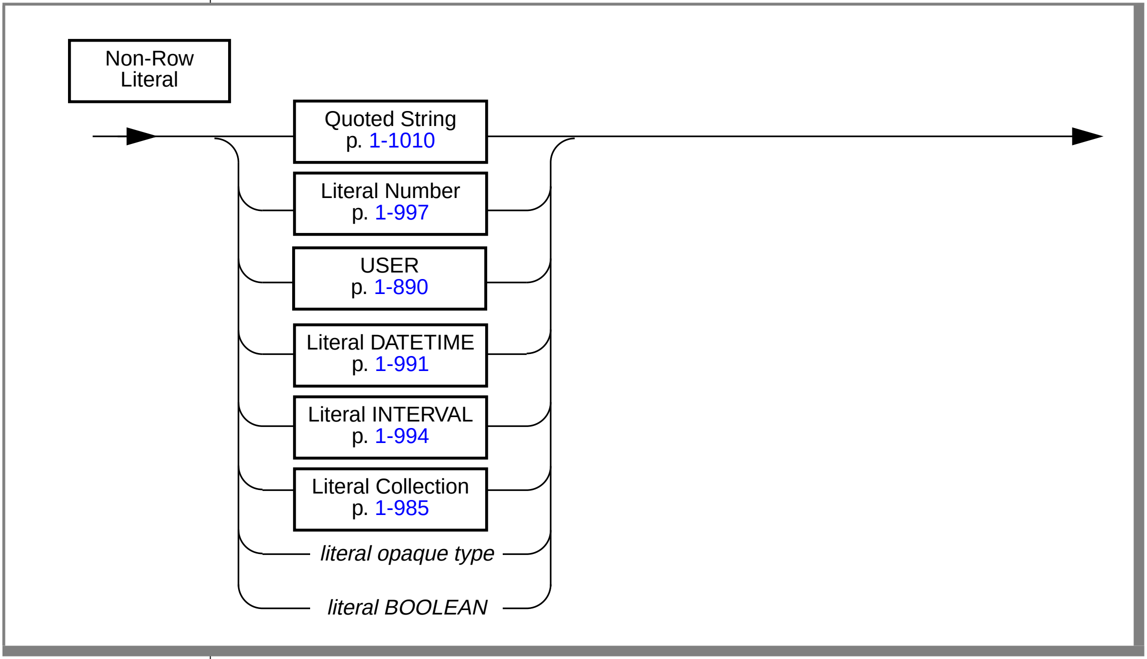


Usage

You can specify literal values for named row types and unnamed row types. The literal row value is introduced with a ROW constructor. The entire literal row value must be enclosed in quotes.

The format of the value for each field of the row type must be compatible with the data type of the corresponding field.

Non-Row Literal Values



Element	Purpose	Restrictions	Syntax
literal opaque type	The 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 type.
literal BOOLEAN	The literal representation of a BOOLEAN value	A literal BOOLEAN value can only be 't' (TRUE) or 'f' (FALSE) and must be specified as a quoted string.	Quoted String, p. 1-1010

Literals of an Unnamed Row Type

To specify a literal value for an unnamed row type, introduce the literal row with the ROW constructor and enclose the values in parentheses. For example, suppose 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. However only the outermost row is enclosed in quotes. For example, suppose you create the following **emp_tab** table:

```
CREATE TABLE emp_tab
(
    emp_name CHAR(10),
    emp_info ROW( stats ROW(x INT, y INT, x 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))" )
```

Field-Level Literal Values

Fields of a row can be literal values for the following built-in data types:

- The CHAR, LVARCHAR, and VARCHAR data types have literal values specified as quoted strings.

  GLS

  The NCHAR and NVARCHAR data types also use quoted strings for literal values. ♦

  For more information, see "Quoted String" on page 1-1010.

- The DECIMAL, FLOAT, INT8, INTEGER, MONEY, SMALLFLOAT and SMALLINT data types have literal values specified by the Literal Number syntax.

  For more information, see "Literal Number" on page 1-997.

- The DATE data type has a literal value specified as a quoted string.

  For more information, see "Quoted String" on page 1-1010.

- The DATETIME data type has literal values specified by the Literal DATETIME syntax.

  For more information, see "Literal DATETIME" on page 1-991.

- The INTERVAL data type has literal values specified by the Literal INTERVAL syntax.

  For more information, see "Literal INTERVAL" on page 1-994.
- The BOOLEAN data type has the literal values of 't' and 'f'.

  For more information, see the description of the BOOLEAN data type in Chapter 2 of the *Informix Guide to SQL: Reference*.

Important: *You cannot specify the simple-large-object data types (BYTE and TEXT) as the field type for a row.*

Field values can also be literal values for the following user-defined data types:

- An opaque data type has a literal value that corresponds to the string representation for the opaque data type. The input support function of the opaque type defines this string representation
- A collection type (SET, MULTISET, or LIST) has a literal value whose specification depends on the context its use.

  For information on literal collection value as a column value or as a collection-variable value, see "Example of Nested Quotation Marks" on page 1-989.
- Another row type, named or unnamed, has a literal value specified as field values enclosed with parentheses.

  A row with a field whose data type is another row is called a *nested row*.

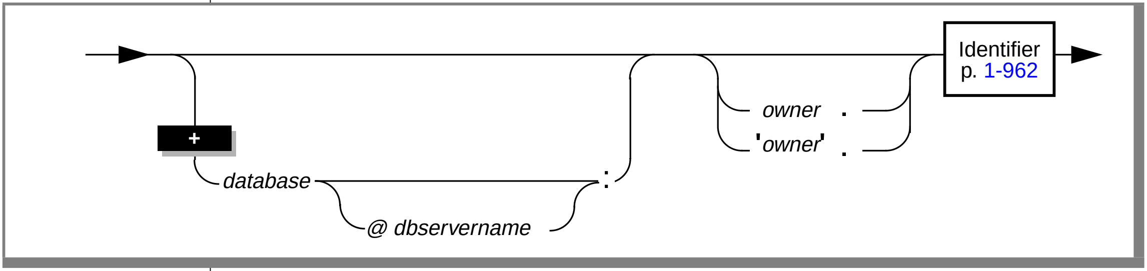
References

See the INSERT, UPDATE, and SELECT statements in this manual. See the CREATE ROW TYPE statement for information on named row types. See "Constructor Expressions" on page 1-895 of the Expression segment for information on ROW constructors. See also the Collection Literal segment.

Procedure Name

The Procedure Name segment specifies the name of a procedure.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the procedure resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the procedure	If you are using an ANSI-compliant database, you must specify an owner for a procedure you do not own. If you do not specify an owner, the default owner is the current user.	The user name must conform to the conventions of your operating system.

Usage

In a statement that calls for a Procedure Name, you can enter an identifier with an optional owner name, database name, and server name. The database and server names allow you to use a procedure stored on a remote database. A Procedure Name with a database name, server name, and owner name is called a fully qualified procedure name.

The actual name of the procedure is an SQL identifier.

ANSI

The owner name is case sensitive. In an ANSI database, if you type quotation marks around the name, it is stored as you type it. If you do not use quotation marks, the name is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of procedures. For more information, see the *Guide to GLS Functionality*. ♦

Routine Overloading

Due to routine overloading, a procedure name does not need to be unique in Universal Server. You can define more than one procedure with the same name and different parameter lists.

Procedures are uniquely identified by their signature. A procedure's signature includes the following items:

- The routine type (FUNCTION or PROCEDURE)
- The routine name
- The number, data type, and order of the procedure's parameters

If a procedure name is not unique, Universal Server uses routine resolution to identify the instance of the procedure to execute. For more information about routine resolution, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

Database Name and Server Name

When you add the database name and server name options, you use a fully qualified procedure name to specify a remote procedure. You can use those options when:

- All arguments passed to the procedure have built-in data types.
- The instance of the procedure that is invoked has built-in data types for all of its parameters.

References

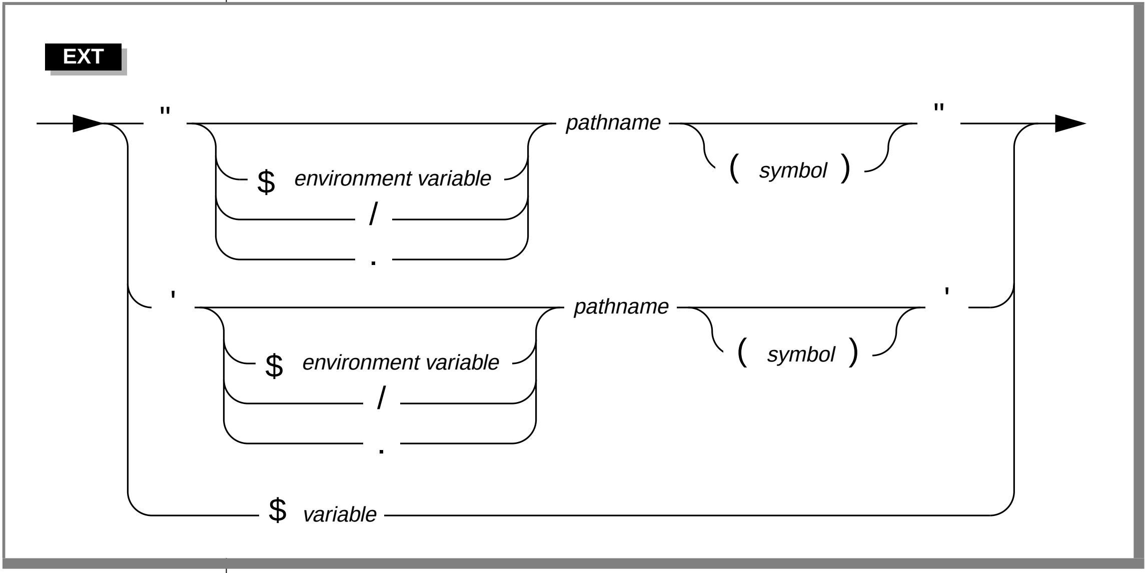
In this manual, see the CREATE FUNCTION, CREATE PROCEDURE, DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, EXECUTE FUNCTION, and EXECUTE PROCEDURE statements. See also the Function Name and Specific Name segments.

In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to create and use SPL routines.

Quoted Pathname

Use a Quoted Pathname to supply a pathname to an executable object file when you register an external routine.

Syntax



Element	Purpose	Restrictions	Syntax
environment variable	A platform-independent location indicator	The *environment variable* name must begin with a dollar sign and be the first word in the *pathname.*	Identifier, p. 1-962
pathname	The pathname to the dynamically loadable executable file	An absolute pathname must begin with a forward slash. A relative pathname need not begin with a period. Each directory name must end with a forward slash. The filename at the end of the pathname must end in **.so** and must refer to an executable file in a shared object library.	Identifier, p. 1-962
symbol	An optional entry point to the dynamically loadable executable	Use a symbol only if the entry point has a different name than the routine you are registering with CREATE FUNCTION or CREATE PROCEDURE. You must enclose a symbol in parentheses.	Identifier, p. 1-962
variable	A platform-independent location indicator that contains the full pathname to the executable object file	You must begin the variable name with a dollar sign.	Identifier, p. 1-962

Usage

A Quoted Pathname must be enclosed in single or double quotation marks. The opening and closing quotation marks must match. The filename in the Quoted Pathname must end in **.so**, because it refers to an executable object file in a shared library.

You can omit a pathname, and enter just a filename, if you want to refer to an internal function.

A pathname can begin with an environment variable, used as a location indicator. An environment variable begins with a dollar sign, and must be the first element in the pathname.

A pathname can also be absolute or relative. An absolute pathname always begins with a forward slash. A relative pathname need not begin with a period, and is relative from the current directory at the time the CREATE PROCEDURE or CREATE FUNCTION statement is run.

If you use a *symbol*, it refers to an optional entry point in the executable object file. Use a symbol only if the entry point has a name other than the name of the routine that you are registering with CREATE PROCEDURE or CREATE FUNCTION.

You can also specify the full pathname as a variable that begins with a dollar sign.

You can include spaces or tabs within a Quoted Pathname.

References

In this manual, see the CREATE FUNCTION and CREATE PROCEDURE statements and the External Routine Reference segment.

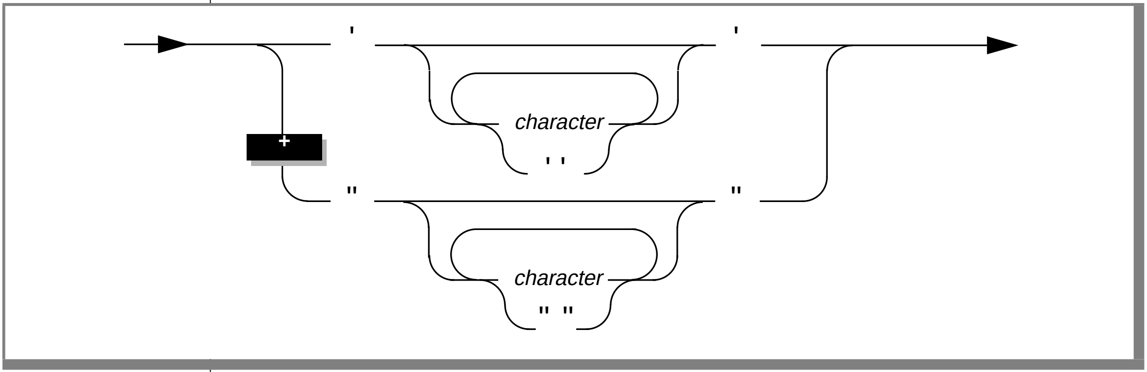
For information about how to create and use user-defined routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual. In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to create and use SPL routines.

See the *Extending INFORMIX-Universal Server: Data Types* manual for information about how to create external routines that define opaque data types and distinct data types.

Quoted String

A quoted string is a string constant that is surrounded by quotation marks. Use the Quoted String segment whenever you see a reference to a quoted string in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
character	A character that forms part of the quoted string	The character or characters in the quoted string cannot be surrounded by double quotes if the **DELIMIDENT** environment variable is set. For additional restrictions see "Restrictions on Specifying Characters in Quoted Strings" on page 1-1011.	Characters are literal values that you enter from the keyboard.

Restrictions on Specifying Characters in Quoted Strings

You must observe the following restrictions when you specify *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 within quoted strings, see "Using Quotes in Strings" on page 1-1012.

GLS

- If you are using a nondefault locale, you can specify non-ASCII characters, including multibyte characters, that the code set of your locale supports. See the discussion of quoted strings in the *Guide to GLS Functionality* for further information. ♦
- When you set the **DELIMIDENT** environment variable, you cannot use double quotes to delimit a quoted string. When **DELIMIDENT** is set, a string enclosed in double quotes is an identifier, not a quoted string. When **DELIMIDENT** is not set, a string enclosed in double quotes is a quoted string, not an identifier. See "Using Quotes in Strings" on page 1-1012 for further information.
- You can enter DATETIME and INTERVAL data as quoted strings. See "DATETIME and INTERVAL Values as Strings" on page 1-1012 for the restrictions that apply to entering DATETIME and INTERVAL data in quoted-string format.
- 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. See "LIKE and MATCHES in a Condition" on page 1-1012 for further information.
- When you insert a value that is a quoted string, you must observe a number of restrictions. See "Inserting Values as Quoted Strings" on page 1-1013 for further information.

Usage

The string constant must be written on a single line; that is, you cannot use embedded new lines.

Using Quotes in Strings

The single quote has no special significance in string constants delimited by double quotes. Likewise, the double quote has no special significance in strings delimited by single quotes. For example, the following strings are valid:

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "no!"'
```

If your string is delimited by double quotes, you can include a double quote in the string by preceding the double quote with another double quote, as shown in the following string:

```
"Enter ""y"" to select this row"
```

When the **DELIMIDENT** environment variable is set, double quotes delimit identifiers, not strings. See "Delimited Identifiers" on page 1-965 for further information on delimited identifiers.

DATETIME and INTERVAL Values as Strings

You can enter DATETIME and INTERVAL data in the literal forms described in the "Literal DATETIME" and "Literal INTERVAL" segments beginning on pages 1-991 and 1-994, respectively, or you can enter them as quoted strings. Valid literals that are entered as character strings are converted automatically into DATETIME or INTERVAL values. The following INSERT statements use quoted strings to enter INTERVAL and DATETIME data:

```
INSERT INTO cust_calls(call_dtime) VALUES ('1993-5-4 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 qualifiers of the column. For the first case in the preceding example, **call_dtime** must be defined with the qualifiers YEAR TO MINUTE 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. See the "Condition" segment beginning on page 1-831 for a complete description of how to use wildcard characters.

Inserting Values as Quoted Strings

If you are inserting a value that is a quoted string, you must adhere to the following conventions:

- Enclose CHAR, VARCHAR, NCHAR, NVARCHAR, DATE, DATETIME, and INTERVAL values in quotation marks.
- Set DATE values in the *mm/dd/yy format*.
- You cannot insert strings longer than 256 bytes.
- Numbers with decimal values must contain a decimal point. You cannot use a comma as a decimal indicator.
- You cannot precede MONEY data with a dollar sign ($) or include commas.
- You can include NULL as a placeholder only if the column accepts null values.

References

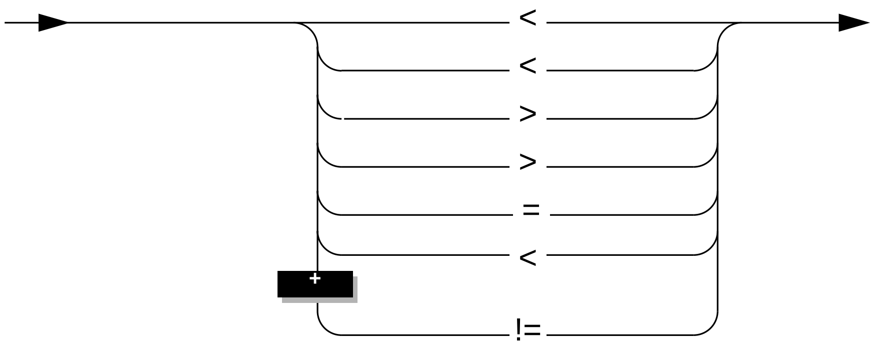
In the *Informix Guide to SQL: Reference*, see the discussion of the **DELIMIDENT** environment variable in Chapter 3.

In the *Guide to GLS Functionality*, see the discussion of quoted strings.

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



Each operator shown in the syntax diagram has a particular meaning.

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 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.

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. See the *Guide to GLS Functionality* for further information on locale-based collation order and the NCHAR and NVARCHAR data types. ♦

Using Operator Functions in Place of Relational Operators

Each relational operator is bound to a particular function, as shown in the table below. The function accepts two values and returns a boolean value of true, false, or unknown.

Relational Operator	Associated Function
<	**lessthan()**
<=	**lessthanorequal()**
>	**greater than()**
>=	**greaterthanorequal()**
=	**equal()**
<>	**notequal()**
!=	**notequal()**

Connecting two expressions with a binary operator is equivalent to invoking the function on the expressions. For example, the following 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 functions associated with the relational operators for all built-in data types. When you develop a user-defined data type, you must define the functions for that type for users to be able to use the relational operator on the type.

Collating Order for 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. On UNIX platforms, 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 ¿).

The following table shows the ASCII code set. The Num column shows the ASCII code numbers, and the Char column shows the ASCII character corresponding to each ASCII code number. ASCII characters are sorted according to their ASCII code number. Thus lowercase letters follow uppercase letters, and both follow numerals. In this table, the caret symbol (^) stands for the CTRL key. For example, ^X means CTRL-X.

Num	Char	Num	Char	Num	Char
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	943	]
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	`
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i

(1 of 2)

Num	Char	Num	Char	Num	Char
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	esc	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32		75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	'	82	R	125	}
40	(	83	S	126	~
41	)	84	T	127	del
42	*	85	U		

(2 of 2)

Support for ASCII Characters in Nondefault Code Sets

GLS

Most code sets in 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, as long as the nondefault code set supports these ASCII characters. ♦

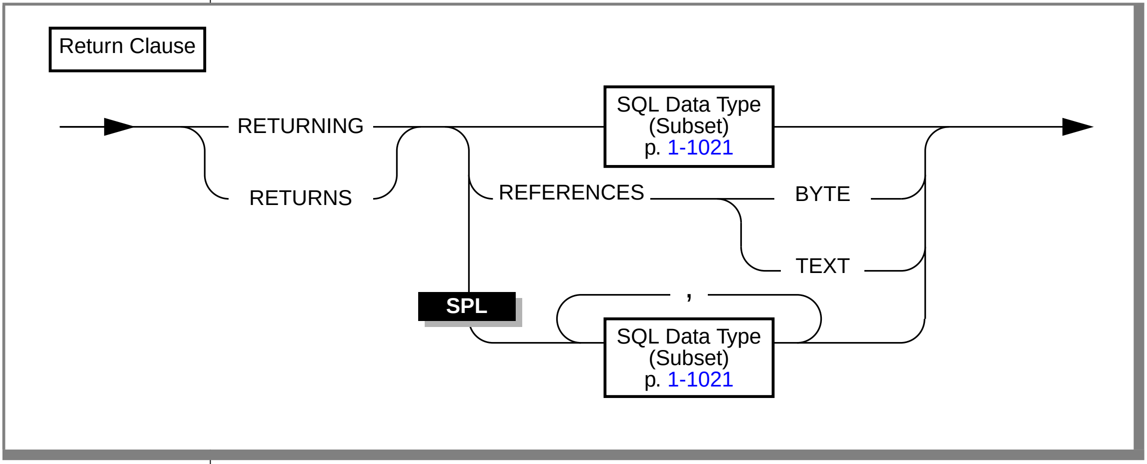
References

In the *Informix Guide to SQL: Tutorial*, see the discussion of relational operators in the SELECT statement in Chapter 2.

In the *Guide to GLS Functionality*, see the discussion of relational operator conditions in the SELECT statement.

Return Clause

Syntax


Return Clause
RETURNING
RETURNS
SQL Data Type (Subset) p. 1-1021
REFERENCES
BYTE
TEXT
SPL
,
SQL Data Type (Subset) p. 1-1021

Usage

The Return clause is used in the CREATE FUNCTION statement to specify the data types of the value or values that a user-defined function returns. In the Return clause, you can use the keywords RETURNING and RETURNS interchangeably.

If you overload functions in your database, the data type of the return value is subject to routine resolution. For more information on routine resolution, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual.

SPL

For an SPL function, you can specify more than one data type in the Return clause.

If you write a stored procedure (a legacy SPL function), you can use a Return clause with the CREATE PROCEDURE statement. However, Informix recommends that you create new SPL functions with the CREATE FUNCTION statement. Any routine you create with a Return clause is considered a function. ♦

EXT

For an external function, specify exactly one value in the Return clause. However, an external function can return more than one row of data if it is an iterator function. For more information, see the description of the ITERATOR routine modifier in the Routine Modifier segment. ♦

SQL Data Types (Subset)

SPL functions and external functions can return values of any data type defined in the database, except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.

An external or SPL function can return values of COLLECTION, SET, MULTISET, LIST, or ROW data type, as the following example shows:

```
CREATE FUNCTION add_types( a dollar, b yen )
   RETURNING collection1, row1 ... ;
```

The calling routine must define variables of the appropriate complex types to hold the values the function returns.

A function can also return a value or values of an opaque or distinct data type that the database defines.

Referencing a Simple Large Object

Neither an SPL function nor an external function can return a TEXT or BYTE value (collectively called *simple large objects*) directly. A function can, however, use the REFERENCES keyword to return a descriptor that contains a pointer to a TEXT or BYTE object.

A function cannot return a CLOB or BLOB, or a pointer to a CLOB or BLOB.

References

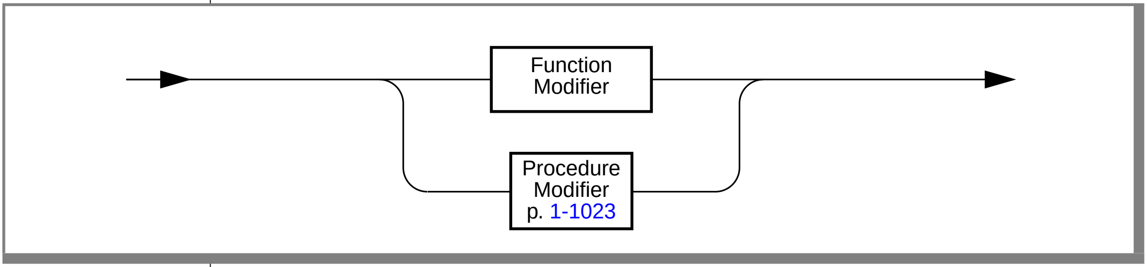
See the CREATE FUNCTION, CREATE PROCEDURE, EXECUTE FUNCTION, EXECUTE PROCEDURE, and CALL statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to create and use SPL functions.

See the *Extending INFORMIX-Universal Server: User-Defined Routines* manual for information about how to create and use external functions.

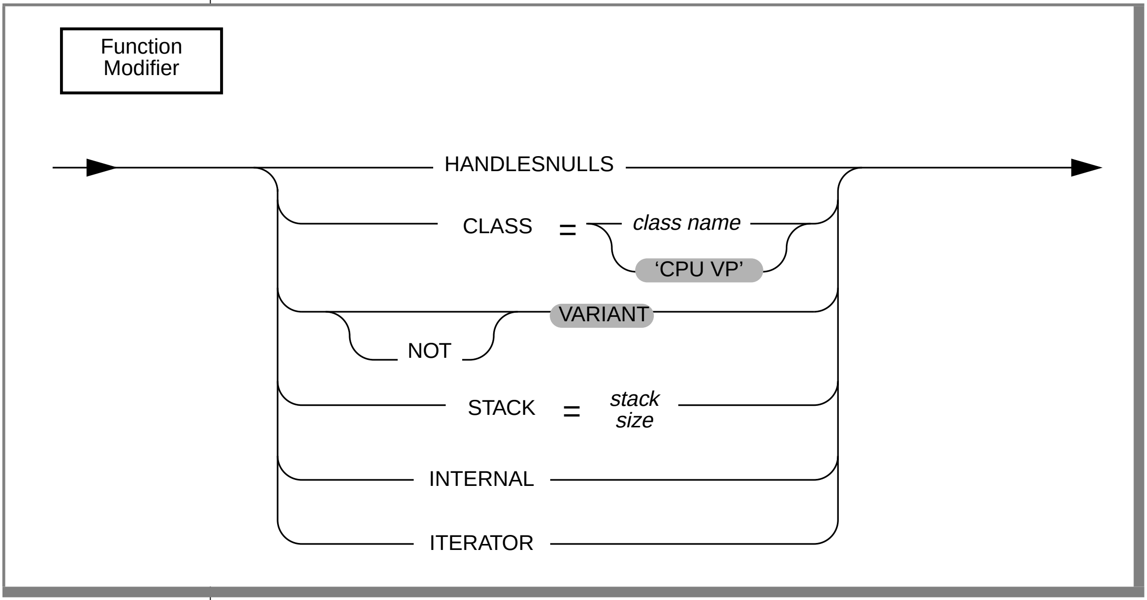
Routine Modifier

Use a Routine Modifier clause to specify attributes of a user-defined routine behaves. The Routine Modifier clause can be a Function Modifier or a Procedure Modifier.



Function Modifier

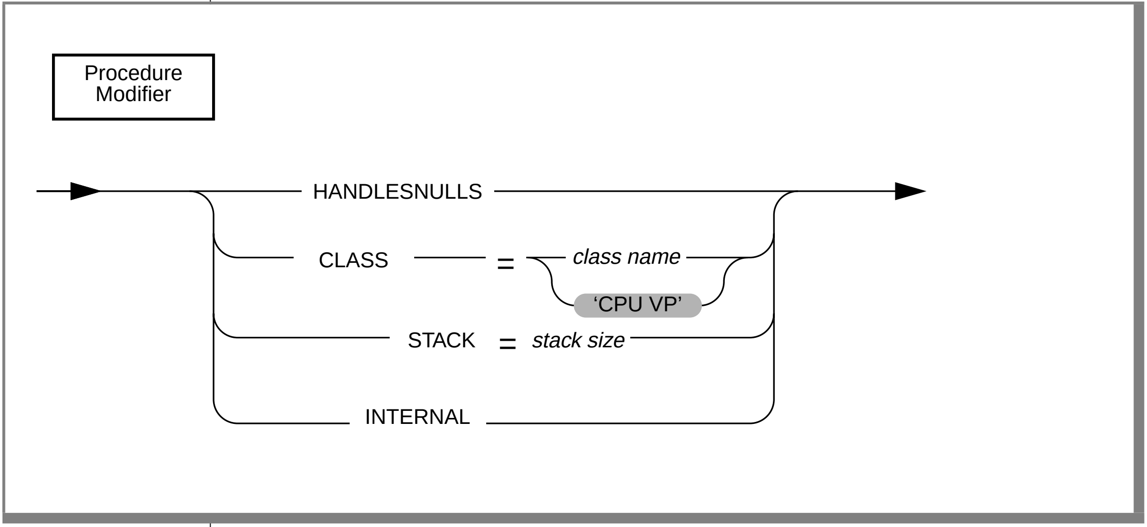
Function modifiers are valid in the WITH clause of the CREATE FUNCTION statement to register a user-defined function.



Element	Purpose	Restrictions	Syntax
class name	The name of the virtual processor class in which the routine is to run	An external function must run in the CPU VP or in an external VP (EVP) class. If you specify an EVP class, the class must already be defined. You must enclose the *class name* in single or double quotation marks.	Quoted String, p. 1-1010
stack size	The size of the stack while the routine is running	The *stack size* must be a positive integer and gives the stack size in bytes. The stack size should be larger than the stack size specified in the STACKSIZE configuration parameter.	Literal Number, p. 1-997

Procedure Modifier

Procedure modifiers are valid in the WITH clause of the CREATE PROCEDURE statement to register a user-defined procedure.



Element	Purpose	Restrictions	Syntax
class name	The name of the virtual processor class in which the routine is to run	An external procedure must run in the CPU VP or in an external VP (EVP) class. If you specify an EVP class, the class must already be defined. You must enclose the *class name* in single or double quotation marks.	Quoted String, p. 1-1010
stack size	The size of the stack while the routine is running	The *stack size* must be a positive integer and gives the stack size in bytes. The stack size should be larger than the stack size specified in the STACKSIZE configuration parameter.	Literal Number, p. 1-997

Modifier Descriptions

The following sections describe each of the routine modifiers.

HANDLESNULLS

EXT

You can use the HANDLESNULLS modifier with both external functions and external procedures. HANDLESNULLS specifies that an external routine can handle NULL values passed to it as arguments. If you do not specify HANDLESNULLS, and if you pass an argument with a NULL value to the routine, the routine does not execute and returns a NULL value.

By default, HANDLESNULLS is not set for external routines. That is, the database server assumes that an external routine does not handle null arguments. If your external routine handles NULL values, specify HANDLESNULLS in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement. ♦

SPL

Do *not* use HANDLESNULLS with SPL routines. SPL routines handle NULL values by default. You do not need to specify HANDLESNULLS so that an SPL routine handles NULL values. ♦

CLASS

EXT

You can use the CLASS modifier with both external functions and external procedures. The CLASS modifier runs the external routine in a virtual processor class (VP class) that you specify. The purpose of setting up classes of virtual processors is to group sets of routines, so that the routines in a group execute within the same context.

You can run external routines written in C in the CPU virtual processor (CPU VP) class or in an external virtual processor (EVP) class that you name with the CLASS modifier. If you do not specify a VP class, the external routine runs in the CPU VP class by default.

If an external routine is *ill-behaved*, you must run it in a class other than the CPU VP. A routine is ill-behaved if it does any of the following:

- Runs for a long time without yielding
- Makes an operating system call that can block other calls, for example, READ, WRITE, SELECT, POLL, PUTMSG, BGETMSG, SEMOP, MSGGET, PAUSE, and WAIT
- Modifies the global state of the virtual processor on which it is running by:
    - Modifying global or static data
    - Opening a file descriptor using OPEN, DUP, IOCTL, SOCKET, or similar commands
    - Changing the current working directory
    - Calling the **brk()**, **sbrk()**, or **malloc()** functions
    - Using operating system threads
- Calls one of the following ill-behaved C library functions: **stdio()**, **getpwent()**, and **gethostbyname()**.

Use the CLASS modifier in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement to name an external VP class for any routine that might cause the database server to hang, stop running, or behave erratically. ♦

SPL

Do not use CLASS with SPL routines. SPL routines always run in the CPU VP. ♦

VARIANT and NOT VARIANT

You can use VARIANT and NOT VARIANT with user-defined functions, both external 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. To define a non-variant function, specify NOT VARIANT in the WITH clause of the CREATE FUNCTION statement. If the function is non-variant, the database server may cache the return values of expensive functions or run parallel queries. You can create functional indexes only on non-variant functions. For more information on functional indexes, see the CREATE INDEX statement. ♦

EXT

You can specify VARIANT or NOT VARIANT in the Routine Modifier clause or in the External Routine Reference clause, which is described on page 1-956. If you specify the modifier in both places, you must use the same modifier in both. ♦

STACK

EXT

You can use the STACK modifier with both external procedures and external functions. The STACK modifier enables the database server to run an external routine in a stack that is larger than the stack size that the STACKSIZE configuration parameter specifies. When the external routine executes, the database server increases the stack size of the routine to the number of bytes specified. When the routine completes, the original stack size is restored. Specify a larger stack size to prevent stack overflow. ♦

SPL

You cannot use the STACK modifier with SPL routines. ♦

INTERNAL

EXT

You can use the INTERNAL modifier with both external procedures and external functions. The INTERNAL modifier specifies that an SQL or SPL statement cannot call the external routine. A routine that is specified INTERNAL is not considered during routine resolution. Use INTERNAL for external routines that define access methods, language managers, and so on. For more information on how to write iterator functions, see the *DataBlade API Programmer's Manual*.

By default, an external routine is *not* internal; that is, an SQL or SPL statement can call the routine. To define an internal function, specify INTERNAL in the WITH clause of the CREATE FUNCTION or CREATE PROCEDURE statement. ♦

SPL

You cannot use the INTERNAL modifier with SPL routines. ♦

ITERATOR

EXT

Use the ITERATOR modifier only with external functions. This modifier is *not* valid for external procedures. The ITERATOR modifier specifies that the external function is an *iterator function*; that is, it returns a set of values and is invoked repeatedly by the database server. An iterator function is similar to an SPL function that contains the RETURN WITH RESUME statement.

By default, an external function is *not* an iterator. To define an iterator function, specify ITERATOR in the WITH clause of the CREATE FUNCTION statement. ♦

SPL

You cannot use the ITERATOR modifier with SPL routines. ♦

E/C

Both an iterator function and an SPL function with RETURN WITH RESUME require a cursor to be executed. The cursor allows the client application to retrieve the values one at a time with the FETCH statement. ♦

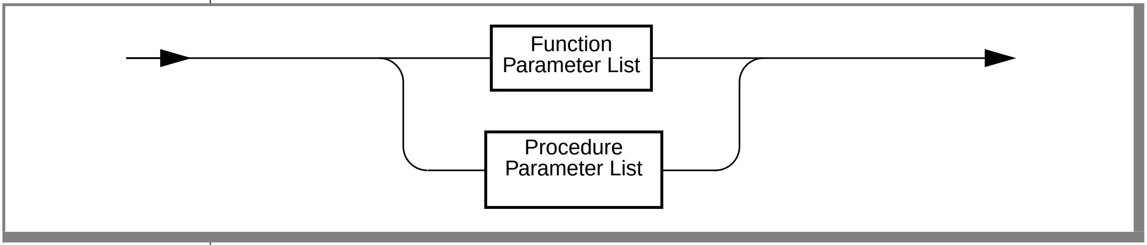
References

In this manual, see the CREATE FUNCTION and CREATE PROCEDURE statements.

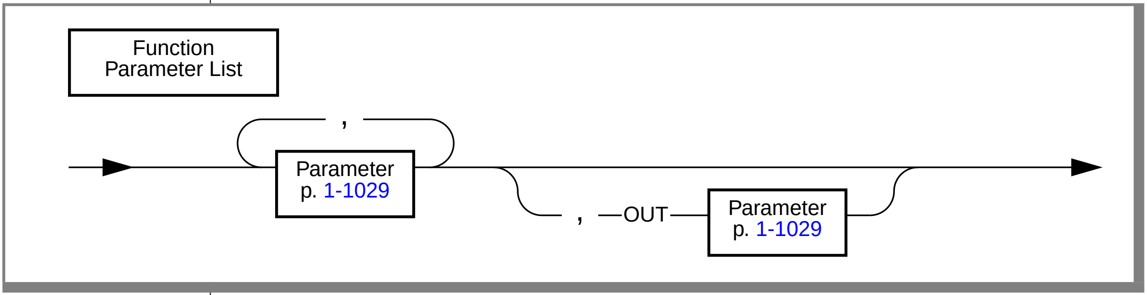
For information about how to create and use external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual and the *DataBlade API Programmer's Manual*. For information about how to create and use SPL routines, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.

Routine Parameter List

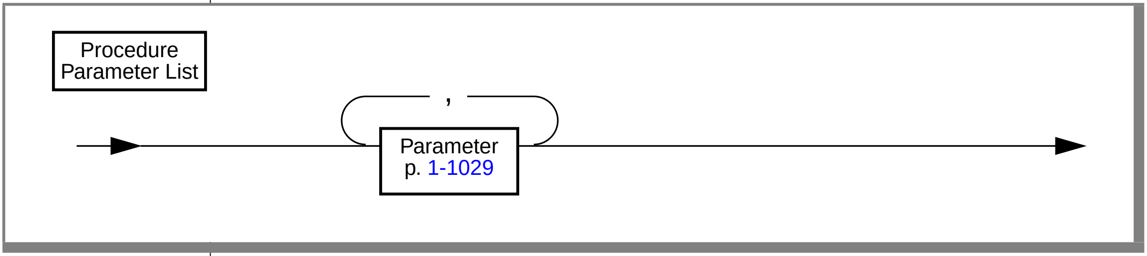
Use the Function Parameter List to define the parameters an external function or SPL function can accept. Use the Procedure Parameter List to define the parameters an external procedure or SPL procedure can accept.



Function Parameter List

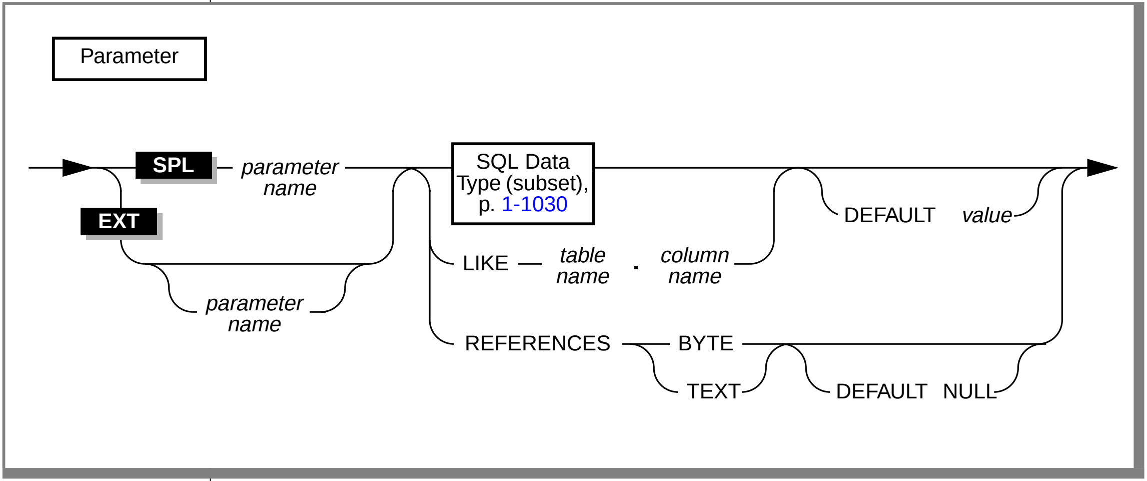


Procedure Parameter List



Parameter

A Parameter is one item in a Function Parameter List or Procedure Parameter List.



Element	Purpose	Restrictions	Syntax
column name	The name of a column whose data type is assigned to the parameter	The column must exist in the specified table.	Identifier, p. 1-962
parameter name	The name of a parameter the routine can accept	The *parameter name* is required for SPL routines and optional for external routines in the CREATE FUNCTION and CREATE PROCEDURE statements.	Identifier, p. 1-962
table name	The name of the table that contains *column name*	The table must exist in the database.	Identifier, p. 1-962
value	The default value that a routine uses if you do not supply a value for the parameter when you call the routine	This value must be a literal. If *value* is a literal, the value must have the same data type as *parameter name*. If *value* is a literal and its type is an opaque type, an input function must be defined on the type.	Literal Number, p. 1-997

Usage

To define a parameter when creating a routine, specify its name (required for SPL routines; optional for external routines) and its data type. The data type can be any data type in the database, except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.

The data type can be the name of an opaque, distinct, or row type you have defined. The data type can also be COLLECTION, SET, MULTISET, LIST, or ROW with the definition of an unnamed row type. For the complete syntax of all the SQL data types, see "Data Type" on page 1-855.

SQL Data Type (Subset)

A routine can define a parameter of any data type defined in the database, except SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB. A parameter can have an opaque type, distinct type, built-in type (except the excluded data types just listed), collection type, named row type, or unnamed row type.

A parameter can also be of type COLLECTION, which is a generic collection type that can accept any SET, MULTISET, or LIST as an argument.

For more information on defining data types for parameters, see "Data Type" on page 1-855.

LIKE Clause

The LIKE keyword specifies that the data type of a parameter is the same as a column defined in the database and changes with the column definition. If you define a parameter with LIKE, the parameter's data type changes as the data type of the column changes.

If any of the arguments for the routine are defined using the LIKE clause, you cannot overload the routine and the routine will not be considered in the routine resolution process.

For example, suppose you create the following routine:

```
CREATE PROCEDURE cost (a LIKE tab.col, b INT)
.
.
.
END PROCEDURE;
```

Now you cannot create another routine named **cost()** in the same database with two arguments. However, you can create a routine named **cost()** with a number of arguments other than two.

REFERENCES Clause

Use the REFERENCES clause to specify that a parameter contains TEXT or BYTE data.

The REFERENCES keyword allows you to use a pointer to a TEXT or BYTE object as a parameter. You cannot use a TEXT or BYTE object directly.

If you use the DEFAULT NULL option in the REFERENCES clause, and you call the routine without a parameter, a null value is used.

Default Value

You can 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 routine is called with fewer arguments than were defined for that routine, the default value is used. If you do not provide a default value for a parameter, and the routine is called with fewer arguments than were defined for that routine, 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 procedure 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;
```

***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, the **DBCENTURY** environment variable can affect how the database server interprets the date value, so the routine might not use the default value that you intended. See the "Informix Guide to SQL: Reference" for more information on the **DBCENTURY** environment variable.*

Specifying an OUT Parameter (Functions Only)

When you register an external function written in C, you can specify that the last parameter in the list is an OUT parameter. The OUT parameter corresponds to a value the function returns indirectly, through a pointer. The value the function returns through the pointer is an extra value, in addition to the value it returns explicitly.

Once you register a function with an OUT parameter, you can use the function with a Statement Local Variable (SLV) in an SQL statement. You can only mark one parameter as OUT, and it must be the last parameter.

For example, the following declaration of a C language function allows you to return an extra value through the **y** parameter:

```
int my_func( int x, int *y );
```

You would register the function with a CREATE FUNCTION statement similar to this one:

```
CREATE FUNCTION my_func( x INT, OUT y INT )
RETURNING INT
EXTERNAL NAME "/usr/lib/local_site.so"
LANGUAGE C
END FUNCTION;
```

If you specify an OUT parameter, and if you use Informix-style parameters, the argument is passed to the OUT parameter by reference.

The OUT parameter is not significant in determining the routine signature.

References

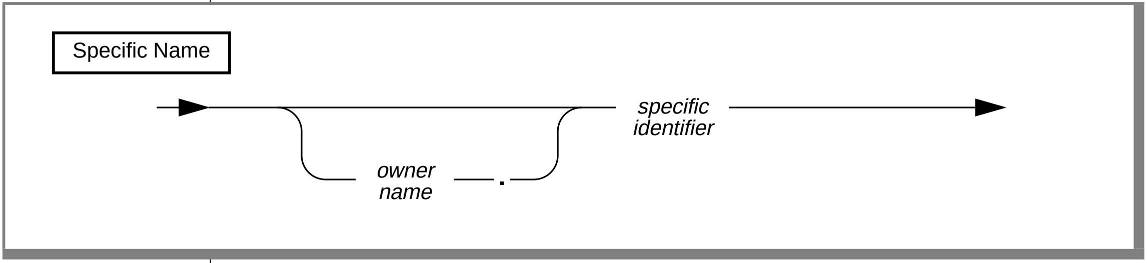
In this manual, see the CREATE FUNCTION and CREATE PROCEDURE statements. See also the Argument segment.

For information about how to create and use external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual. For information about how to create and use SPL routines, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.

Specific Name

Use a Specific Name to give a routine a name that is unique in the database or name space.

Syntax



Element	Purpose	Restrictions	Syntax
owner name	The name of the owner of the routine	This name, if used, must be the same *owner name* used in the Function Name or Procedure Name for this routine. If no *owner name* is specified in the routine name, then the *owner name* you use in the Specific Name must be the user id of the person creating the routine. If you omit *owner name*, the server uses the user id of the person creating the routine.	Identifier, p. 1-962
specific identifier	The unique name of the routine	In a non-ANSI database, the *specific identifier* must be unique within the database. In other words, two specific names cannot have the same *specific identifier* even if they have two different owners. In an ANSI database, the *specific identifier* must be unique for the owner. In other words, the same *specific identifier* can be used for two routines within the same database if the routines have different owners. The specific identifier can be up to 128 characters long.	Identifier, p. 1-962

Usage

A Specific Name is a unique identifier that you define in a CREATE PROCEDURE or CREATE FUNCTION statement to serve as an alternate name for a routine.Because you can create user-defined routines to overload routines, a database can have more than one routine with the same name and different parameter lists. You can assign a routine a Specific Name that uniquely identifies the specific routine.

If you give a routine a Specific Name when you create it, you can later alter or drop that routine using the Specific Name only. Otherwise, you need to include the parameter data types with the routine name when you drop the routine, if the routine name alone does not uniquely identify the routine.

You can use a Specific Name in the following SQL statements:

- DROP
- GRANT
- REVOKE
- UPDATE STATISTICS

In an ANSI database, you can use the same specific identifier for two routines within the same database if the routines have different owners.

The Specific Name must be unique within the name space in which it is created:

- For ANSI-compliant databases, the name space is the schema.
- For databases that are not ANSI compliant, the name space is the database.

References

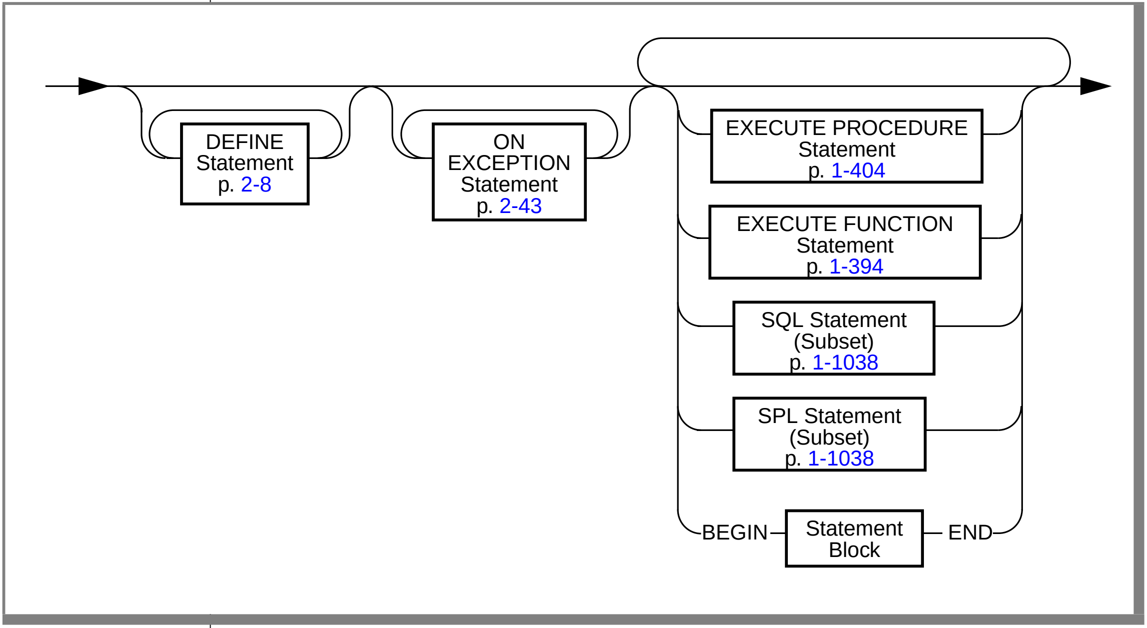
In this manual, see the CREATE FUNCTION, CREATE PROCEDURE, DROP FUNCTION, DROP PROCEDURE, DROP ROUTINE, EXECUTE FUNCTION, and EXECUTE PROCEDURE statements. See also the Function Name and Procedure Name segments.

For information about how to create and use external routines, see the *Extending INFORMIX-Universal Server: User-Defined Routines* manual. For information about how to create and use SPL routines, see Chapter 14 in the *Informix Guide to SQL: Tutorial*.

Statement Block

Use a Statement Block, instead of an External Routine Reference, when you write an SPL routine.

Syntax


DEFINE Statement p. 2-8
ON EXCEPTION Statement p. 2-43
EXECUTE PROCEDURE Statement p. 1-404
EXECUTE FUNCTION Statement p. 1-394
SQL Statement (Subset) p. 1-1038
SPL Statement (Subset) p. 1-1038
BEGIN
Statement Block
END

Usage

If the statement block portion of the statement is empty, no operation takes place when you call the routine. You might use such a routine in the development stage when you want to establish the existence of a routine but have not yet coded it.

Also, you cannot close the current database or select a new database within a routine. And you cannot drop the current stored routine within a stored routine. You can, however, drop another routine.

Subset of SQL Statements Allowed in the Statement Block

You can use any SQL statement in the statement block, except those listed in the following table.

Figure 1-7
SQL Statements That Cannot Be Used in an SPL Routine

ALLOCATE COLLECTION	EXECUTE
ALLOCATE DESCRIPTOR	EXECUTE IMMEDIATE
ALLOCATE ROW	FETCH
CLOSE	FLUSH
CLOSE DATABASE	FREE
CONNECT	GET DESCRIPTOR
CREATE DATABASE	INFO
CREATE FUNCTION	LOAD
CREATE FUNCTION FROM	OPEN
CREATE PROCEDURE	OUTPUT
CREATE PROCEDURE FROM	PREPARE
CREATE ROUTINE	PUT
DEALLOCATE COLLECTION	ROLLFORWARD DATABASE
DEALLOCATE DESCRIPTOR	SET CONNECTION
DEALLOCATE ROW	SET DESCRIPTOR
DECLARE	UNLOAD
DESCRIBE	WHENEVER
DISCONNECT	

Subset of SPL Statements Allowed in the Statement Block

You can use any of the following SPL statements in the statement block.

Figure 1-8
SPL Statements that Can Be Used in an SPL Routine

CALL	LET
CONTINUE	RAISE EXCEPTION
EXIT	RETURN
FOR	SYSTEM
FOREACH	TRACE
IF	WHILE

Restrictions on SELECT Statement

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.

Support for Roles and User Identity

You can use roles with routines you create. You can execute role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements within a routine. You can also grant privileges to roles with the GRANT statement within a routine. Privileges that a user has acquired through enabling a role or by a SET SESSION AUTHORIZATION statement are not relinquished when a routine is executed.

For further information about roles, see the CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE statements in this guide.

Restrictions on a Routine Called in a Data Manipulation Statement

If a routine is called as part of an INSERT, UPDATE, DELETE, or SELECT statement, the called routine cannot execute any statement listed in Figure 1-9. This restriction ensures that the routine cannot make changes that affect the SQL statement that contains the routine call.

Figure 1-9
SQL Statements Not Allowed in an SPL Routine That a Data Manipulation Statement Calls

ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
BEGIN WORK
COMMIT WORK
CREATE TRIGGER
DELETE
DROP DATABASE
DROP INDEX
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP VIEW
INSERT
RENAME COLUMN
RENAME TABLE
ROLLBACK WORK
SET CONSTRAINTS
UPDATE

For example, if you use the following INSERT statement, the execution of the called procedure **dup_name** is restricted:

```
CREATE PROCEDURE sp_insert ()
.
.
.
INSERT INTO q_customer
    VALUES (SELECT * FROM customer
        WHERE dup_name ('lname') = 2)
.
.
.
END PROCEDURE;
```

In this example, **dup_name** cannot execute the statements listed in Figure 1-9. However, if **dup_name** is called within a statement that is not an INSERT, UPDATE, SELECT, or DELETE statement (namely EXECUTE PROCEDURE), **dup_name** can execute the statements listed in Figure 1-9.

You can use the BEGIN WORK and COMMIT WORK statements in procedures. You can start a transaction, finish a transaction, or start and finish a transaction in a procedure. If you start a transaction in a procedure that is executed remotely, you must finish the transaction before the procedure exits.

Warning: *When you specify a date value in an expression in any statement in the statement block, 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, the **DBCENTURY** environment variable can affect how the database server interprets the date value, so the routine might produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the **DBCENTURY** environment variable.*

Adding Comments to an SPL Routine

To add a comment to any line of a routine, place a double-dash (--) before the comment or enclose the comment in braces ({}). The double dash complies with the ANSI standard. The curly brackets are an Informix extension to the ANSI standard.

References

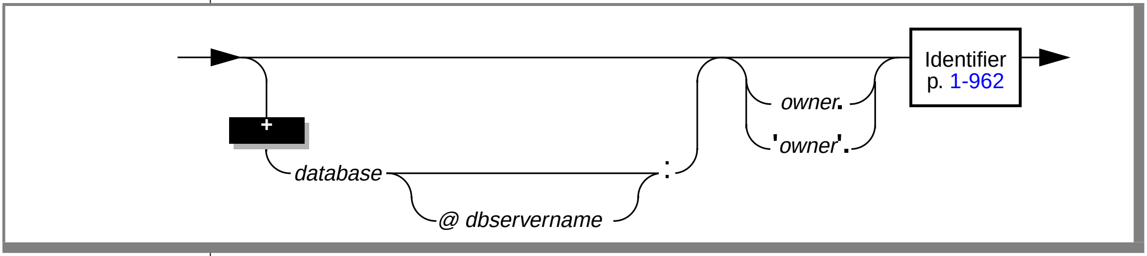
In this manual, see the CREATE FUNCTION and CREATE PROCEDURE statements.

In the *Informix Guide to SQL: Tutorial*, see Chapter 14 for information about how to create and use SPL routines.

Synonym Name

The Synonym Name segment specifies the name of a synonym. Use the Synonym Name segment whenever you see a reference to a synonym name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the synonym resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the Universal Server database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the synonym	If you are using an ANSI-compliant database, you must specify the owner for a synonym that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the synonym is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of synonyms. For more information, see the *Guide to GLS Functionality*. ♦

If you are creating the synonym, the *name* of the synonym must be unique within a database. The *name* cannot be the same as table names, temporary table names, or view names. It is possible to have a public and private synonym with the same name.

ANSI

If you are creating the synonym, the combination *owner.name* must be unique within a database.

The owner name is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored in uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

References

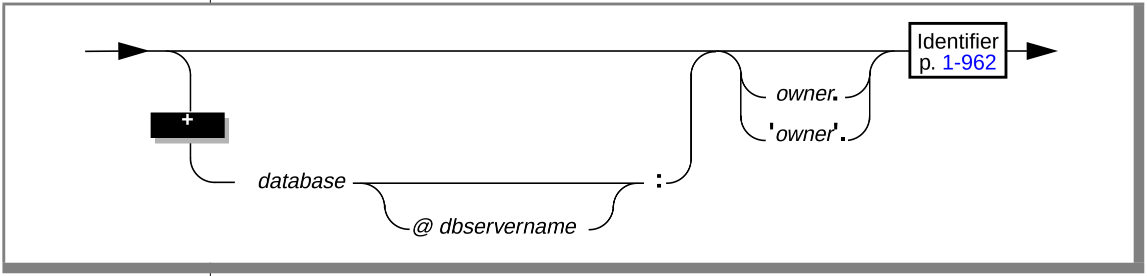
See the CREATE SYNONYM statement in this manual for information on creating synonyms.

In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 11.

Table Name

The Table Name segment specifies the name of a table. Use the Table Name segment whenever you see a reference to a table name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the table resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the Universal Server database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the table	If you are using an ANSI-compliant database, you must specify the owner for a table that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters. In SELECT statements and other statements that access tables in an ANSI-compliant database, the table owner that you specify must exactly match the actual owner of the table. See "Case Sensitivity in ANSI-Compliant Databases" on page 1-1045 for further information on this restriction.	The user name must conform to the conventions of your operating system.

Usage

The name of a table is an SQL identifier. The following example shows a table specification:

```
empinfo@personnel:emp_names
```

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of tables. For more information, see the *Guide to GLS Functionality*. ♦

If you are creating or renaming a table, the *name* of the table must be unique among all the tables, synonyms, temporary tables, and views that already exist in the database.

ANSI

If you are creating or renaming a table, you must make sure that the combination of *owner* and *name* is unique within a database.

In an ANSI-compliant database, the table name must include *owner.* unless you are the owner. For system catalog tables, the owner is **informix**. ♦

Case Sensitivity in ANSI-Compliant Databases

ANSI

The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in quotes. Put quotes around the owner portion of a name if you want the owner to be read exactly as written. In the following example, the name **cathl** in the first statement is upshifted to **CATHL** before it is used; the name **nancy** in the second statement is not upshifted:

```
SELECT * FROM cathl.customer

SELECT * FROM 'nancy'.customer
```

No problem exists if you create a table with an implicit owner in uppercase letters and the owner's real login name is also in uppercase letters. For example, suppose that you are the user **BROWN,** and you create a view with the following statement:

```
CREATE VIEW newcust AS
    SELECT fname, lname FROM customer WHERE state = 'NJ'
```

You, **BROWN**, can run the following SELECT statements on the view:

```
SELECT * FROM brown.newcust

SELECT * FROM newcust

SELECT * FROM systables WHERE tabname = newcust
    AND owner = USER
```

In the first query in the preceding example, the database server automatically upshifts **brown** before the SELECT statement executes. In the second query, the database server returns the owner name **BROWN** already upshifted. In the third query, USER returns the login name as it is stored—in this case, in uppercase letters. If you are the 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'
```

If you are **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
```

♦

References

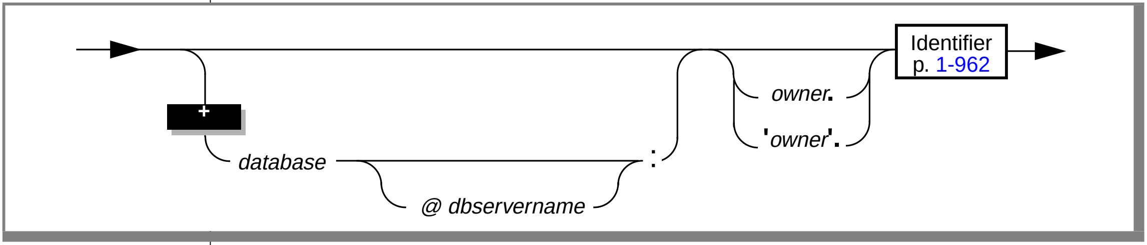
See the CREATE TABLE statement in this manual for information on creating tables.

In the *Informix Guide to SQL: Tutorial*, see the discussion of tables in Chapter 9.

View Name

The View Name segment specifies the name of a view. Use the View Name segment whenever you see a reference to a view name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the view resides	The database must exist.	Database Name, p. 1-852
dbservername	The name of the Universal Server database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-852
owner	The user name of the owner of the view	If you are using an ANSI-compliant database, you must specify the owner for a view that you do not own. If you put quotation marks around the name you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The name of a view is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of views. For more information, see the *Guide to GLS Functionality*. ♦

The use of the prefix *owner.* is optional; however, if you use it, the database server does check *owner* for accuracy. If you are creating a view, the *name* of the view must be unique among all the tables, synonyms, temporary tables, and views that already exist in the database.

ANSI

If you are creating a view, the *owner.view-name* must be unique among all the tables, synonyms, and views that already exist in the database.

The owner name is case sensitive. 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. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-1045. ♦

References

See the CREATE VIEW statement in this manual for information about how to create views.

In the *Informix Guide to SQL: Tutorial*, see the discussions of views in Chapter 11.

Chapter

2

SPL Statements

You can use Stored Procedure Language (SPL) statements to write routines, and you can store these SPL routines in the database. These 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

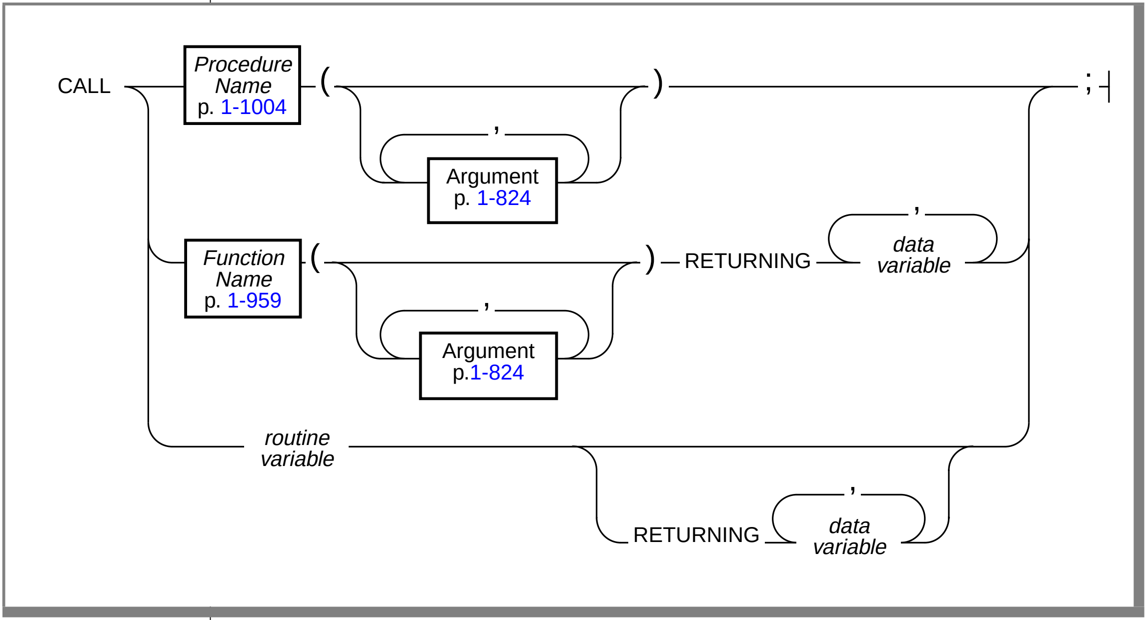
If a statement is composed of multiple clauses, the statement description provides the same set of information for each clause.

For task-oriented information about using SPL routines, see Chapter 14 of the *Informix Guide to SQL: Tutorial*.

CALL

Use the CALL statement to execute a routine from within an SPL routine.

Syntax



Element	Purpose	Restrictions	Syntax
data variable	The name of a variable that receives the value or values a function returns	The data type of *data variable* must be appropriate for the value the function returns.	Identifier, p. 1-962
routine variable	The name of a variable that is set to the name of an SPL routine or external routine	The *routine variable* must have the data type CHAR, VARCHAR, NCHAR, or NVARCHAR. The routine name you assign to *SPL variable* must be non-null and the name of an existing routine.	Identifier, p. 1-962

Usage

The CALL statement invokes one of the following:

- A procedure named *procedure name*
- A function named *function name*
- Any routine named by *routine variable*

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 you place the statement within an SPL routine executed by the program or DB-Access.

If you use CALL with a procedure name, you *cannot* specify a RETURNING clause. If you use CALL with a function name, you *must* specify a RETURNING clause.

Specifying Arguments

Specifying arguments is optional, whether you use CALL with a procedure or a function.

When you use CALL to execute a routine, you have the option of specifying parameter names for the arguments you pass to the routine. For example, each of the following examples is valid for a routine that has three parameters of VARCHAR type, 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 of specifying arguments is described in more detail in the Argument segment on page 1-824.

Receiving Input from the Called Routine

The RETURNING clause, used only with functions, specifies the *data variable* that receives the values the function returns. If you are calling a procedure, do not use the RETURNING clause.

The following example shows two routine calls.

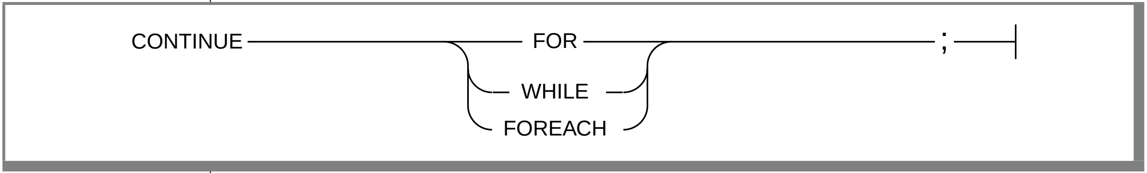
```
CREATE PROCEDURE not_much()

    DEFINE i, j, k INT;
    CALL no_args (10,20);
    CALL yes_args (5) RETURNING i, j, k;

END PROCEDURE
```

One routine call is to a procedure (**no_args**), which 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**.

CONTINUE

Use the CONTINUE statement to start the next iteration of the innermost FOR, WHILE, or FOREACH loop.

Syntax



Usage

When you encounter a CONTINUE statement, the 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 SPL function **loop_skip** 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 i WITH RESUME statement.

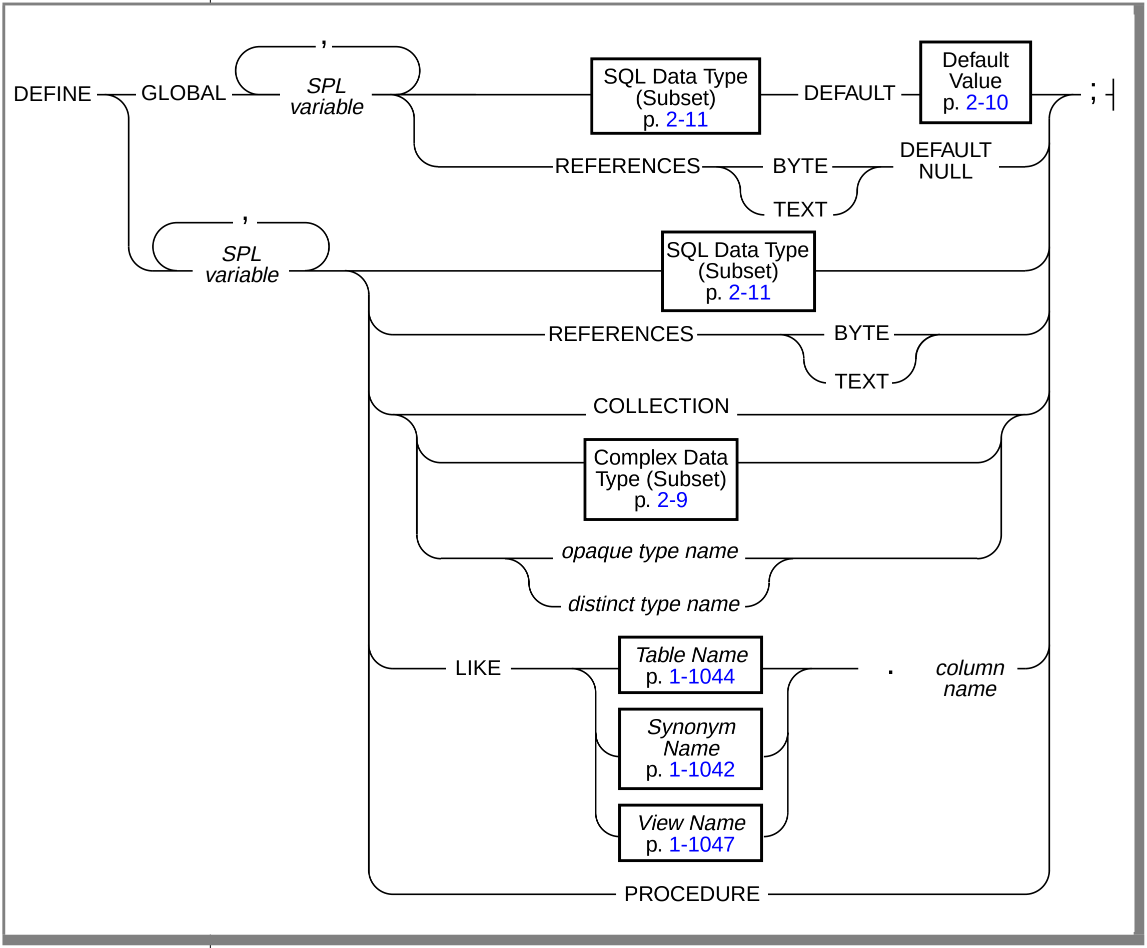
```
CREATE FUNCTION loop_skip()
    RETURNING INT;
    DEFINE i INT;
    .
    .
    .
    FOR i IN (3 TO 15 STEP 2)
    INSERT INTO testtable values(i, null, null);
        IF i = 11
            CONTINUE FOR;
        END IF;
        RETURN i WITH RESUME;
    END FOR;

END FUNCTION;
```

The CONTINUE statement generates errors if it cannot find the identified loop.

DEFINE

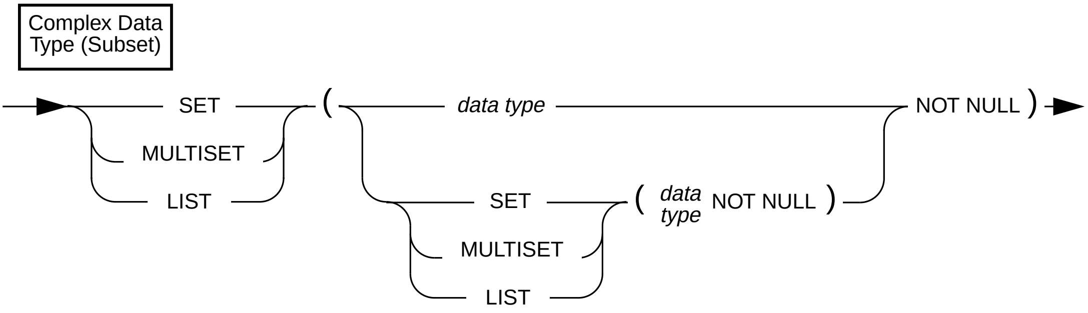
Use the DEFINE statement to declare variables that an SPL routine uses and to assign them data types.

Syntax



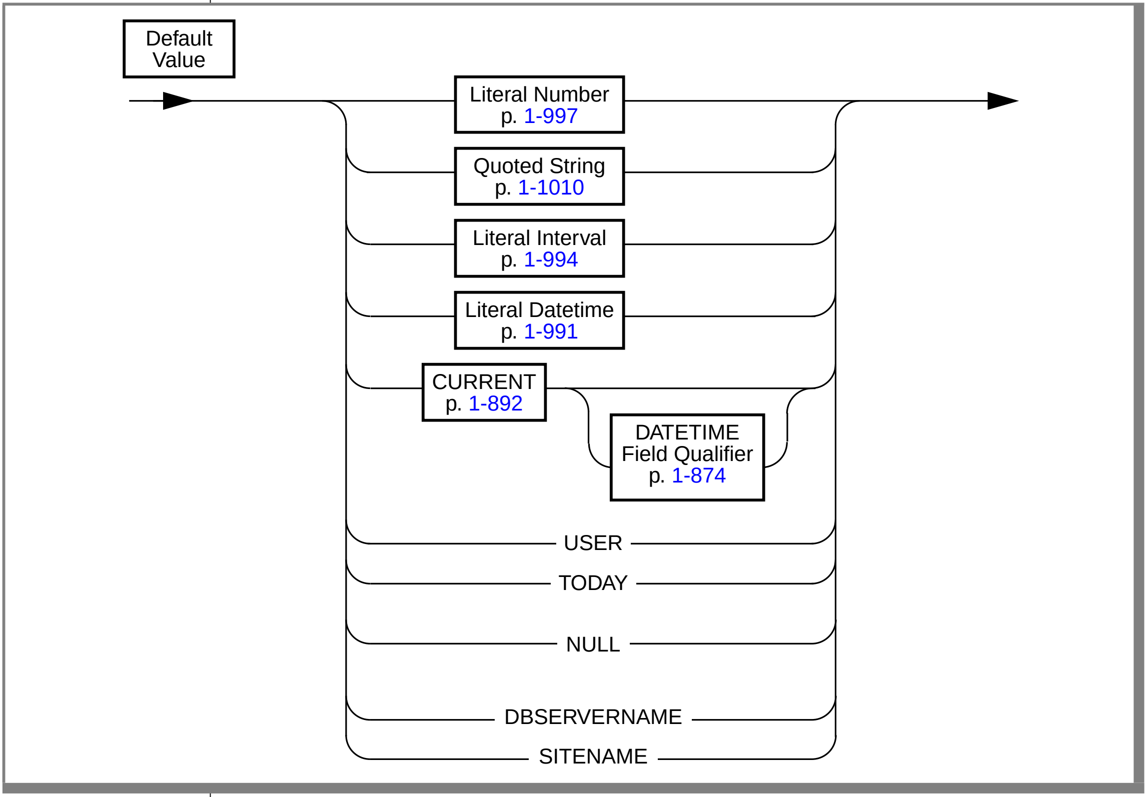
Element	Purpose	Restrictions	Syntax
column name	The name of a column in the table.	The column must exist in the table by the time you run the SPL routine.	Identifier, p. 1-962
distinct type name	The name of a distinct type.	The distinct type must be defined in the database by the time you run the SPL routine.	Identifier, p. 1-962
opaque type name	The name of an opaque type.	The opaque type must be defined in the database by the time you run the SPL routine.	Identifier, p. 1-962
SPL variable	The name of the SPL variable that is being defined.	The name must be unique within the statement block.	Identifier, p. 1-962

Complex Data Type (Subset)



Element	Purpose	Restrictions	Syntax
data type	The data type of the elements of a collection	The data type must match the data type of the elements of the collection the variable will contain. The data type can be any data type except a SERIAL, SERIAL8, TEXT, BYTE, CLOB, or BLOB.	Identifier, p. 1-962

Default Value Clause



Usage

The DEFINE statement is not an executable statement. It must appear after the routine header and before any other statements.

If you define a local variable (by using DEFINE without the GLOBAL keyword), the scope of the variable is the statement block in which it is defined. You can use the variable anywhere within the statement block. You can also use the same variable name outside the statement block with a different definition.

If you define a variable with the GLOBAL keyword, the variable is global in scope and is available outside the statement block and to other routines.

SQL Data Type Subset

In defining variables, the SQL data type subset includes all the SQL data types except SERIAL, SERIAL8, TEXT, BYTE, CLOB, and BLOB.

Variables of INT data type hold data from SERIAL columns, and variables of INT8 data type hold data from SERIAL8 columns.

Referencing TEXT and BYTE Variables

The REFERENCES keyword lets you use TEXT and BYTE variables. TEXT and BYTE variables do not contain the actual data but are simply pointers to the data. The REFERENCES keyword is a reminder that the SPL variable is just a pointer. Use an SPL variables that references a TEXT or BYTE data type exactly as you would any other variable.

You cannot define a variable that references a CLOB or BLOB data type.

Declaring GLOBAL Variables

The GLOBAL keyword indicates that the variables that follow are available to other routines through the global environment. The global environment is the memory that is used by all the routines that run within a given DB-Access or SQL API session.

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 data types of global variables you use in your SPL routine must match the data types of variables in the global environment.

Databases do not share global variables. The database server does not share global variables with application development tools.

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 procedure has defined the global variable **gl_out**:

```
CREATE PROCEDURE proc1()
    .
    .
    .
    DEFINE GLOBAL gl_out INT DEFAULT 13;
    .
    .
    .
    LET gl_out = gl_out + 1;
    END PROCEDURE;

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 the value of **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.

Providing Default Values

You can provide a literal value or a null value as the default for a global variable. You can also use a call to an SQL function to provide the default value. The following example uses the SITENAME function to provide a default value. It also defines a global BYTE variable.

```
CREATE PROCEDURE gl_def()
    DEFINE GLOBAL gl_site CHAR(18) DEFAULT SITENAME;
    DEFINE GLOBAL gl_byte REFERENCES BYTE DEFAULT NULL;
    .
    .
    .
END PROCEDURE
```

SITENAME or DBSERVERNAME

If you use the value returned by SITENAME or DBSERVERNAME as the default, the variable must be a CHAR, VARCHAR, NCHAR, or NVARCHAR value of at least 18 characters.

USER

If you use USER as the default, the variable must be a CHAR, VARCHAR, NCHAR, or NVARCHAR value of at least 8 characters.

CURRENT

If you use CURRENT as the default, the variable must be a DATETIME value. If the YEAR TO FRACTION keyword has qualified your variable, you can use CURRENT without qualifiers. If your variable uses another set of qualifiers, you must provide the same qualifiers when you use CURRENT as the default value. The following example defines a DATETIME variable with qualifiers and uses CURRENT with matching qualifiers:

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
        DEFAULT CURRENT YEAR TO MONTH;
```

TODAY

If you use TODAY as the default, the variable must be a DATE value.

TEXT and BYTE

The only default value possible for a TEXT or BYTE 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
```

Declaring Local Variables

Most local variables (that is, nonglobal variables) do not allow defaults. The following example shows 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;
```

Declaring Collection Variables

A variable of type COLLECTION, SET, MULTISET, and LIST is a collection variable and holds a collection fetched from the database. You cannot define a collection variable as global (with the GLOBAL keyword) or with a default value.

A variable defined with the type COLLECTION is an untyped collection variable. An untyped collection variable is generic and can hold a collection of any 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 type.

You must define the elements of a typed collection variable as NOT NULL, 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 );
```

Note that 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, s within collection and row types, and so on.

If you define a variable of COLLECTION type, the variable acquires varying type assignments if it is reused 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.

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 specific 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 allowed expression described in "Expression" on page 1-876.

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 gives *SPL 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 Variables of Type PROCEDURE

The PROCEDURE type indicates that in the current scope, *SPL variable* is a call to an SPL routine or external routine. In this release of INFORMIX-Universal Server, the DEFINE statement does not have a FUNCTION keyword. Use the PROCEDURE keyword, whether you are calling a procedure or function.

Defining a variable of PROCEDURE type indicates that in the current statement scope, *SPL variable* is not a call to an SQL function or a system function. For example, the following statement defines **length** as an SPL routine, not as the SQL LENGTH function:

```
DEFINE length PROCEDURE;
.
.
.
LET x = length (a,b,c)
```

This definition disables the SQL LENGTH function within the scope of the statement block. You would use such a definition if you had already created an SPL routine or an external routine with the name **length**.

If you create a 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 *SPL variable* is not a BYTE or TEXT value but a pointer to the BYTE or TEXT value. However, you can use the variable as though it holds the data.

The following example defines a local BYTE variable:

```
CREATE PROCEDURE use_blob()

    DEFINE i INT;
    DEFINE l_blob REFERENCES BYTE;

END PROCEDURE --use_blob
```

If you pass a variable of TEXT or BYTE 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.

You cannot declare a variable to hold or reference a CLOB or BLOB data type.

Redeclaring Variables

If you define the same variable twice within the same statement block, you receive an error. However, you can redefine a variable within a nested block, in which case it temporarily hides the outer declaration. The following example produces an error:

```
CREATE PROCEDURE example1()

    DEFINE n INT;
    DEFINE j INT;
    DEFINE n CHAR (1); -- redefinition produces an error
    .
    .
    .
END PROCEDURE;
```

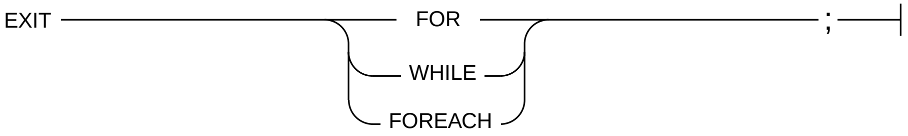
The database server allows the redeclaration in the following example. Within the nested statement block, n is a character variable. Outside the block, n is an integer variable.

```
CREATE PROCEDURE example2()

    DEFINE n INT;
    DEFINE j INT;
    .
    .
    .
    BEGIN
    DEFINE n CHAR (1);  -- character n masks integer variable
                        -- locally
    .
    .
    .
END PROCEDURE;
```

EXIT

Use the EXIT statement to stop the execution of a FOR, WHILE, or FOREACH loop.

Syntax



Usage

The EXIT statement causes the innermost loop of the indicated type (FOR, WHILE, or FOREACH) to terminate. Execution resumes at the first statement outside the loop.

If the EXIT statement cannot find the identified loop, it fails.

If the EXIT statement is used outside all loops, it generates errors.

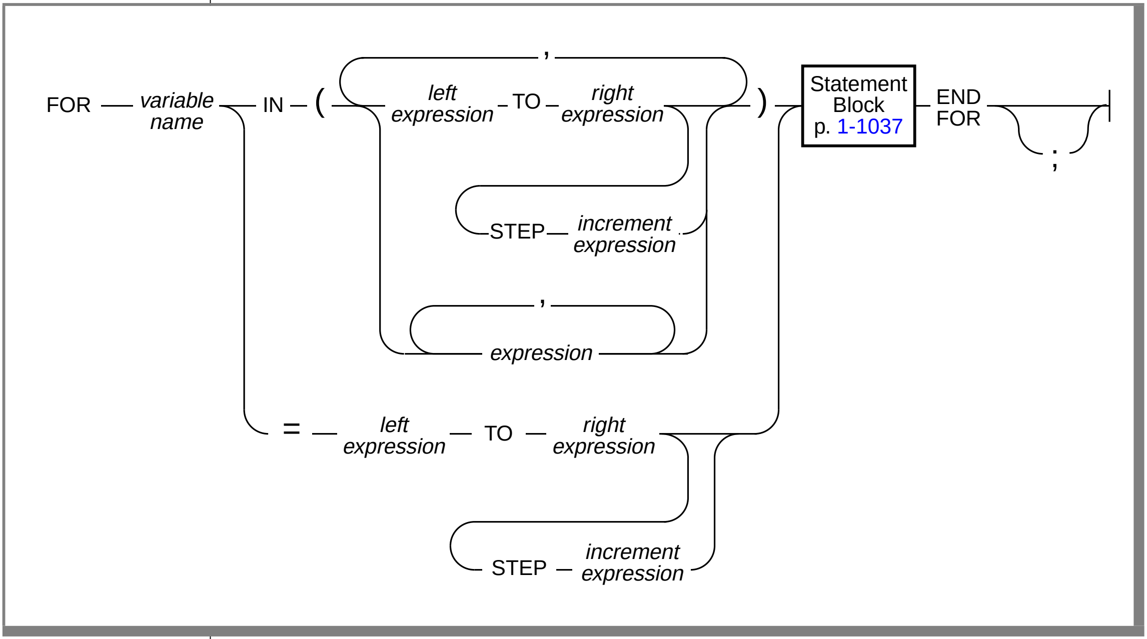
The following example uses an EXIT FOR statement. In the FOR loop, when j becomes 6, the IF condition i = 5 in the WHILE loop is true. The FOR loop stops executing, and the procedure continues at the next statement outside the FOR loop (in this case, the END PROCEDURE statement). In this example, the 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 establish a finite number of iterations for a loop.

Syntax



Element	Purpose	Restrictions	Syntax
expression	A numeric or character value against which *variable name* is compared to determine if the loop should be executed	The data type of *expression* must match the data type of *variable name*. You can use the output of a SELECT statement as an *expression*.	Expression, p. 1-876
increment expression	A positive or negative value by which *variable name* is incremented. Defaults to +1 or -1 depending on *left expression* and *right expression*.	The increment expression cannot evaluate to 0.	Expression, p. 1-876
left expression	The starting expression of a range	The value of *left expression* must match the data type of *variable name*. It must be either INT or SMALLINT.	Expression, p. 1-876
right expression	The ending expression in the range. The size of *right expression* relative to *left expression* determines if the range is stepped through positively or negatively.	The value of *right expression* must match the data type of *variable name*. It must be either INT or SMALLINT.	Expression, p. 1-876
variable name	The value of this variable determines how many times the loop executes.	You must have already defined this variable, and the variable must be valid within this statement block. If you are using *variable name* with a range of values and the TO keyword, you must define *variable name* explicitly as either INT or SMALLINT.	Identifier, p. 1-962

Usage

The database server computes all expressions before the FOR statement executes. If one or more of the expressions are variables, and their values change during the loop, the change has no effect on the iterations of the loop.

The FOR loop terminates when *variable name* takes on the values of each element in the expression list or range in succession or when it encounters an EXIT FOR statement.

The database server generates an error if an assignment within the body of the FOR statement attempts to modify the value of *variable name*.

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 expression* option implicitly sets the number of increments. If you use the TO keyword, *variable name* 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 expression* the value of -1 if *right expression* is less than *left expression*, or +1 if *right expression* is more than *left expression*. If *increment expression* 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 *variable name* to the value of *left expression*. In subsequent iterations, the server adds *increment expression* to the value of *variable name* and checks *increment expression* to determine whether the value of *variable name* 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 *variable name* to the value of the first expression specified. In subsequent iterations, *variable name* takes on the value of the next expression. When the 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

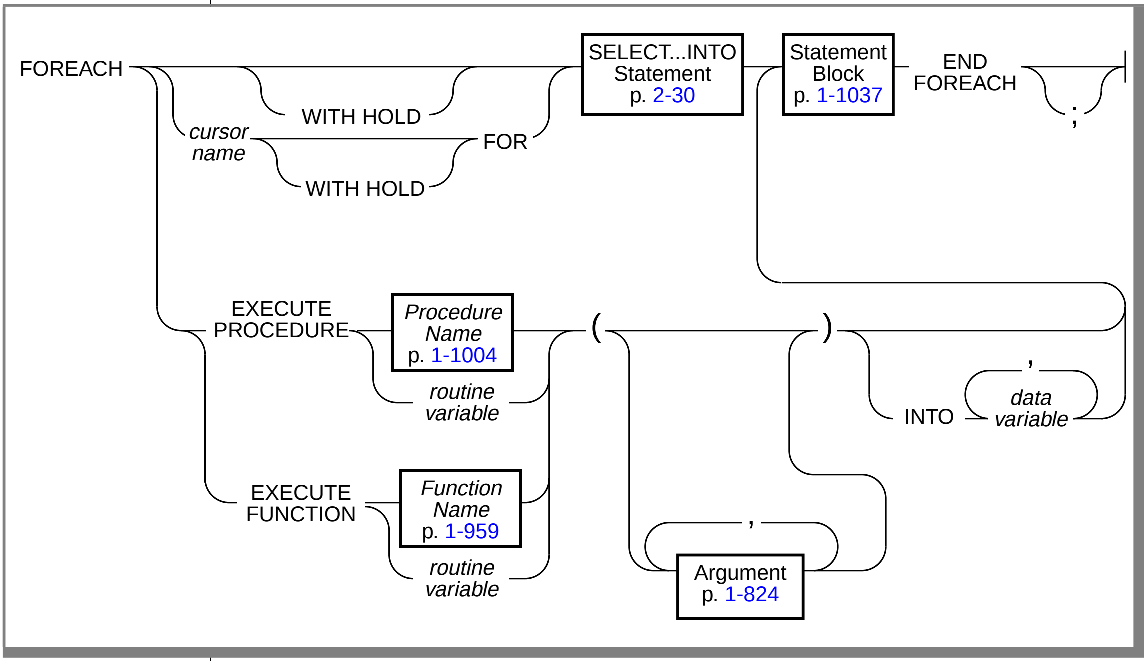
If *variable name* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture that uses an integer variable. Values in the expression list include the value that is returned from a SELECT statement, a sum of an integer variable and a constant, the values that are returned from an SPL procedure 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
```

FOREACH

Use a FOREACH loop to select and handle a set of rows or a collection.

Syntax



Element	Purpose	Restrictions	Syntax
cursor name	An identifier that you supply as a name for the FOREACH loop	Each cursor name within a routine must be unique.	Identifier, p. 1-962
data variable	The name of an SPL variable in the calling routine that will receive the value or values the called function returns	The data type of *variable name* must be appropriate for the value that is being returned.	Identifier, p. 1-962
routine variable	The name of an SPL variable in the calling routine that contains the name of a routine to be executed	The data type of *routine variable* must be CHAR, VARCHAR, NCHAR, NVARCHAR.	Identifier, p. 1-962

Usage

A FOREACH loop is the procedural equivalent of using a cursor. When Universal Server executes a FOREACH statement, it takes the following actions:

1. It declares and implicitly opens a cursor.
2. It obtains the first row, the first collection element, or the first set of return values (depending upon the syntax of your FOREACH statement).
3. It assigns each variable in the variable list 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, the next collection element, or the next set of return values, and it repeats steps 3 and 4.
6. It terminates the loop when it finds no more rows, the end of the collection, or the last set of return values. It closes the implicit cursor when the loop terminates.

Because the statement block can contain additional FOREACH statements, you can nest cursors. No limit exists to the number of cursors that can be nested.

An SPL routine that returns more than one row, collection element, or set of values is called a cursor routine. An SPL routine that returns only one row or value is a noncursor routine.

The following SPL procedure illustrates the three types of 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 more values.
- 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 the ON EXCEPTION statement on page 2-43.)
- 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

With an ordinary cursor that fetches a set of rows or values, you can use a SELECT ... INTO statement. With an ordinary cursor, SELECT ... INTO can also include the UNION and ORDER BY clauses, but it cannot include the INTO TEMP clause. The syntax of a SELECT statement is shown on page 1-593.

The 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 keyword specifies that the cursor should remain open when a transaction closes (that is, is committed or rolled back).

Updating or Deleting Rows Identified by Cursor Name

Use the WHERE CURRENT OF *cursor name* clause to update or delete the current row of *cursor name*.

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*. 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.

Tip: *To access only one element of a collection variable, you do not need to declare a cursor. For information on how to select a single element, see "Selecting From a Collection Variable" on page 1-610***.** *For information on how to insert a single element, see "Inserting Into a Collection Variable" on page 1-506.*

If you are using a collection cursor to fetch individual elements from a collection variable, you must use a restricted form of the SELECT statement called a *collection query.* The collection query 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.*
- You must use a collection query within a FOREACH loop.
- You cannot use the WITH HOLD option on the FOREACH statement.
- The data type of the element variable must be the same as the element type of the collection.
- The element variable can have 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 field list must be an asterisk (*).
- If the collection contains row types, the select field list can be a list of one or more field names.
- The select field list cannot contain an expression.
- The collection query cannot specify a WHERE clause, a HAVING clause, a GROUP BY clause, or an ORDER BY clause.

The following excerpt from an SPL routine shows a collection query within a FOREACH loop:

```
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 within the FOREACH loop is the collection query. The statement selects one element at a time from the collection variable **b** into the element variable **a**.

In the collection query, the select field 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 on using a Collection Derived Table, see page 1-827.

The next example shows a collection query that uses a list of row type fields in its select field 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;
.
.
```

In this example, 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 names **e** as an alias for the collection derived table **employees**. The alias exists as long as the SELECT statement executes.

To update an element of a collection, 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. For more information on selecting from a collection variable, see the SELECT statement on page 1-593.

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 Routine in the FOREACH Loop

In general, use the following guidelines for calling another routine from an SPL routine:

- To call an SPL procedure or external procedure, use EXECUTE PROCEDURE *procedure name*.
- To call an SPL function or external function, use EXECUTE FUNCTION *function name*.
- Do not use EXECUTE FUNCTION *procedure name* in any case.

Important: *Universal Server allows you to use an EXECUTE PROCEDURE statement to execute SPL functions (routines that were formerly called stored procedures that return a value). However, Informix recommends that you use the EXECUTE PROCEDURE statement only with procedures and the EXECUTE FUNCTION statement only with functions.*

If you use EXECUTE PROCEDURE, Universal Server looks first for a procedure of the name you specify. If it finds the procedure, the server executes it. If Universal Server does not find the procedure, it looks for a function of the same name to execute. If the server finds neither a function nor a procedure, it issues an error message.

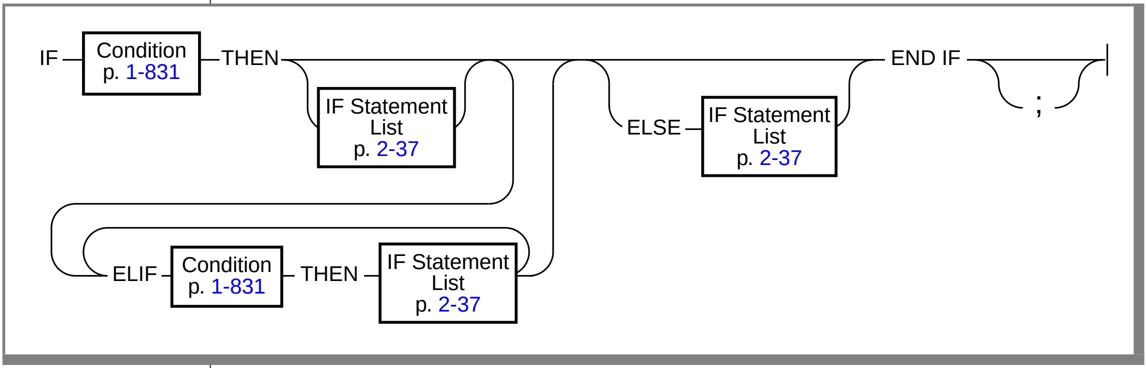
If you use EXECUTE FUNCTION, Universal Server looks for a function of the name you specify. If it does not find a function of that name, the server issues an error message.

A called function can return zero or more values or rows. The type and count of each variable in the variable list must match each value that the function returns.

IF

Use an IF statement to create a branch within an SPL routine.

Syntax


IF
Condition p. 1-831
THEN
IF Statement List p. 2-37
ELSE
IF Statement List p. 2-37
END IF
;
ELIF
Condition p. 1-831
THEN
IF Statement List p. 2-37

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.

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 and 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 0 is returned.

```
CREATE FUNCTION str_compare (str1 CHAR(20), str2 CHAR(20))
    RETURNING INT;

    DEFINE result INT;

    IF str1 > str2 then
        result =1;
    ELIF str2 > str1 THEN
        result = -1;
    ELSE
        result = 0;
    END IF
    RETURN result;

END FUNCTION -- str_compare
```

The ELIF Clause

Use the ELIF clause to specify one or more additional conditions to evaluate.

If you specify an ELIF clause, and the IF condition is false, the ELIF condition is evaluated. If the ELIF condition is true, the statements that follow the ELIF clause execute.

The ELSE Clause

The ELSE clause executes if no true previous condition exists in the IF clause or any of the ELIF clauses.

Conditions in an IF Statement

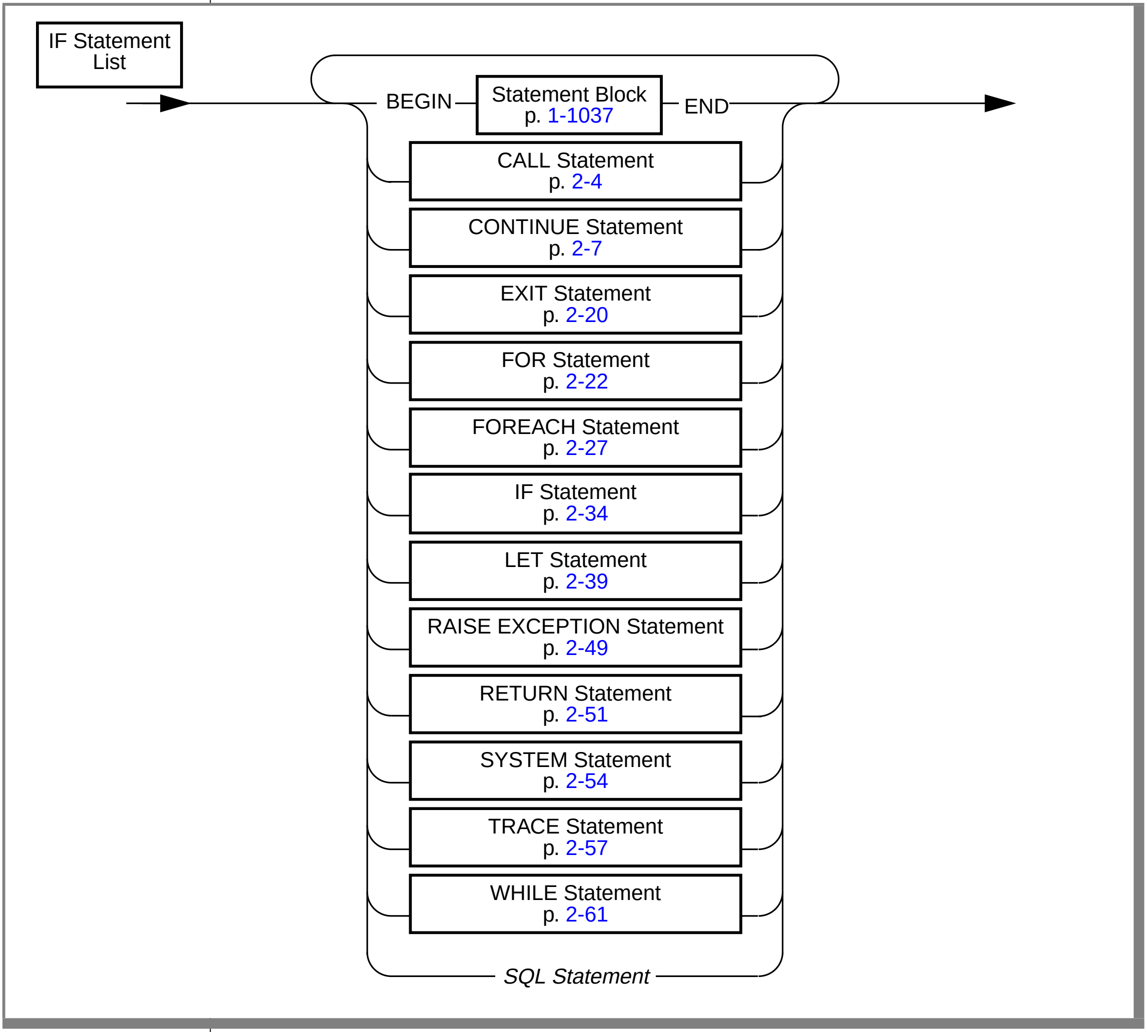
Conditions in an IF statement are evaluated in the same way as conditions in a WHILE statement.

If any expression that the condition contains evaluates to null, the condition automatically becomes untrue. 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.

An expression within the condition that has an UNKNOWN value (due to the use of an uninitialized variable) causes an immediate error. The statement terminates and raises an exception.

IF Statement List



Subset of SQL Statements Allowed in an IF Statement

You can use any SQL statement in the statement block except the ones in the following list.

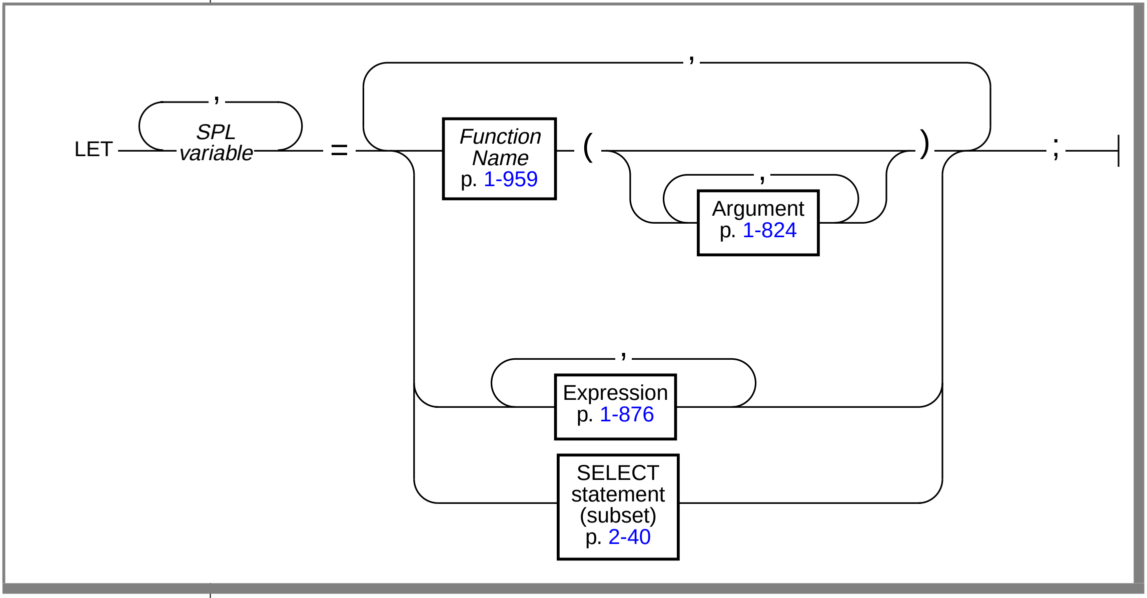
ALLOCATE DESCRIPTOR
CHECK TABLE
CLOSE DATABASE
CONNECT
CREATE DATABASE
CREATE PROCEDURE
DATABASE
DEALLOCATE DESCRIPTOR
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
INFO
LOAD
OPEN
OUTPUT
PREPARE
PUT
REPAIR TABLE
ROLLFORWARD DATABASE
SET CONNECTION
SET DESCRIPTOR
START DATABASE
UNLOAD
WHENEVER

You can use a SELECT statement only if you use the INTO TEMP clause to put the results of the SELECT statement into a temporary table.

LET

Use the LET statement to assign values to variables or to call a function from an SPL routine and assign the return value or values to variables.

Syntax



Element	Purpose	Restrictions	Syntax
SPL variable	An SPL variable that receives the value the function returns or is set to the result of the expression	The SPL variable must be defined in the routine and must be valid in the statement block.	Identifier, p. 1-962

Usage

If you assign a value to a single variable, you make a *simple assignment*. If you assign values to two or more variables, you make a *compound assignment*.

You can also assign the value of an expression to a variable. At runtime, the value of the SPL expression is computed first. The resulting value is cast to the data type of *variable*, if possible, and the assignment occurs. If conversion is not possible, an error occurs, and the value of *variable name* is undefined.

A compound assignment assigns multiple expressions to multiple variables. The data types of expressions in the expression list does 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 *Informix Guide to SQL: Reference*.)

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 operate on other values. For example, the following statement is illegal:

```
LET a,b = (c,d) + (10,15); -- ILLEGAL EXPRESSION
```

Using a SELECT Statement in a LET Statement

The examples in this section use a SELECT statement within a LET statement. You can use a SELECT statement to assign values to one or more variables on the left side of the = operator, as the following examples show:

```
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 illegal:

```
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 more information on FOREACH, see page 2-27.

Calling a Function in a LET Statement

You can call an SPL function or an external function in a LET statement and assign the returned value or values to variables. You must specify all the necessary arguments to the function unless the function's arguments have default values.

If you name one of the parameters in the called function, with syntax such as **name = 'smith'**, you must name all the parameters.

The named variable receives the returned value from a function call. A function can return more than one value into a list of variable names. However, you must enclose a function that returns more than one value in a FOREACH loop.

The following example shows several LET statements. The first two are valid LET statements that contain function calls. The third LET statement is not legal, 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 legal LET statements.

```
LET a, b, c = proc1(name = 'grok', age = 17);
LET a, b, c = 7, proc2('orange', 'green');

LET a, b = proc1() + proc2(); -- ILLEGAL CODE
```

A function called in a LET statement can have an argument of COLLECTION, SET, MULTISET, or LIST. COLLECTION is a generic collection data type that includes SET, MULTISET, and LIST collection types.

You can then assign the value returned by the function to a variable, for example:

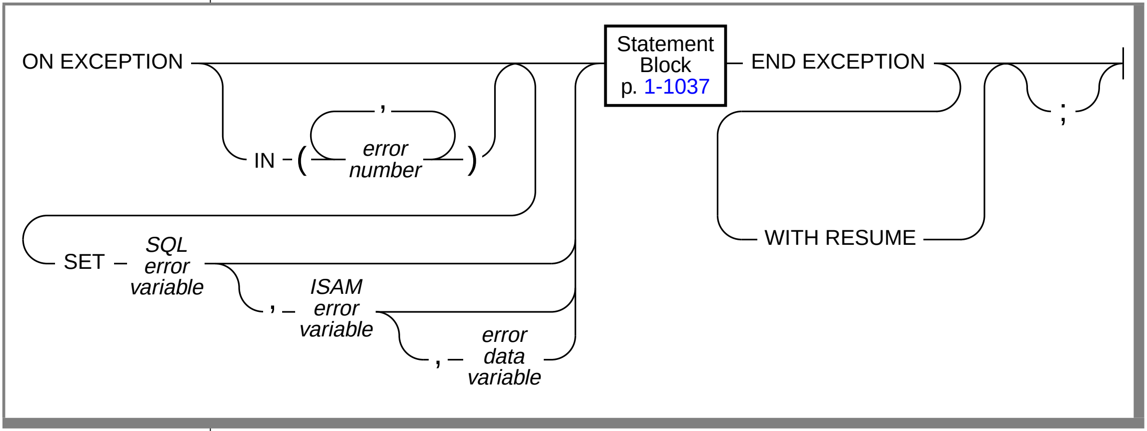
```
LET d = function1(collection1);
LET a = function2(set1);
```

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 that are taken for a particular error or a set of errors.

Syntax



Element	Purpose	Restrictions	Syntax
error data variable	An SPL variable that contains a string returned by an SQL error or a user-defined exception	The variable must be a character data type to receive the error information. It must be valid in the current statement block.	Identifier, p. 1-962
error number	An SQL error number, or an error number created by a RAISE EXCEPTION statement, that is to be trapped	The error number must be of integer data type. It must be valid in the current statement block.	Literal number, p. 1-997
ISAM error variable	A variable that receives the ISAM error number of the exception raised	The error variable must be of integer data type. It must be valid in the current statement block.	Identifier, p. 1-962
SQL error variable	A variable that receives the SQL error number of the exception raised	The error variable must be a character data type. It must be valid in the current statement block.	Identifier, p. 1-962

Usage

The ON EXCEPTION statement, together with the RAISE EXCEPTION statement, provides an error-trapping and error-recovery mechanism for SPL. The ON EXCEPTION statement defines a list of errors that are to be trapped as the SPL routine executes and specifies the action (within the statement block) to take when the trap is triggered. If the IN clause is omitted, all errors are trapped.

You can use more than one ON EXCEPTION statement within a given statement block.

The scope of an ON EXCEPTION statement is the statement block that follows the ON EXCEPTION statement, all the statement blocks that are nested within that following statement block, and all the statement blocks that follow the ON EXCEPTION statement.

The exceptions that are trapped can be either system- or user-defined.

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 returns to the variable. If you specify a variable to receive the returned error text, but none exists, an empty string goes into the variable.

Placement of the ON EXCEPTION Statement

ON EXCEPTION is a declarative statement, not an executable statement. For this reason, you must use the ON EXCEPTION statement before any executable statement and after any DEFINE statement in an SPL routine.

The following example shows the correct placement of an ON EXCEPTION statement. Use an ON EXCEPTION statement after the DEFINE statement and before the body of the routine.

The following SPL function inserts a set of values into a table. If the table does not exist, it is created, and the values are inserted. The SPL 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 particular 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 routine, application, or interactive user), and execution aborts.

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 is necessary so that other DELETEs
             -- don't 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

A trap is triggered if either the SQL error code or the ISAM error code matches an exception code in the list of error numbers. The search through the list 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 to set up default trapping. A summary of the sequence of statements in the following example would be: “Test for an error. If error -210, -211, or -212 occurs, take action A. If error -300 occurs, take action B. If any other error occurs, take action C.” 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
.
.
.
```

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 provided an *error data variable*, any error text that the database server returns is put into the *error data variable*. Error text includes information such as the offending table or column name.

Forcing the Routine to Continue

The example on page 2-45 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 routine, the result is that the count of salespeople names occurs even if the error occurred.

Continuing Execution After an Exception Occurs

If you do not include the WITH RESUME keyword in your ON EXCEPTION statement, 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 the ON EXCEPTION statement is not contained within a statement or statement block, the SPL routine executes a RETURN statement with no arguments to terminate. That is, the routine returns a successful status and no values.

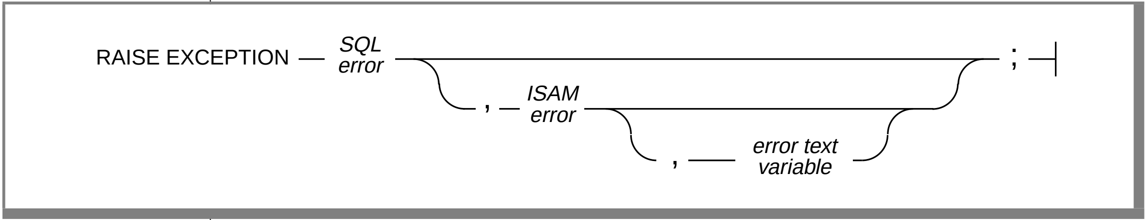
Errors Within the ON EXCEPTION Statement Block

To prevent an infinite loop, if an error occurs during execution of the statement block of an error trap, the search for another trap does not include the current trap.

RAISE EXCEPTION

Use the RAISE EXCEPTION statement to simulate the generation of an error.

Syntax



Element	Purpose	Restrictions	Syntax
error text variable	An SPL variable that contains the error text	The SPL variable must be a character data type and must be valid in the statement block.	Identifier, p. 1-962
ISAM error	A variable or expression that represents an ISAM error number. The default value is 0.	The variable or expression must evaluate to a SMALLINT value. You can place a minus sign before the error number.	Expression, p. 1-876
SQL error	A variable or expression that represents an SQL error number	The variable or expression must evaluate to a SMALLINT value. You can place a minus sign before the error number.	Expression, p. 1-876

Usage

Use the RAISE EXCEPTION statement to simulate an error. An ON EXCEPTION statement can trap the generated error.

If you omit the *ISAM error* parameter, the database server sets the ISAM error code to zero when the exception is raised. If you want to use the *error text variable* parameter but not specify a value for *ISAM error*, you can specify 0 as the value of *ISAM error*.

For example, the following statement raises the error number -9999 and returns the stated text:

```
RAISE EXCEPTION -9999, 0, 'You broke the rules';
```

The statement can raise either system-generated exceptions or user-generated exceptions.

In the following example, a negative value for a raises exception 9999. The code should contain an ON EXCEPTION statement that traps for an exception of 9999.

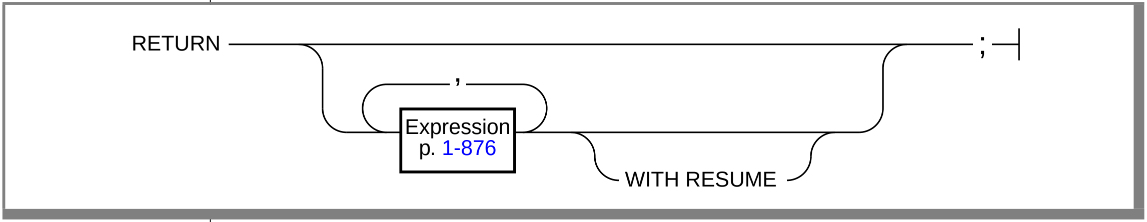
```
FOREACH SELECT c1 INTO a FROM t
IF a < 0 THEN
RAISE EXCEPTION 9999-- emergency exit
END IF
END FOREACH
```

See the ON EXCEPTION statement for more information about the scope and compatibility of exceptions.

RETURN

Use the RETURN statement to designate the values that the routine returns to the calling module.

Syntax



Usage

The RETURN statement returns values to the calling module. An SPL routine that returns one or more values is called an SPL function.

All the RETURN statements in an SPL function must be consistent with the RETURNING clause of the CREATE FUNCTION 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 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 routine or program expects one or more return values, it is equivalent to returning the expected number of null values to the calling program.

In the following example, the SPL function includes two acceptable 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 privs...
        RETURN;         -
            - return no values.
    END EXCEPTION;
    SELECT DISTINCT descript INTO des FROM stock
            WHERE stocknum = stockno;
    RETURN des;

END FUNCTION;
```

A RETURN statement without any expressions exits only if the SPL function is declared not to return values; otherwise it returns nulls.

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 functio*n is an external function that returns one or more rows of data and therefore requires a cursor to execute.

The 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. If a function executes a RETURN WITH RESUME statement, the calling routine or program must call the function within a FOREACH loop.

ESQL

If an SPL routine executes a RETURN WITH RESUME statement, a FETCH statement in an application that is written in an SQL API can call it. ♦

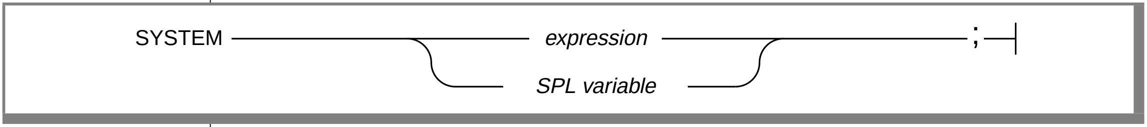
The following example shows a cursor routine that another routine can call. After the RETURN i WITH RESUME statement returns each value to the calling routine or program, the next line of **sequence()** executes the next time **sequence()** is called. If **backwards** equals 0, no value is returned to the calling routine or program, and execution of **sequence** stops.

```
CREATE FUNCTION sequence (limit INT, backwards INT)

    RETURNING INT;
    DEFINE i INT;

    FOR i IN (1 TO limit STEP 1)
        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 IF

END FUNCTION; -- sequence
```

SYSTEM

Use the SYSTEM statement to make an operating-system command run from within an SPL routine.

Syntax



Element	Purpose	Restrictions	Syntax
expression	Any expression that is a user-executable operating-system command	You cannot specify that the command run in the background.	Operating-system dependent
SPL variable	An SPL variable that contains a valid operating-system command	The SPL variable must be of CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING data type,	Identifier, p. 1-962

Usage

If the supplied expression is not a character expression, *expression* is converted to a character expression before the operating-system command is made. The complete character expression passes to the operating system and executes as an operating-system command.

The operating-system command that the SYSTEM statement 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.

Your SPL routine cannot use a value or values that the command returns.

If the operating-system command fails (that is, if the operating system 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.

In routines that contain SYSTEM statements, the operating-system command runs with the permissions of the user who is executing the routine.

Specifying Environment Variables in SYSTEM Statements

When the operating-system command that SYSTEM specifies is executed, no guarantee exists that the environment variables that the user application set are passed to the operating system. To ensure that the environment variables that the application set are carried forward to the operating system, enter a SYSTEM command that sets the environment variables before you enter the SYSTEM command that causes the operating-system command to execute.

For information on the operating-system commands that set environment variables, see Chapter 3 of the *Informix Guide to SQL: Reference*.

Examples of the SYSTEM Statement in SPL Routines

The following example shows the use of a SYSTEM statement within an SPL routine. The SYSTEM statement in this example causes the operating system to send a mail message to the system administrator.

```
CREATE PROCEDURE sensitive_update()
    .
    .
    .
    LET mailcall = 'mail headhoncho < alert'
    -- code that evaluates if operator tries to execute a
    -- certain command, then sends email to system
    -- administrator
    SYSTEM mailcall
    .
    .
    .
END PROCEDURE; -- sensitive_update
```

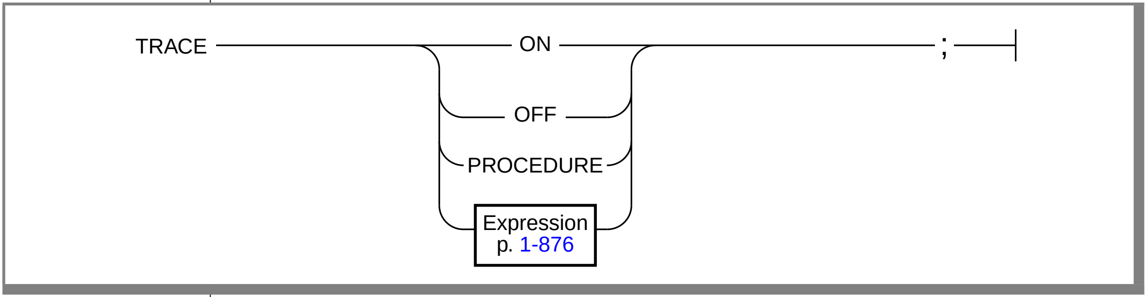
You can use a double-pipe symbol (||) to concatenate expressions with a SYSTEM statement, as the following example shows:

```
CREATE PROCEDURE sensitive_update2()
    DEFINE user1 char(15);
    DEFINE user2 char(15);
    LET user1 = 'joe';
    LET user2 = 'mary';
    .
    .
    .
    -- code that evaluates if operator tries to execute a
    -- certain command, then sends email to system
    -- administrator
    SYSTEM 'mail -s violation' ||user1 || ' ' || user2
                     || '< violation_file';
    .
    .
    .
END PROCEDURE;
```

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 prints the current values of the following items:

- 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 the SPL routine calls inherit the trace state. That is, a called routine 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.

Printing 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. However, you must first use the SET DEBUG statement to establish a trace-output file.

The following example uses a TRACE statement with an expression after it uses a TRACE OFF statement:

```
CREATE PROCEDURE tracing ()
    DEFINE i INT;
BEGIN
    ON EXCEPTION IN (1)
    END EXCEPTION; -- do nothing
    TRACE OFF;
    SET DEBUG FILE TO '/tmp/foo.trace';
    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
```

The following example shows additional TRACE statements:

```
PROCEDURE testproc()
    DEFINE i INT;

    TRACE OFF;
    SET DEBUG FILE TO '/tmp/test.trace';
    TRACE 'Entering foo';

    TRACE PROCEDURE;
    LET i = testtoo();

    TRACE ON;
    LET i = i + 1;

    TRACE OFF;
    TRACE 'i+1 = ' || i+1;
    TRACE 'Exiting testproc';

    SET DEBUG FILE TO '/tmp/test2.trace';

END PROCEDURE;
```

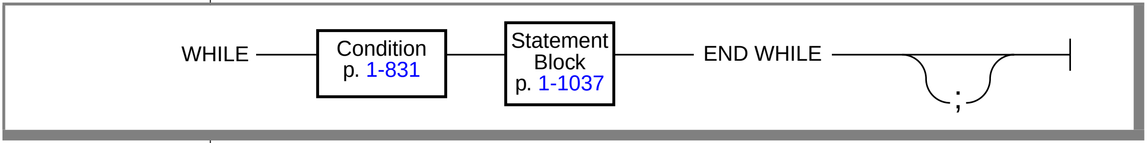
Looking at the Traced Output

To see the traced output, use an editor or utility to display or read the contents of the file.

WHILE

Use the WHILE statement to establish an indefinite loop within an SPL routine.

Syntax



Usage

The condition is evaluated once at the beginning of the loop, and subsequently at the beginning of each iteration. The statement block is executed 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 automatically becomes not true unless you are explicitly testing for the IS NULL condition.

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.

```
CREATE PROCEDURE simp_while()
    DEFINE i INT;
    DEFINE pf_name CHAR(15);
    WHILE EXISTS (SELECT fname INTO pf_name 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;
```

Index